University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Dragon: Processing Node Discovery Protocol Based on Static Attributes for Homogeneous and Heterogeneous Wireless Sensor Networks

Roman Kolcun

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, February 2015

Declaration

I declare that the work described in this thesis is my own and has not been submitted for any other degree or diploma. The work of others is properly referenced and the list is provided in bibliography.

Abstract

Wireless Sensor Networks (WSNs) are networks consisting of small, battery-powered computers with short-range radio communication and sensing capabilities. These computers (referred to as nodes) are used to sense one or more variables using one or more sensors and report these readings to a base-station via a multi-hop communication. Often, these WSNs are deployed to detect a phenomenon. Detection of this phenomenon usually depends on readings from several sensors in different locations. Therefore, sensor readings are periodically collected at the base-station which processes these data or forwards them to a cloud. This base-station also represents a gateway for users to access and communicate with the WSN. It allows a user to submit a query, whose execution retrieves data from relevant sensor nodes and the result of the computation over these data is detection of a phenomenon. In a typical node, radio is responsible for far more energy consumption when compared to the CPU or most of the sensors. Therefore, it has always been researchers' intention to lower the network communication to the lowest possible level. Because nodes closer to the base-station transfer more data, their batteries are depleted faster which may lead to part of the network being unreachable. Additionally, because a user accesses the WSN via a base-station, it represents a single point of failure. One of the solutions to overcome this problem is to allow a user to communicate and submit a query via *any* node in the network. However, building a fully decentralised and energy-efficient framework allowing any node to accept and execute a query submitted by a user brings several new challenges. First, a node needs to be able to communicate with any other node in the network, not only the base-station, without relying on any central entity. Second, any node must be able to identify all the nodes which monitor the same phenomenon. And third, a node which processes the data must be chosen in such way, that the overall communication of the whole network is minimised.

In this thesis we present Dragon, a framework for in-network data stream processing. Dragon allows communication among any pair of nodes via optimal or near optimal routes. This is achieved without the need to first discover or establish a path between two communicating nodes. Dragon also allows any node to find a list of all other nodes fulfilling given static criteria. The search for these nodes requires communication with only close (possibly multi-hop) neighbourhood. Finding a list of nodes observing the same phenomenon and requesting data directly from these nodes allows any node in the network to accept and execute a snapshot (one-time) query with a very low network overhead and in a timely manor. Finally, Dragon introduces a distributed algorithm for discovery of a processing node for continuous queries in WSNs. The algorithm follows the cost gradient to the node with the lowest communication cost, hence decreasing the overall network traffic and communication delay.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Julie A. McCann, who has guided me through my PhD and has always helped me with a support and good advice.

I am also grateful to my colleagues Pedro Martins, Michael Breza, Shusen Yang, Usman Adeel, and Evangelos Spyrou who have always been there for me whenever I needed an advice, help, or just to go for a beer.

I cannot express how grateful I am for my parents, Iraida and Michal, who have been showing me the beauty of research throughout my life.

Last, but not least I would like to express my deepest gratitude to my wife, Radka, who has always supported me in pursuing my dreams and has always stood by my side.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Wireless Sensor Networks

One of the basic discovery methods used by scientists from the early beginnings of mankind has been observation. Planets were discovered by observing the night sky, the fact that the Earth is round was discovered by observing the movement of the stars and the Sun. From early ages scientists developed new techniques and instruments which would help them with the observation of a given phenomenon. By monitoring the phenomenon, scientists tried to better understand the underlying causes. With the introduction of computers scientists gained a powerful tool for processing large quantities of data. However, collecting these data was a tedious process which often involved human intervention. As computers became more common, scientists started to use them not only to process the data but also to collect them. They attached sensors to computers to monitor the phenomenon in real time. This solution was sufficient for observing phenomenon occurring at large scale, e.g. monitoring atmospheric pressure. However, if scientists wanted to better understand distributed phenomena with spatio-temporal characteristics, e.g. how smoke spreads in a building during a fire, it required installation of many computers with sensors attached throughout the building. This approach was usually neither simple, nor feasible.

With the miniaturisation of computers, new types of small, battery-powered devices become available. These very small devices are capable of simple sensing, computation, and wireless communication (from now on referred to as *sensor nodes* or just *nodes*). They could be deployed by researchers in order to monitor a given phenomenon by sensing a variable at a predefined rate and send collected data to a central computer. These sensor nodes are capable of wireless communication which allow them to exchange data and talk to each other. Even though one single sensor node cannot do much, by organising all of the nodes into a network we can get a powerful observation tool capable of measuring distributed spatio-temporal phenomena.

We refer to these sensor nodes organised in a network as a Wireless Sensor Network (WSN from here, WSNs in the plural form). WSNs allow us to study the environment and distributed spatio-temporal phenomena within the environment in a distributed way. They can help us to better understand the causes and even allow us to control the environment.

To better demonstrate what a WSN can be used for, let us introduce a simple scenario. We will use this scenario throughout this thesis as we extend the capabilities of the WSN with new features allowing it to provide a user with richer capabilities.

Scenario:
A remote oil field in a desert consists of many oil pumps, oil reservoirs, and tens of miles of various pipes delivering a wide range of liquids or gases. These pipes create a complex interconnected system, i.e. a pipe can be split into more pipes or several pipes may merge into one. Each morning an engineer comes to the site to check the volume of oil pumped from the ground within the last 24 hours. If the volume of oil differs from the long-term average he has to investigate the cause. If a pump is broken he needs to call a pump engineer to fix it. If all pumps are OK he has to check miles and miles of pipes and search for a possible leak. If a leak is found the nearest valve has to be closed and a replacement pipe ordered.

This type of operation is still common in many areas where human resources are wasted on tedious repeated jobs which could be automated. Additionally, when a pipe bursts after an engineer leaves, it will not be noticed until the next visit by an engineer. The oil leak could not only damage environment, it would also cause significant loss of income. Additional costs could be associated with the repairing of devices which were damaged by a continuous oil leak.

This situation could have been prevented if the information were available to close the correct valve immediately after the leak was detected.

In the rest of this thesis we will show how a WSN could help with the monitoring of oil pipes while increasing the reliability, decreasing running costs, and preventing pipe leaks in the case of a pipe burst.

1.2 Application of a Wireless Sensor Network

The scenario described above could benefit from the use of a WSN. A simple WSN could consist of just one sensor node at the tank monitoring the volume increase in the last hour. These data are sent via long-range communication, e.g. GSM or satellite. An application running in a cloud would processes the data and detect fluctuation in received values. If the received value is lower than the long-term average the application notifies a response team which will investigate the decrease in volume. However, this solution will not help the team to localise the problem. The team will still have to check all the pumps and pipes to find the problem. This will not prevent a pipe burst which can still cause a lot of damage to the environment and the equipment.

In order to distinguish between a pipe leak and the dysfunctional oil pump multiple sensor nodes need to be deployed. One sensor node can be deployed at the beginning of the pipe, one at the end, and several in the middle in order to achieve multi-hop connectivity. A more powerful node with more computational power, unlimited battery life, and a wide range communication radio is dedicated as a *base-station* which collects data from the whole network and rely them to the cloud. An application running in the cloud computes volume of oil passing through the sensor at the beginning of the pipe and compares it with the volume of oil passing through the sensor at the end of the pipe. If there is any difference it means that there is a pipe leak. Other sensor readings from the pipe could be used to detect the segment of the pipe where a leak occurred. In case there is no oil passing through the sensor at the beginning of the pipe, it means there is a problem with the oil pump. Now, the application can better determine the cause of the problem and notify the response team with greater precision as to what and where the problem is.

This type of deployment can be seen as the first generation of WSNs. Data from *all* sensor nodes are periodically collected at the base-station which forwards them to a cloud, where reasoning about the data takes place. The WSN is seen only as a source of data. For this type of network algorithms focusing on reliable delivering of sensed data were developed. Collection protocols such as CTP [GFJ$^+$09] or RPL [WTC$^+$12] became very popular and are still widely used.

However, we argue that this approach is not scalable for future needs. It is estimated that by the year 2020 the number of connected devices to the Internet will be between 30 billion [Res13] and as much as 212 billion (including 30.1 billion of autonomous things) with a market value of $8.9 trillion [IDC13]. We assume that this vast amount devices will not create one huge network but will rather be organised in smaller networks consisting of hundreds of nodes. Each network will be used to monitor a single unit, e.g. a building, a street, or a factory.

In order to be able to process these vast amounts of data streams originating in hundreds of millions WSNs, new techniques are being researched. As an example, a new protocol called Constrained Application Protocol (CoAP) [SHB14] was designed to represent a bridge between a WSN and the Internet. Kovatsch *et al.* presented *Californium* [KLS14] - a scalable cloud service capable of handling hundreds of thousands of concurrent CoAP connections between the cloud and many 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 sensing the data and processing them. Furthermore, data centres are already responsible for 1.4% of World-wide energy consumption, growing by 12% every year [ULS14], excluding power consumption of the network infrastructure that scales accordingly.

Lets have a closer look at a wireless sensor node and the key factors influencing its operational lifetime. There are several key characteristics describing a wireless sensor node. Firstly, the

CPU has only limited capabilities and is rather slow when compared to current CPUs used, for example, in cell phones. Secondly, each node has only a very small memory available, usually, a couple of kB. Thirdly, its radio communication is low-powered, hence its communication range is rather short and the transmission rate is just a few hundreds of kbps. And last but and the most important, it is battery powered which means that a node has only a limited amount of energy during its lifetime. This energy is used to power both, the CPU and the radio. However, radio is responsible for much more energy consumption when compared to the energy used by CPU [ZG04]. For example, current draw of CPU of a popular TelosB node is 1.8 mA in an active mode, while radio consumes as much as 23 mA in receiving or transmitting mode [Crob]. Therefore, most of the research in WSNs has been focusing on decreasing the network communication as it can lead to the largest energy savings. By saving energy a node can operate longer and prolong the lifetime of the WSN.

The topology of a WSN can change instantly because the nodes are battery-powered. Any node can die instantly without any prior warning. Similarly, any node can be added to the network. Not only the nodes are unstable. Communication link quality between two nodes is dynamic as well. A link between two neighbours can instantly be lost due to weather conditions, traffic, or masses of people moving by. Each node has to be able to adapt to new environments by communicating with its neighbours.

Many researchers have been trying to minimise communication traffic by using various approaches. One of the popular approaches to decrease the network traffic is by pushing a part of an application logic from the cloud or a base-station (off-line processing) into the WSN. There are three key benefits leading from this approach: *i*) decreasing the network traffic, hence prolonging the lifetime of the network, *ii*) decreasing the load on back-end servers, and *iii*) decrease the response time.

One of the simplest methods to lower the network traffic is by aggregating several values into one. For example, a node may first collect sensed data from its neighbours, aggregate them (e.g. compute an average of values), and only then send the value to the base-station. However, in this case it is still required that all of the nodes sense and send the sensed data.

We have mentioned that the first generation of WSNs were used to collect data only. The next generation of WSNs were deployed not only to collect data from every sensor but also to *detect* a phenomenon. In this case a user is interested in data only if a certain condition is met. In the case of our scenario it could be an oil leak. A simple application running in a cloud could detect a leak by comparing two consecutive readings from the same sensor and computing: $\Delta V = V_i - V_{i-1}$, where V_i is the volume of oil flown through the sensor during the time interval i. Lets say, that there is a leak if the difference is more than 10% of the current reading, i.e. if $\Delta V > 0.1 \times V_i$. This type of computation could easily be pushed into the network as every node can detect an oil leak by comparing the previous reading with the current one. A base-station can push a *filter* to every sensing node. The filter can be just a simple interval of values. If the value falls outside the range of the interval it is sent to the base-station. Otherwise, the value is discarded locally. This simple piece of logic can significantly decrease the network traffic while still it is able to detect large pipe leaks. On the downside, small pipe leaks (i.e. those that increase by less than 10% each sampling interval) will remain undetected.

The filtering technique is applicable only in cases where the decision to send or to discard the value can be done by the node locally, without communicating with other nodes. The decision is based on the current, and possibly historical, readings of one single node. The filtering approach is not applicable if the decision depends on readings from several nodes.

The next step to push computation into the WSN involved the insight that the WSN could be seen as a collection of continuous data streams, produced by sensor nodes. If all of these data streams are put together we can see the WSN as a stream of continuous relational data. Each node is represented by a row and each column represents a variable sensed by a node at given time. Researchers then proposed various *Data Stream Management Systems (DSMS)* which operated on these data streams. DSMS execute *queries* which are then pushed into a network and the result is reported back to the base-station. The operation over streams of data can be expressed using an SQL-like language. There is no standardised language for continuous data stream processing, however, many of the languages designed for this purpose use a similar design. For example, Continuous Query Language (CQL) proposed by Arasu *et al.* is as a part of DSMS called STREAM [ABB+03]. Carney *et al.* presented Aurora [CcC+02] which processes continuous queries expressed in a language very similar to SQL. The simplicity and expressiveness of SQL is behind the fact that basically every data stream processing framework uses an SQL-like language to express the computation over the data streams. The DSMS then translates the query into a *Query Execution Plan (QEP)* which specifies how the query will be executed. It is during the translation of query into a QEP when an optimisation takes place. The optimisation is responsible for generating such a QEP so that the network traffic is minimised.

Many of the current DSMS for WSNs are designed to support only a subset of the continuous SQL-like language. These DSMS focus on execution of queries of a specific type, e.g. aggregation, ordering, join of several data streams, etc. These queries could be categorised according to several criteria. The first criterion shows how many nodes are affected by the query. In the case of a *universal query* sensor readings from every node in the network are required. In the case of a *subset query* data from a subset of the nodes satisfy the query. In this thesis we focus only on *subset queries* as they allow to easily lower the network traffic by requesting the data only form nodes fulfilling given criteria. In our scenario a subset query will require readings only from sensors deployed on one particular pipe.

The second criterion is to categorise queries according to the confidence of the correctness of the result. Some queries may require only an approximation of the result with a certain degree of confidence [UTK13]. In this case the query can be optimised using various summary structures like histograms or Bloom filters [Blo70]. Otherwise, the query requires exact readings from sensor nodes. In this thesis we assume that the user requires exact readings every time a query is submitted. In our scenario a query will require the exact readings of volume passed through the sensor, not only approximation.

The third criterion distinguish the range of queries that the DSMS accepts. Some DSMS accept only a specific types of queries, e.g. AVG, MAX, or MIN [UTK13]). Others support a wider range including *joins* (known from relational databases) between data stream. In this thesis we present a framework capable of performing a wide range of queries including aggregation and join queries.

The last criterion 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. 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. In our scenario an example of a snapshot query would be an engineer checking all sensor readings on a specific pipe p. If the engineer wants to check whether given pipe has no leaks he does not have to communicate with nodes on given pipe. The engineer can submit a query to any node in the network and ask it to find all sensors monitoring the pipe p. The node will find all those nodes, request data from them, and report the data back to the engineer. An example of a continuous query is an application which for each segment of the pipe compares the volume of oil entering the segment with the volume of oil exiting the segment. As we have described in the scenario, the pipes are not linear - a pipe can be split into more pipes or several pipes may be joined into one. If the volumes are not the same it means there is a leak at a given segment and the response team needs to be notified. In this thesis we focus on both, snapshot and continuous queries.

Most of the DSMS presented by researchers so far heavily rely on a base-station [SBB09, SBB10, GBG+11, MJIG10]. The base-station plays a crucial role either in pre-processing the query, post-processing the partial results obtained from the network, or both. However, we argue that a base-station can not be depended upon for various reasons: it may be in a form of a mobile sink which visits the network infrequently, it may fail for various reasons like vandalism or physical destruction, or it can cease to function due to a software error.

In the case when there is no base-station most of the DSMS will not be able to operate. A base-station usually represents a gateway between a user and a WSN. The base-station receives queries from users via long-range communication from the Internet. After pre-processing the query it pushes the QEP into the network and waits for results. Most of the current DSMS compute only partial results inside the network and the final computation has to be done at the base-station. The final result is then sent to a cloud where users can access it.

We argue, that this centralised approach is not scalable as WSNs become omnipresent. Additionally, if the base-station looses its Internet connection, the whole WSN will become inaccessible. In many cases, the Internet connection might not even be available (or not feasible) at the first place. In our scenario the oil field might be in a deserted place, with no GSM signal. Engineers will have to physically go to a base-station whenever they need to read sensor data. In our thesis we present a *fully distributed* approach to in-network data stream processing where all nodes can receive queries from users. Every node can find nodes fulfilling query requirements and response to the user with a result. Our DSMS allows engineers to submit queries no matter where they are located, provided they are within a communication range of any of the sensor node. However, our approach does not require every node to be equipped with hardware capable of communication with a user. In scenarios, where equipping every sensor node with additional hardware is not feasible, we allow the user to communicate and submit queries via a subset of nodes, or even a single base-station. The advantage of our approach is that we allow the user to interact directly with the WSN without the need for an Internet connection.

Another reason for pushing the computation close to data sources and not relying on a base-station comes with the spread of *actuation* WSNs. An actuation WSN is a network which contains actuators, i.e. devices which are capable of influencing the sensed environment. A simple example is switch controlling an air conditioning. A set of sensors may sense the temperature at various places in a room. If the average temperature goes above a threshold value, the AC is turned on, if the average value is lower then a threshold, the AC is turned off. In our scenario an actuator would be a valve. If a pipe leak is detected a message to the nearest valve is sent. The valve is closed in order to stop the leak and minimise the damage. Suppose a traditional method is used. First data are collected at the base-station which then forwards them to a cloud. In case the Internet connection is down, it may take several hours until the Internet connection is available again. Once data are in the cloud, the application will eventually process them. This additional delay may also be significant if there is a vast amount of data arriving every second. Once the application detects a leak, a message is sent to the base-station, provided the Internet connection is working. The base-station then forwards the actuation message to the correct valve.

The last area of research, which has not caught much attention from the research community is *heterogeneity* of WSNs. Currently, a vast majority of research work done in the area of WSN assumes that the networks are homogeneous, i.e. all sensor nodes are the same. A small group of researchers see the heterogeneity of the networks only in uneven distribution of the residual energy. However, we argue that as the number of WSNs grow, more and more of the networks will be heterogeneous on hardware level. Especially, as the old networks will be extended or upgraded. We can see a similar trend also in cloud computing - at the beginning most of the computers in a data-centre were the same. However, as the data-centre is extended and upgraded, new machines are brought, which cause large differences in the computational power. Similarly, we expect the same to happen to WSNs. If we go back to our scenario, the owner of the network may want to extend the current network, and add new sensor nodes in order to decrease the granularity of pipe segments so that a potential leak can be better localised. The nodes already deployed might not be available any more so new, more powerful nodes are added to the network. This causes the network to become heterogeneous.

Heterogeneity brings new challenges to in-network data stream processing. When choosing a node which will process all data streams we need to take into consideration not only where the node is located but also whether it is capable of processing data from a given number of data streams. In the scenario described above, when new sensors are added, more data streams must be processed. The old generation of nodes might not be capable of such computation. During the process when a new processing node is chosen, only the new, more powerful, nodes should be considered.

1.3 Challenges

Detecting an event in an environment monitored by a WSN often requires sensor readings from several sensor nodes. Data produced by these sensor nodes need to shipped to a common node which can process them and detect the event. Sending data requires multi-hop communication due to the short range, low-powered radios. Radio communication in WSNs is the largest consumer of energy. Energy is a finite resource which determines for how long the WSN can operate. Therefore, the biggest challenge is to find a common node inside the network, which will receive and process data streams from all sensor nodes contributing to the detection of an event; the position of this common node has to be chosen in a way that leads to minimising the communication. This is especially challenging in networks with no central node which has a global knowledge about the nodes or the topology of the network. Every node in the network must be equal and capable to accept and satisfy queries without communicating with a central point. In order to tackle this challenge we need to equip every sensor node with following abilities:

1. Allow any node in the network to directly communicate with any other node. A node needs to be able to communicate with other nodes directly, without involvement of a third party, in order to be able to request data from, or to send data to a node. Allowing peer-to-peer communication in a WSN is challenging due to the memory constraints and dynamic nature of WSNs.

2. Each node can be characterised by a set of *static attributes*, e.g. node's ID, room it is deployed in, or what kind of sensor readings it can provide. Any node in the network should be able to identify all nodes fulfilling given static requirements. Finding a list of nodes with given static attributes is challenging without flooding the whole network with a request.

3. Allow any node to satisfy a *snapshot* query submitted by a user, while keeping the communication overhead low. A node has to be able to find all nodes which contribute to the query and request data from them in a timely manner.

4. Allow any node to satisfy a *continuous* query submitted by a user. Similar to a snapshot query, the node has to be able to identify all contributing nodes. Additionally, the node has to find a common node which will be processing data from all data streams. The common node has to be chosen in such a way that the overall communication during the execution of the query is minimised. While searching for the common node in a

heterogeneous network, node's resources (i.e. CPU and RAM) have to be taken into account.

The reason why we have chosen these challenges is to enable the WSNs to support new kinds of applications. Allowing any node in the network to accept queries from a user will enable engineers to deploy the WSNs in new areas, where having a single base-station would not be a feasible solution. An example of such scenario is a remote oil field refinery where having a permanent Internet connection, not only for the base-station, but also for users of this WSN, might be impossible or very expensive. Additionally, by removing base-stations, we can make these WSNs completely distributed, without any single point of failure.

1.4 Contributions

The main contribution of this thesis is that we present a DSMS framework called Dragon capable of executing snapshot and continuous queries submitted by a user to any node in the network. Every node can act as a gateway providing WSN's data or control capabilities to a user. The query execution does not require any central node for pre-processing the query or for post-processing the final result. Particularly, contributions of our thesis are:

1. We present a routing algorithm for WSNs which allows point-to-point communication amongst any two nodes in the network. The routing paths used by the algorithm are optimal or near-optimal in the terms of the number of hops. Because the routing is based on routing tables, after initialisation the communication between two nodes can begin instantly, without discovering the path first. We also present a simple and fast routing table update algorithm which can cope with node failures.

2. We present a Distributed Static Attribute Table (DSAT) which is used to store static information about each node in a scalable way allowing any node in the network to find a list of nodes fulfilling given static attributes by communicating with only its near

neighbourhood nodes. DSAT allows any node to easily find all the nodes satisfying the query while keeping the communication overhead and the delay low.

3. We present a DSMS framework built on top of the routing algorithm and DSAT which can satisfy snapshot queries submitted by users. The framework can easily find all nodes satisfying given query and request data from these nodes with very low communication overhead and in timely manner.

4. We present a DSMS framework for continuous queries submitted by users via any node in the network. The framework is identical with the one for snapshot queries in terms of identifying which nodes satisfy the query. The framework uses the list of source nodes to choose a node (referred to as a *processing node*) in the network which periodically receives data from all the source nodes and performs the computation defined in the query submitted by the user. When choosing the node the framework tries to minimise the sum of distances from the processing node to all the sources. If the condition specified in the query is met, the processing node notifies the user. In the case of heterogeneous network the processing node's capabilities (in terms of CPU and RAM) are taken into account.

1.5 Structure of Thesis

The structure of this thesis is as follows. Before we describe individual Dragon's sub-system we present our evaluation environment in Chapter 2. In Chapter 3 we present a new peer-to-peer routing algorithm for WSNs and that show it can find optimal or near optimal routes while being able to handle the dynamic essence of WSNs. In Chapter 4 we describe the Distributed Static Attribute Table which is used to store static attributes of all the nodes in the network in a distributed way. By distributing the data throughout the network we achieve that every node can easily access this table by communicating with close (possibly multi-hop) neighbourhood only. Chapter 5 describes execution of snapshot queries in our framework, while the following Chapter 6 outlines algorithms for discovering processing nodes for continuous queries. These algorithms are presented for both homogeneous and heterogeneous networks. In Chapter 7 we summarise the ideas presented in the thesis and outline possible future extensions.

1.6 Publications

Below is the list of published and submitted publications related to this thesis.

- Roman Kolcun and Julie A. McCann. DRAGON: Data discovery and collection architecture for distributed IoT. *In Internet of Things 2014 - The 4th International Conference on the Internet of Things (IoT 2014)*, Cambridge, USA, Oct 2014. [KM14]

  This paper describes the novel peer-to-peer routing algorithm covered in Chapters 3, the way static attributes of every node in the network can be stored in a distributed manner as described in Chapter 4, and how these two sub-systems can be used to evaluate a snapshot query submitted by a user to any node in the network, described in Chapter 5.

- Roman Kolcun, David Boyle and Julie A. McCann. Efficient Distributed Query Processing. *IEEE Transactions on Automation Science and Engineering (T-ASE)* for the *Special Issue on Advances and Applications of Internet of Things for Smart Automated Systems. under* 2^{nd} *revision*

  This is an extended version of the previous paper. The paper was invited as one of the nine papers to be submitted in an extended version in the journal.

- Roman Kolcun, David Boyle and Julie A. McCann. Optimal Processing Node Discovery Algorithm for Distributed Computing in IoT. *In Internet of Things 2015 - The 5th International Conference on the Internet of Things (IOT) 2015 (IoT 2015)*, Seoul, Korea, Oct 2015. [KBM15]

  The paper builds on top of the previous paper and shows how to choose a node which process data for a continuous query. In this paper we assume that a network is homogeneous and every node can process the query. The paper describes algorithm for finding a node in the network whose weighted sum of distances to all source nodes is minimised. The algorithm is described in Chapter 6.

- Roman Kolcun, David Boyle and Julie A. McCann. Processing of continuous queries in heterogeneous WSNs. *to be submitted.*

  In this paper we will describe how a node processing a continuous query can be chosen in a heterogeneous network, where only a subset of the nodes are capable of processing the query. The algorithm is described in Chapter 6.

Chapter 2

Testbed

2.1 Introduction

In this chapter we introduce our test environment where we evaluated all DRAGON's sub-systems. DRAGON has been written in nesC programming language, running on top of TinyOS [LMP$^+$05] operating system, and evaluated in TOSSIM [LLWC03] simulator. TOSSIM has been chosen for its reasonably accurate communication model. It can simulate radio noise which causes changes in the link quality and leads to packet loss. Therefore DRAGON , as all other algorithms for WSNs, has been designed to cope with packet losses and none of the DRAGON's sub-systems rely on reliable communication. The second reason why we chose TOSSIM simulator is that Innet [MJIG08, MJIG10], the state-of-the-art competitor at the time, was written in nesC and evaluated in TOSSIM. Therefore our comparison is fair as both algorithms are executed on top the same topologies and in the same simulator.

We have used the built-in radio and noise model with default MAC layer. We assume the nodes are synchronised and operate with 15% duty cycling. The packet size was set to 30 bytes.

In our evaluation of DRAGON platform we studied the influence of various factors on the performance of the platform. Particularly, we evaluated influence of *i*) topology type and *ii*) density of the network.

Topology is one of the main factors which influence the performance of algorithms used in a WSN. As we have mentioned in the previous chapter, some platforms can work only in networks of a specific topology. An example is the Combs, Needles, Haystacks algorithm which works only in a grid topology. We argue, that the platform for WSN should be able to perform well in any type of topology. Therefore, we evaluate Dragon on two sets of topologies: *i*) *uniform* topologies and *ii*) *random* topologies. We exclude the *grid* topology as in real life deployment it is rather unrealistic scenario. Even if the nodes are physically laid out in a grid, the network topology will rather resemble a uniform topology. We argue, that if an algorithm performs well in a uniform topology, it will also perform well in a grid topology. All topologies were generated using the standard network generating tool provided with TinyOS.

The second main factor which influence how well an algorithm performs is the density of the network. If we define number of node's neighbours as the *degree* of a node, then the network density is the average number of node degrees in the network. We evaluated Dragon using four different densities: *i*) *dense* with 12 neighbours on average, *ii*) *medium dense* with an average of 10 neighbours per node, *iii*) *medium sparse* with an average of 7 neighbours per node, and *iv*) *sparse* with only 5 neighbours per node on average.

In our thesis we target deployments with hundreds of nodes as we expect them to be a dominant deployment size of WSNs in the future. Other frameworks for in-network data stream processing were evaluated on networks of various sizes: PEJA [LCC08] and TPSJ [YLOT07] on networks of several hundreds of nodes, Synopsis Join [YLZ06] on a network consisting of 400 nodes, Mediated Join [CNS07, CN07] on a network of 1655 nodes, SENS-Join [SBB09] on networks ranging from 1000 to 2500 nodes, and the state-of-the-art competitor Innet [MJIG08, MJIG10] on networks ranging from 50 to 200 nodes with the focus on networks with 100 nodes. At the beginning we also started experimenting with 100 node networks but in order to show the scalability we decided to run experiments on 250 node networks. Because in order to thoroughly evaluate our framework we ran thousands of experiments it was unfeasible to execute the experiments on networks of any other sizes. The initial experiments which were executed on smaller networks showed similar improvements than the experiments executed on 250 node networks.

For each network density three various networks were generated in order to minimise the influence of a certain topology layout. Every experiment was run for 10 times in order to minimise influence of the randomness of WSNs. In total, the final number of experiments for each Dragon's sub-system was 2 (topologies) ×4 (densities) ×3 (networks) ×10 (experiments) = 240

Results presented in the evaluation parts of the following chapters are presented for each topology separately. The results shown are averages grouped by network density.

2.2 Uniform Topologies

When generating uniform network topology, the area is split into equally sized grids. Next, in every grid same number of nodes are placed randomly. Different network densities can be achieved either by making the area smaller or by extending the communication range. Following four Figures 2.1–2.4 are examples of uniform topologies with various densities. These densities can be clearly identified by looking at the number of links between nodes.

2.3 Random Topologies

In random topologies nodes are placed randomly. Therefore these topologies may create clusters of nodes which are loosely interconnected by a few links only as can be seen in Figure 2.5 for a dense topology and Figure 2.6 for medium dense topology. As the networks get more sparse we can see that clustering gets even more obvious as can be seen in Figure 2.7 and 2.8, both depicting medium sparse topologies. These highly irregular topologies are typically the cause of off-the-chart results in research generally and likewise we also encountered edge-case results while evaluating our algorithms. Regular topologies are especially important for algorithms which rely on the fact that nodes in the network are of similar degree, i.e. all nodes have approximately the same number of neighbours.

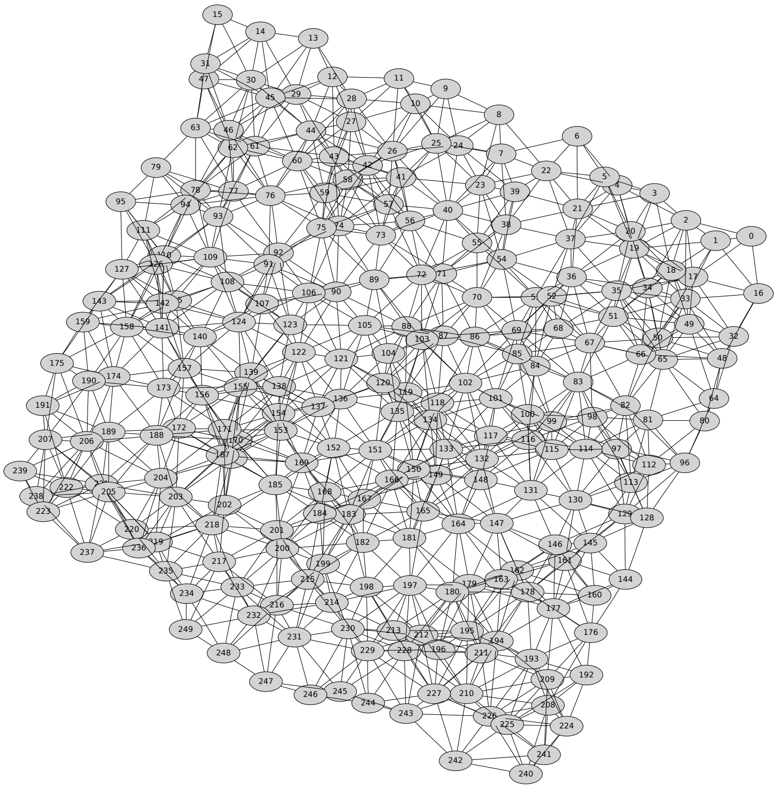


Figure 2.1: Uniform 250 Node Dense Network

In the case of a random sparse topology (Figure 2.9) we can see that the graphic representation of the generated network looks pretty messy. It may seem there is a little difference between the uniform sparse topology and the random sparse topology. However, if we compare the distribution of node degrees, shown in Figure 2.10, we can see that the random network topology has much wider range of node degrees. We can see the same trend when we compare node degrees distribution of other network densities. However, node degree distribution for other network densities are not presented as the difference between uniform and random topologies are obvious from the graphical network layouts.

2.4 Communication Primitives

The essential property of wireless communication is that a message transmitted by a node can be received by all nodes within the communication range from the sender. In this thesis we rely on the usage of several types of communication primitives. Description of these primitives

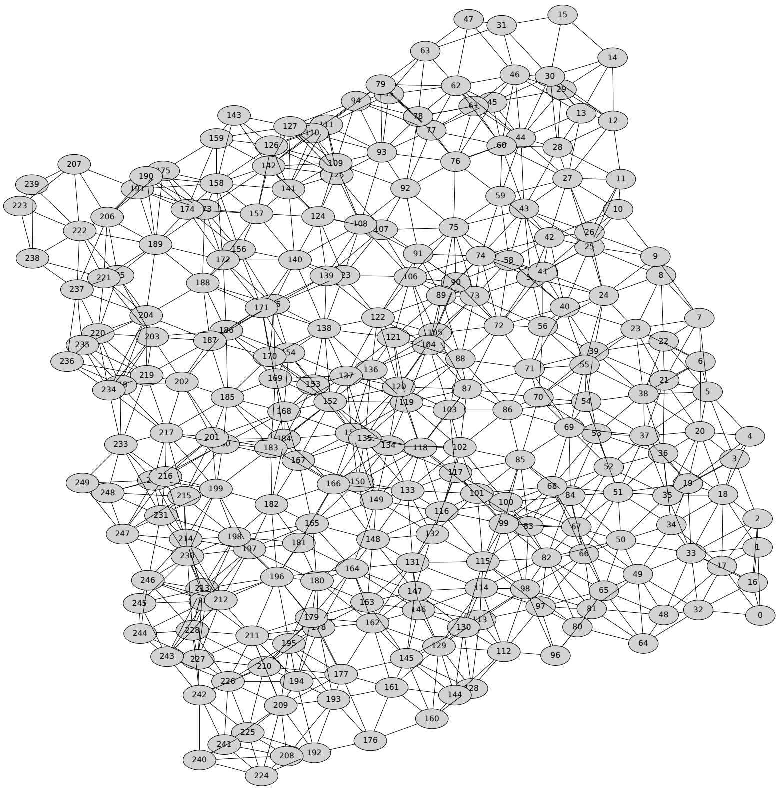


Figure 2.2: Uniform 250 Node Medium Dense Network

is presented below:

Broadcast is the simplest type of wireless communication. Broadcast message is received and processed by all nodes within the communication range. However, there is no guarantee that all nodes (or any, for that matter) within the communication range will receive the message. A node may not receive a message due to the interference, bad link quality, or because its radio is turned off at that moment. In the description of the algorithms this type of communication is called via Broadcast(*message*) procedure.

Unicast is a directed one-hop communication towards one node. It is built on top of the broadcast, with the difference that the header of the message includes ID of the destination node. Because a unicast message is also transmitted wirelessly, the message is received by all nodes within the communication range. However, as a node processes the received message, it first compares the ID included in the message header with its own ID. If both IDs match, the message is processed. Otherwise, the message is discarded. However,

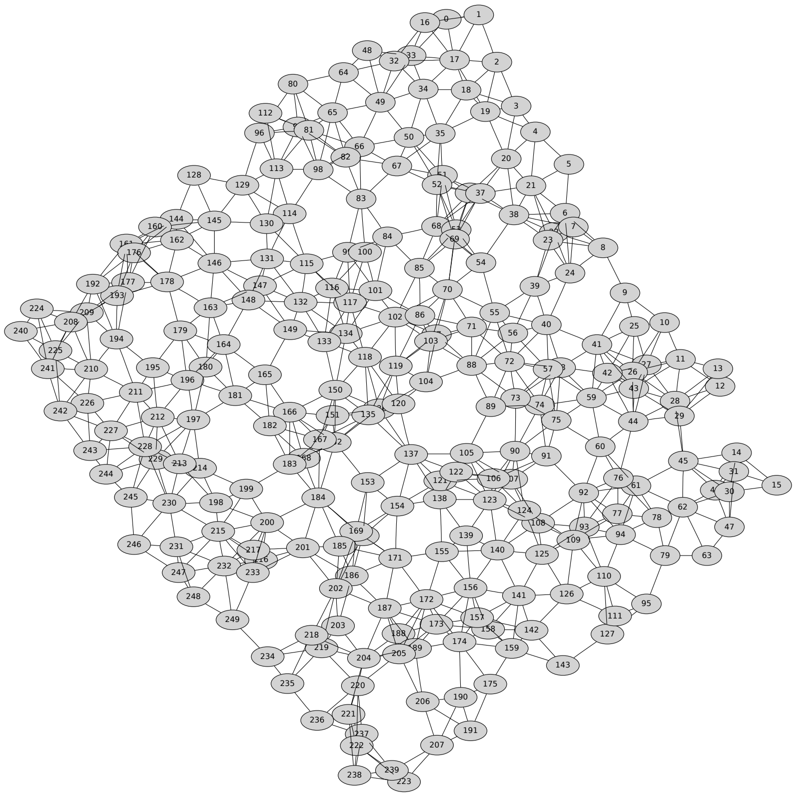


Figure 2.3: Uniform 250 Node Medium Sparse Network

because the unicast is built on top of the broadcast the message delivery is unreliable and there is no guarantee that the message is received by the destination node. In the description of the algorithms we use the notation Unicast($message, destinationID$).

Reliable Unicast is the same as the regular unicast, with the difference that the receiver acknowledges receiving of the message. Acknowledgement is another unicast message sent by the receiver to the sender. The sender waits for a predefined time for the acknowledgement to arrive. If the acknowledgement is not received within this time, i.e. either the message or the acknowledgement was lost, the sender re-sends the message. If the message is not delivered after predefined times of retransmissions, the message is discarded. In the description of the algorithms we use the notation Unicast($message, destinationId, ACK$).

Receiving a message is a basic capability of every node. It occurs when a node receives a broadcast or a unicast message where the destination is the node which received the message. Every message includes an ID of the sender node. In the algorithms presented

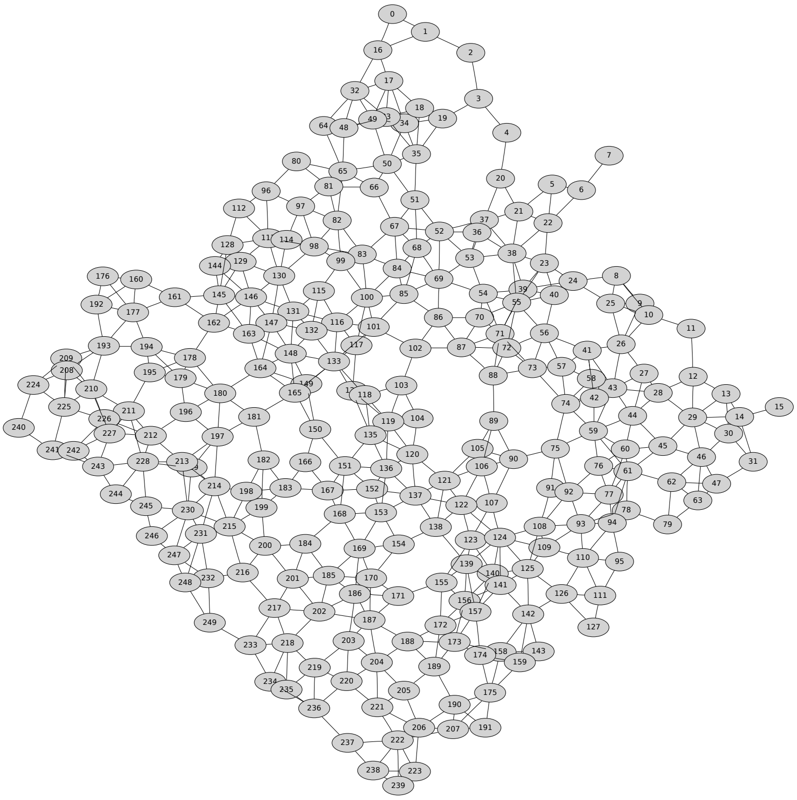


Figure 2.4: Uniform 250 Node Sparse Network

in this thesis, this is operation is shown as RECEIVE($message, senderID$).

Snooping occurs when a unicast message is received by a node which is not specified as the destination node in the header of the message. When the message is parsed and the ID in the header does not match the node's ID, the node may discard the message, or process it. By snooping a message a node can receive important information which it may use in the near future. In the algorithms presented in this thesis this operation is shown as SNOOP($message, senderID$).

CRC stands for Cyclic Redundancy Check and it is a hash of the message. CRC is appended to the footer of each message. Upon receiving a message a node computes CRC of the message and compares it with the value in the footer of the message. If these values do not match, the message is discarded as it is assumed to be corrupted.

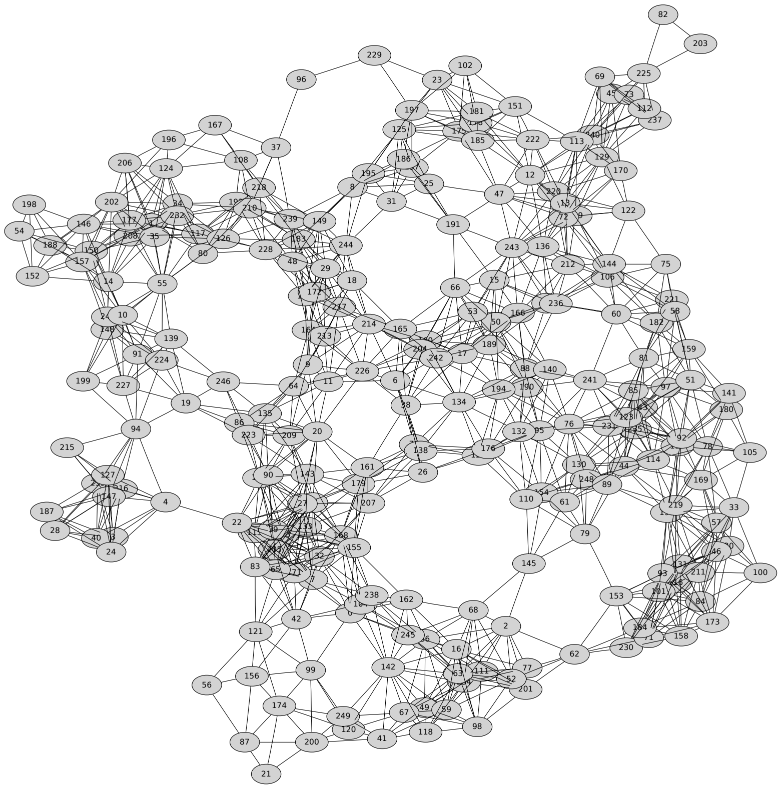


Figure 2.5: Random 250 Node Dense Network

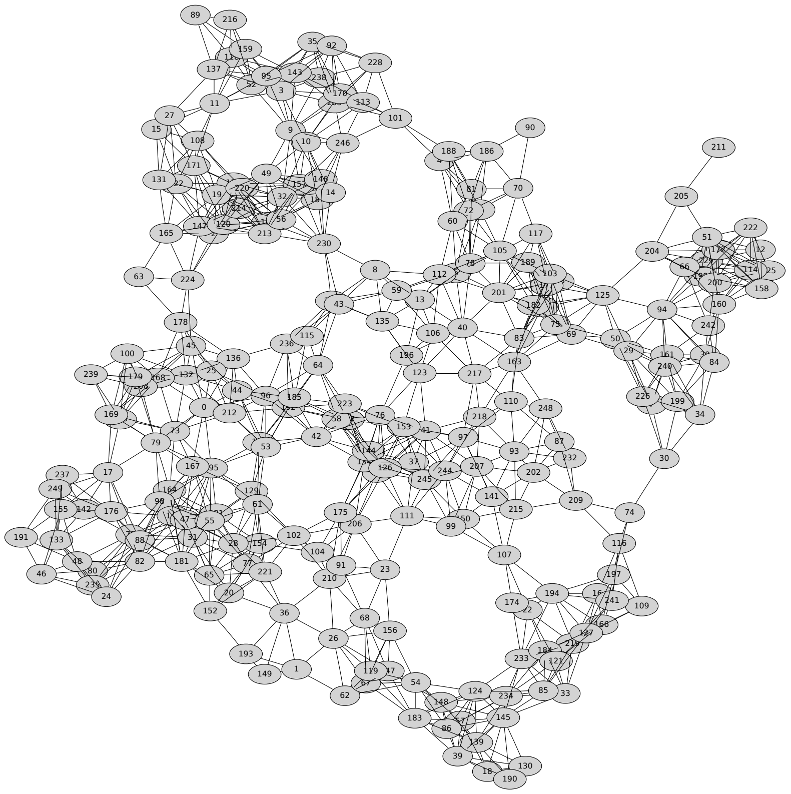


Figure 2.6: Random 250 Node Medium Dense Network

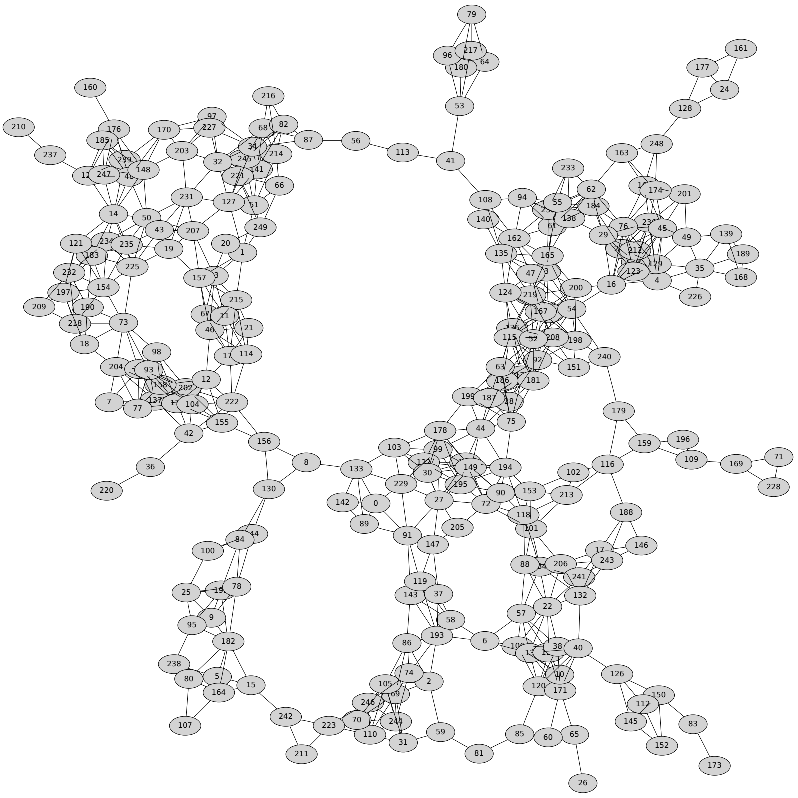


Figure 2.7: Random 250 Node Medium Sparse Network #1

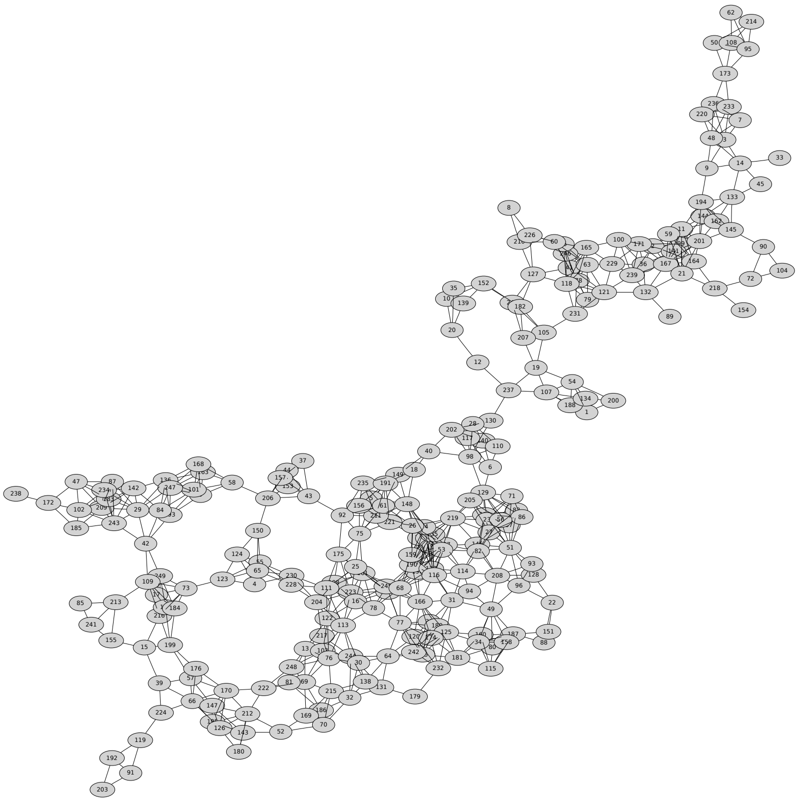


Figure 2.8: Random 250 Node Medium Sparse Network #2

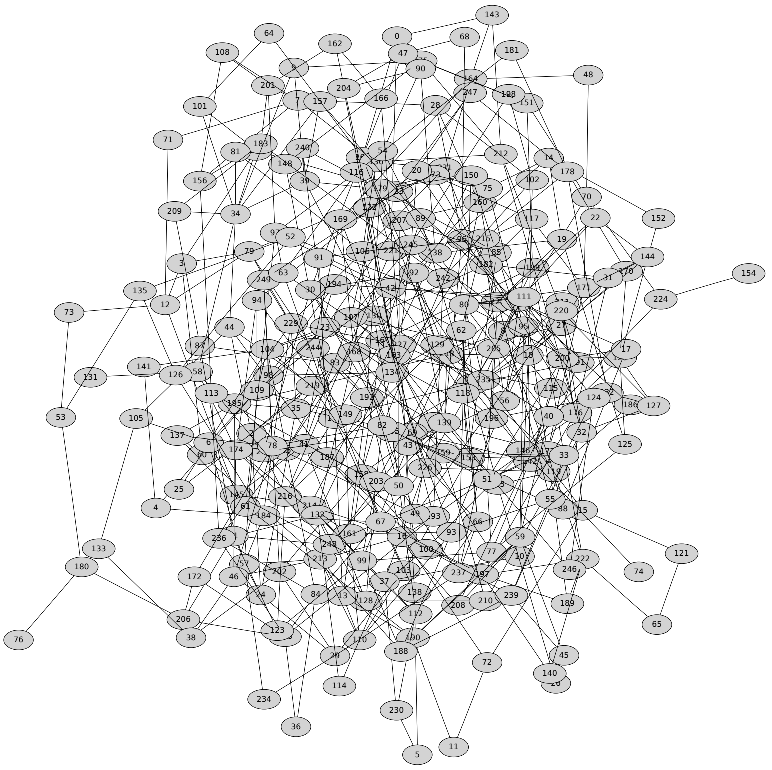


Figure 2.9: Random 250 Nodes Sparse Network

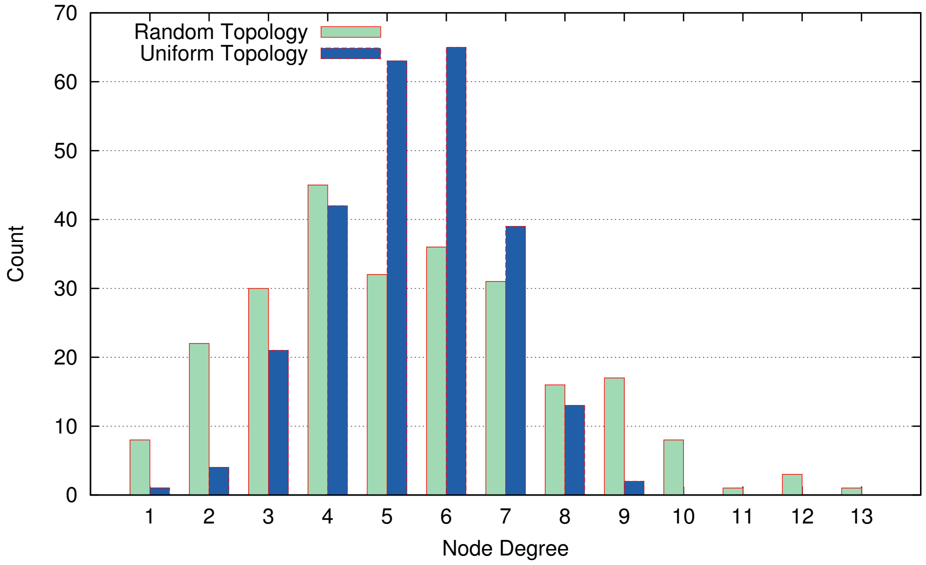


Figure 2.10: Node Degree Histogram for Spare Uniform and Sparse Random Topology

Chapter 3

Routing Table Discovery

3.1 Introduction & Related Work

Routing algorithms are essential for every WSN as they define how data flow within the network. The algorithms differs from each other based on their capabilities, effectiveness, amount of memory required to store the routing state, agility, and energy awareness.

Routing protocols for WSNs could be categorised into two main groups depending on the functionality they provide: *i*) the protocols that route data only towards *one or several predefined base-stations* or *ii*) the protocols that allow *peer-to-peer* communication amongst any nodes in the network. Protocols which belong to the first group are usually based on building *routing trees* which are rooted in the base-station and span throughout the network. Every node in the network forwards all data towards the base-station via its parent. Protocols form the latter group support peer-to-peer communication, i.e. any node in the network can send a message to any other node. Below we will present algorithms from both groups in more details.

3.1.1 Routing Towards a Base-station

Historically, many WSNs were deployed to collect data about a certain phenomena in a predefined geographic area. The nodes in such a network sense a variable at a predefined rate and reports the sensed data towards one base-station. The whole network may contain several base-stations, in which case a node forwards data towards the closest base-station only.

The two most common routing protocols are *Collection Tree Protocol* (CTP) [GFJ$^+$09] and *Routing Protocol for Low Power and Lossy Network* (RPL) [WTC$^+$12]. CTP is a standard library of TinyOS [LMP$^+$05] operating system, while RPL is a standard library shipped with Contiki [DGV04] operating system. There is also an RPL implementation for TinyOS. Both protocols are based on creating a directed acyclic graph rooted in the base-station. Each node in the network forwards all data towards the closest base-station.

Disadvantage of building rigid routing trees is that the nodes on the path towards the base-station have to transfer more data than other nodes, hence their batteries deplete faster. This especially applies to the nodes closer to the base-station. To tackle this problem Lindsay *et al.* proposed a *Disjoint Paths* and a *Braided Paths* algorithms [LRS01]. Disjoint Paths algorithm constructs a small number of alternative paths from each sensor to the base-station. These paths are sensor-disjoint, i.e. paths have no intermediate nodes in common. Braided Paths algorithm creates partially disjoint paths from the primary path, i.e. for each node on the path an alternative path is created which does not contain given node. Therefore, in the case of a node failure an alternative path can be used without the need find it first.

Alternative approach to creating rigid routing trees is the Backpressure protocol (BCP) [MSKG10] presented by Moeller *et al.* In networks with BCP the routing decision depends on the size of the packet queue and the packet rate amongst two nodes. Each node holds a queue of packets, where a base-station has a queue of zero length. A node forwards packet to a neighbour only if the neighbours' queue is shorter than the queue of the sending node. The received packet is put on the top of the queue and in the next iteration forwarded to a node with a shorter queue.

Another older approach which tries to eliminate rigid routing trees is *hierarchical routing.* This approach breaks the network into clusters, each of which has a *cluster-head*. Nodes send data to these cluster-heads which are then responsible for delivering data to the base-station.

Heinzelman *et al.* presented *Low-Energy Adaptive Clustering Hierarchy (LEACH)* [HCB00], one of the popular clustering algorithm whose goal is to reduce energy consumption of the nodes in a WSN. The operation of the algorithm is split into two phases: *i*) the *setup phase* during which cluster-heads are elected and nodes choose which cluster they will be part of, and *ii*) the *steady phase* during which sensor nodes transfer data to the cluster-heads which aggregate received data and forwards them to the base-station.Cluster heads election is distributed and nodes do not require any global knowledge of the network. Disadvantage of this algorithm is that the communication between the nodes and the cluster-heads as well as the communication between the cluster-heads and the base-station is single-hop only, which limits the size of the network. Additionally, cluster-head selection does not take into account the residual energy of the node.

One of the extensions of LEACH algorithm presented by Lindsey and Raghavendra, *Power-Efficient Gathering in Sensor Information Systems (PEGASIS)* [LR02], creates chains of sensor nodes where each node aggregates data received from the previous node with its local data. In each iteration a random node from the chain is chosen to forward aggregated data to the base-station. Disadvantage of PEGASIS is that it assumes that each node has global knowledge of the network layout, particularly, the position of the nodes.

Younis and Fahmy presented another LEACH extension called *Hybrid, Energy-Efficient Distributed Clustering (HEED)* [YF04]. Unlike LEACH it operates in multi-hop networks and for cluster selection it uses both, the residual energy of the node and the node degree or density. This way the algorithm is able to better balance the energy consumption amongst the nodes, hence prolong the lifetime of the network.

3.1.2 Peer-to-peer Routing

For the networks deployed for the purpose of monitoring and continuous collection of data, peer-to-peer (P2P) communication is not necessary. Each node only needs to know how to deliver data to one of the base-stations. However, as the WSNs become more common and serve wider range of purposes, communication amongst the nodes in the network will become more important. WSNs will not be used only to collect data but also to react to the environment and control it via *actuators*. Nodes inside the network will need to send messages directly to other nodes, while lowering the overall traffic. With actuation networks, i.e. networks that contain actuator nodes, this requirement will become even more important as the nodes will need to send an actuation message directly to the actuator. For that purpose, routing protocols which allow P2P communication were developed.

These P2P protocols could be categorised into four groups, depending on how they locate and forward messages in order to the communicate with a peer: *i*) geographic routing, *ii*) routing based on trees, *iii*) hierarchical routing, and *iv*) ad-hoc shortest path routing.

Geographic routing

In geographic routing each node is not addressed by its ID or IP address but by its geographic location. The routing decision is then based on the position of the node making the forwarding decision, the position of the destination node, and the position of the neighbours of the forwarding node. The neighbour which is closest to the destination node is chosen as the next-hop. As geographic routing heavily relies on the exact geographic position of the nodes, either specialised hardware is required (e.g. GPS) or a localisation algorithm needs to be used. However, specialised hardware increases the price of the node, increases the energy requirements, and are often not very precise. Similarly, using localisation algorithms ([BHE00, BP00, DpEG01]) not only leads to additional network traffic, but these algorithms are often not very precise.

Amongst others, Karp and Kung proposed Greedy Perimeter Stateless Routing (GPSR) [KK00], Kuhn *et al.* proposed Geometric Ad-hoc Routing [KWZZ03], and Yu *et al.* proposed Geographic and Energy Aware Routing (GEAR) [YGE01], all of which implement greedy geographic routing. Even though this type of greedy routing works well under ideal condition, it fails when a *routing void* is encountered. The routing void is a situation when there is no neighbour which is closer to the destination node. In practice this situation is common when there is either an error in the localisation algorithm or a physical obstacle prevents radio communication between nearby nodes.

Additionally, there is no practical way of porting geographic routing to the three dimensional space [IvS09], e.g. in a network deployed in a building or a tower.

Routing trees

In networks where P2P communication is based on *routing trees* the nodes are organised in one or several trees where each node stores its parent's ID only. The root of a tree is represented by a more powerful node storing connectivity information of the whole network. The packet is first routed to the root of a tree, where the central router computes the shortest path to the destination. The packet is then routed downwards towards the destination via this shortest path. The advantage of this approach is minimal memory requirements on the nodes and simplicity of the routing algorithm. The disadvantage is potentially high *routing stretch*, i.e. the ratio between the length of the found path and the optimal one, and the requirement that the central router is aware of the whole network topology. Additionally, top-level nodes may become overloaded by the network traffic, especially in large networks.

In order to tackle some of the disadvantages mentioned above, several improvements to the routing trees have been introduced. The principle of improvements is based on storing meta-data on the nodes within the network. Dedicated nodes store meta-data about all nodes in a sub-tree rooted in given node. Then a node can decide to route a packet down a tree without forwarding it to the root node. In RPL [WTC$^+$12] the base-station holds a routing table for the whole network. However, any node in the network, provided it has enough memory, can store a routing table for a sub-tree rooted in given node. These nodes are referred to as *routing nodes*. If a routing node receives a packet, it first checks its local routing table whether it contains a record for the destination. If so, the packet is forwarded directly to the destination. Otherwise, it is forwarded towards the root.

Duquennoy *et al.* presented an opportunistic version of RPL called Opportunistic RPL (ORPL) [DLV13]. Here, each node uses a Bloom filter [Blo70] to store all node IDs from the sub-tree rooted in given node. Each node then uses the summary to decide whether the packet should be forwarded up towards the root of the tree or down the sub-tree rooted in given node. Additionally, when the packet travels up the tree, it does not necessary follow the spanning tree. Any node, which is closer to the root can opportunistically forward the packet. These two improvements can significantly lower the routing stretch. However, the possibility of a false positive in Bloom filters is the main disadvantage of this approach. In this case a special algorithm is required which can recover from the situation when a packet is routed down the tree based on a false positive. The packet has to be sent to the root of the tree which can then find the correct path to the destination.

Mihaylov *et al.* use a similar approach in their Innet algorithm [MJIG08], however, in order to lower the routing stretch, they build up to three routing trees, each rooted in a different part of the network. Each node stores a summary for each sub-tree rooted in given node. The search for the destination node is performed in all routing trees in parallel. Additionally, in order to avoid the problem with false positives of Bloom filters, the packet is always also forwarded up, until it reaches the root of a tree. The packet stores the path it takes until it reaches the destination node. The destination node replies to the source node by reversing this path. As the reply packet travels back to the source node, each node uses several techniques to find a shortcut between the communicating nodes. Innet was designed to support a long-term communication, i.e. the communicating peers exchange messages for a longer period of time. Therefore, the main goal of the algorithm is to minimise the routing stretch. The higher cost of the search phase is outbalanced by savings that could be achieved during the long term communication amongst the nodes.

Hierarchical routing

In *hierarchical routing* each node is a part of multi-level hierarchically organised clusters [IvS09, CnDLR12]. At the lowest level 0, each node is a member of its own singleton cluster. Then, a neighbourhood of level 0 clusters are organised into level 1 cluster, which in turn is grouped into level 2 cluster, until all nodes are member of one (or very few) big cluster. At each level a node is a member of exactly one cluster.

In the centre of each cluster is a *cluster-head*. At each level i the cluster-head is advertised R_i hops away. The R depends exponentially on the level i. A node can be a member of a level i cluster if it is at most r_i hops away from the cluster-head, where $r_i \leq R_{i-1}$. In practice, usually $R_i = 2^i$ and $r_i = \lfloor R_i/2 \rfloor$. Each node is addressable by concatenating the cluster-head ID at each level (e.g. $X.Y.Z$, where node's ID is X, Y is a level 1 cluster-head, and Z is a level 2 cluster-head).

The *routing table*, stored at each node, consists of entries for each cluster-head the node received an advertisement from. Remember that each cluster head is at most R_i hops away at every level. Because the routing table is stored at every node, each node in a network acts as a router. When a node receives a packet it tries to find the record in the routing table for the cluster-head from the lowest level. For example, if the packet's destination is $X.Y.Z$, the node first tries to locate a record for X. If it is not found, it tries the same for Y. Otherwise, it locates the record for Z. Because, Z is the top level cluster, every node in the network will know a route to it. The packet is routed towards the first found record. Because $r_i \leq R_{i-1}$ it is guaranteed that as the packet is routed towards the level i cluster-head, there will be a node on the path which knows the route towards the level $i-1$ cluster-head. Therefore, the packet will eventually reach the destination node.

Ad-hoc routing

Unlike other routing algorithms, *ad-hoc routing* does not require any global preparation phase during which the network is prepared for P2P communication. However, when a node needs to communicate with another peer, a path between the nodes needs to be established first. This is usually done by flooding the network with a request [JM96, PR99, LG00]. The request contains the source node ID, the destination node ID, and a path taken by the request so far. Each node, unless the node is the destination, which receives the request adds itself into the path and re-broadcast the request. Once the destination node receives the request it replies back to the source node by reversing the path of relay nodes. The algorithm leads to discovery of the shortest path between two nodes.

The disadvantage of this approach is a very expensive path discovery as the whole network is flooded with a request. Even though, other approaches, like routing via trees, also rely on the path discovery, the search in those networks is more directed and does not flood the whole network.

In this thesis we describe a fully distributed system, which, from its definition, cannot rely on a single node. Therefore, we developed a distributed algorithm for WSNs which allows peer-to-peer communication among any two nodes in the network. In this chapter we give a full description of the design of our P2P algorithm and how it is able to cope with node failures.

3.2 Routing Table Discovery Algorithm

Routing algorithm is an essential part of the Dragon framework as many of its subsystems rely on it. Routing is based on a *routing table* stored at *every* node. The routing table stores for each node in the network three pieces of information: the *destination*, the *next hop*, and the *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 or ETX as used in CTP [GFJ$^+$09]).

Routing Table Discovery process is split into two phases: the *learning* phase during which each node learns the routing table and the *commit* phase during which each node makes sure that the learned routing table is complete.

During the *learning* phase, which is formally described in Algorithm 1 and 2, each node runs an algorithm *inspired* by Tajibnapis' Netchage [Taj77]. Netchange is 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.

Algorithm 1 Routing Table Discovery - Part 1

```
Preamble: on on expiration of a periodic timer do execute SendRT($rt, packet$)
    $rt$ is a link to the routing table
    $packet$ is a link to an empty packet

 1: procedure SendRT($rt, packet$)
 2:     for all $record$ in $rt$ do
 3:         if is $record$ is updated then
 4:             add $record$ to the $packet$
 5:         end if
 6:     end for
 7:     if the $packet$ is not empty then
 8:         Broadcast($packet$)
 9:     end if
10: end procedure
```

At the beginning of the algorithm, each node creates a record in its local routing table and broadcasts a routing table discovery packet to all its neighbours. A routing table discovery packet contains a list of $\langle destination, distance\rangle$ pairs. Upon receiving a routing table discovery packet the receiving node updates its records in the routing table. If there is no record for the *destination* a new record is created (line 15). If there is a record and the received *distance* is shorter than the one already learned, the routing record is updated. In both cases as the "next hop" is set as the node the discovery packet was received from. The record is marked as "updated". Every iteration records which are marked as "updated" are broadcast to all neighbours. Once the updated record was broadcast it is unmarked.

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 two techniques: *i*) proactively broadcasting better paths, should a node identify one and *ii*) by exploiting overhearing of neighbours updating their routing tables.

This process is demonstrated in Figure 3.1. We assume that node $n_2, n_3,$ and n_4 are neighbours and know about each other (Fig. 3.1a). At some point, node n_2 receives a discovery message from n_1 (Fig. 3.1b). Node n_2 inserts this record into its routing table and mark it as "updated". In the next iteration, this record is broadcast and is received by node n_3 but not by node n_4 (e.g. due to the interference) (Fig. 3.1c). When in the following iteration node n_3 broadcasts

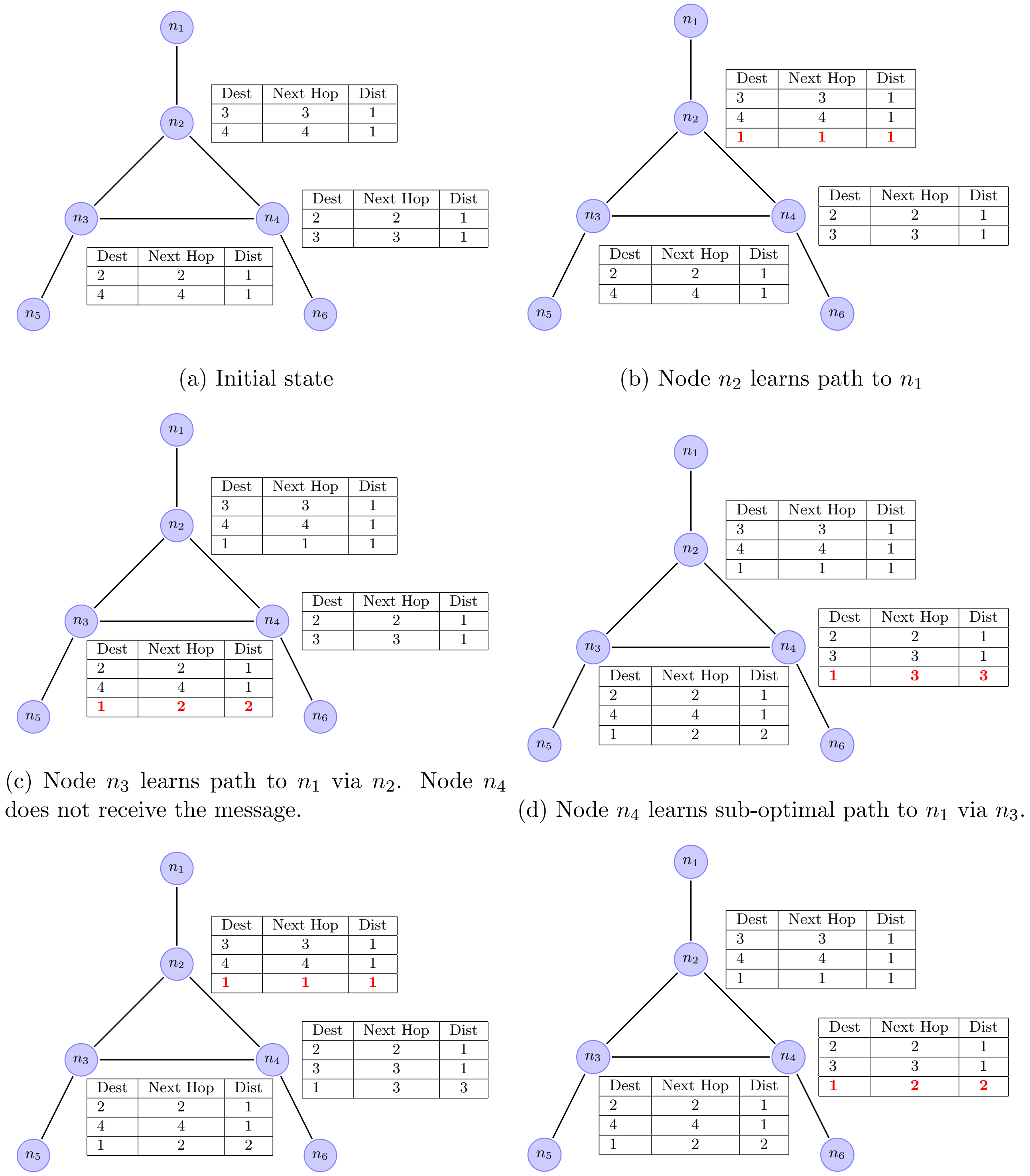
Algorithm 2 Routing Table Discovery - Part 2

```
Preamble: on receiving a message of type RT Discovery sent by procedure SendRT do
    execute ReceiveRT($rt, senderId$)
    $rt$ is a part of routing table received from a neighbour
    $senderId$ is ID of the sender of the packet

11: procedure ReceiveRT($rt, senderId$)
12:     for all $record$ in $rt$ do
13:         $localRecord \leftarrow$ find $record$ in local routing table
14:         if $localRecord$ is not found then
15:             add $record$ to local routing table
16:             mark $record$ as updated
17:         else if $localRecord.hops > record.hops + 1$ then    ▷ Local route is longer then the neighbour's one
18:             $localRecord.hops \leftarrow record.hops$
19:             $localRecord.nextHop \leftarrow senderId$
20:             mark $localRecord$ as updated
21:         else if $localRecord.hops \leq record.hops - 2$ then
22:             mark $localRecord$ as updated    ▷ The node has a shorter route than the neighbour
23:         end if
24:     end for
25: end procedure
```

the update record about n_1 it is also received by node n_4 (Fig. 3.1d). Currently, node n_4 has a sub-optimal path to node n_1 via node n_3 with a distance of 3 hops. When node n_4 broadcasts the updated record about node n_1 it is also received by node n_2. It compares its distance to node n_1 with the one received from n_4. Because distance from node n_2 to n_1 is 1, it means that the distance to n_1 of any of its neighbour should not be more than 2. Because node's n_4 distance to n_1 is 3, node n_2 marks the record to n_1 as "updated" so it will be broadcast in the next iteration (Fig 3.1e, Alg. line 21). Node n_2 re-broadcasts the record and the record is received by node n_4. Node n_4 updates its routing table with an optimal path to n_1 and this record will be propagated further (Fig. 3.1f).

This pro-active broadcasting of better routing paths can significantly improve initially learned routing paths while having only a minimal impact on number of messages transferred during the learning phase.

Once a node has not updated its routing table for some predefined time Δt it assumes that the routing table is complete and the *commit* phase starts. The node broadcasts a commit



Dest	Next Hop	Dist
3	3	1
4	4	1

Dest	Next Hop	Dist
2	2	1
3	3	1

Dest	Next Hop	Dist
2	2	1
4	4	1

(a) Initial state

Dest	Next Hop	Dist
3	3	1
4	4	1
1	1	1

Dest	Next Hop	Dist
2	2	1
3	3	1

Dest	Next Hop	Dist
2	2	1
4	4	1

(b) Node n_2 learns path to n_1

Dest	Next Hop	Dist
3	3	1
4	4	1
1	1	1

Dest	Next Hop	Dist
2	2	1
3	3	1

Dest	Next Hop	Dist
2	2	1
4	4	1
1	2	2

(c) Node n_3 learns path to n_1 via n_2. Node n_4 does not receive the message.

Dest	Next Hop	Dist
3	3	1
4	4	1
1	1	1

Dest	Next Hop	Dist
2	2	1
3	3	1
1	3	3

Dest	Next Hop	Dist
2	2	1
4	4	1
1	2	2

(d) Node n_4 learns sub-optimal path to n_1 via n_3.

Dest	Next Hop	Dist
3	3	1
4	4	1
1	1	1

Dest	Next Hop	Dist
2	2	1
3	3	1
1	3	3

Dest	Next Hop	Dist
2	2	1
4	4	1
1	2	2

(e) Node n_2 pro-actively broadcast path to n_1.

Dest	Next Hop	Dist
3	3	1
4	4	1
1	1	1

Dest	Next Hop	Dist
2	2	1
3	3	1
1	2	2

Dest	Next Hop	Dist
2	2	1
4	4	1
1	2	2

(f) Node n_4 learns optimal path to n_1 via n_2.

Figure 3.1: Pro-active broadcasting of routing table records. Records marked in red are "updated" records which will be broadcast in the next iteration of the Routing Table Discovery algorithm.

message, which contains a number of records in the routing table, and waits for a reply. If the node receives the same number from all its neighbours it finishes the routing table discovery phase. Otherwise, it keeps listening for routing table updates. If a node does not receive the commit message from a neighbour, it assumes that the message or the reply may have been lost so it re-sends the commit message and requests an acknowledgement to ensure that the message is not lost for the second time.

After the *commit* phase the platform enters *stable* phase in which each node broadcasts a small portion of its routing table in a round robin fashion as a heartbeat message in case there is no other message scheduled.

The biggest advantage of our routing platform is the support of peer-to-peer communication along optimal or near-optimal paths. On the other hand, the disadvantages are the space requirements for storing the whole routing table at every node and the need to update large part of the table in the case of a neighbour failure.

The space requirements could be decreased by storing the list of $\langle destination, distance \rangle$ pairs for each neighbour, where the neighbour is the next hop, instead of storing tuples of $\langle destination, next\_hop, distance \rangle$. The cost to store the routing table 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. We leave further reduction of the space requirements as an opportunity for further research. The problem of routing table updates in the case of a node failure is addressed in the following section.

3.3 Routing Table Update Algorithm

In case a node detects a failure of a neighbour, the node executes a failure recovery procedure. A disadvantage of the algorithms based on a routing table is, that if a node fails, all destination nodes for which the failed node was set as the "next hop" become unreachable and these records need to be updated. The process of updating the routing tables of nodes in the network is described in Algorithm 3.

The whole process is illustrated in Figure 3.2 on a small, simple WSN. Each sub-figure lists the first six records of a routing table for four nodes: n_2, n_4, n_5, and n_7. The updated routing records

Algorithm 3 Routing Table Update - Part 1

```
Preamble: on detecting a neighbour failed do execute FailedNeighbourDetected($failedNode$)
    $failedNode$ - a neighbour which failed

procedure FailedNeighbourDetected($failedNode$)
    mark $failedNode$ as failed
    mark all records, for which $failedNode$ is the "next hop", as unreachable
    $packet \leftarrow failedNode$ and the list of all unreachable destinations
    Broadcast($packet$)
end procedure

Preamble: on receiving a message of type Failed Node sent by FailedNeighbourDetected
    or TimerExpires do execute ReceiveFailedNodeMsg($packet, senderId$)
    $packet$ - contains information about the failed node, distance of the sender from the failed
    node, and the list of unreachable destinations of the sender
    $senderId$ - ID of the sender of the packet

procedure ReceiveFailedNodeMsg($packet, senderId$)
    $failedNode \leftarrow packet.failedNode$
    $distance \leftarrow packet.distance$
    $unreachableDestinations \leftarrow packet.unreachableDestinations$
    mark $failedNode$ as failed
    if my distance to $failedNode < distance$ then
        return
    end if
    for all $destination$ in $unreachableDestinations$ do
        add $sender$ to a list associated with $destination$ in a global variable $unreachableList$

    end for
    if first message for the $failedNode$ then
        set timer for time in the future depending on my distance to $failedNode$
    end if
end procedure
```

in the routing table are shown in green or red colour. These marked records are broadcast in the next iteration. We show only a small part of the routing table for a subset of nodes due to the space constraints. Additionally, each sub-figure displays a list of messages sent during the time period depicted in the sub-figure. The node which broadcasts a message is shown in red. The message may contain information about the failed node (marked as "F"), distance to the failed node (marked as "D"), list of unreachable nodes (marked as "U"), and routing table records (marked as "RT").

Algorithm 4 Routing Table Update - Part 2

Preamble: on expiration of a timer associated with the *failedNode* set up by ReceiveFailedNodeMsg(**do**)execute TimerExpires(*failedNode*)
failedNode - node which failed

```
22: procedure TimerExpires(failedNode)
23:     for all destination in unreachableList do
24:         record ← find record for destination in local routing table
25:         if record.nextHop is in the list associated with destination then
26:             mark record as unreachable
27:         else
28:             mark record as updated
29:         end if
30:     end for
31:     if there is a record marked as unreachable then
32:         packet ← failedNode along with the list of unreachable destinations
33:         Broadcast(packet)
34:     else
35:         packet ← failedNode only
36:         Broadcast(packet)
37:     end if
38: end procedure
```

If a node has not received five consecutive heartbeat messages from a neighbour it assumes the neighbour has failed. Once a node detects failure of a neighbour it executes FailedNeighbourDetected procedure which marks the failed node as "failed" 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 destinations, and the node's distance to the failed node (line 2–5). This process can be seen in Figure 3.2b, where nodes $n_2, n_4,$ and $, n_5$ marked node n_1 as the failed one. Each of these nodes broadcast a message notifying their neighbours about the failed node and informing them about destinations they have lost a path to.

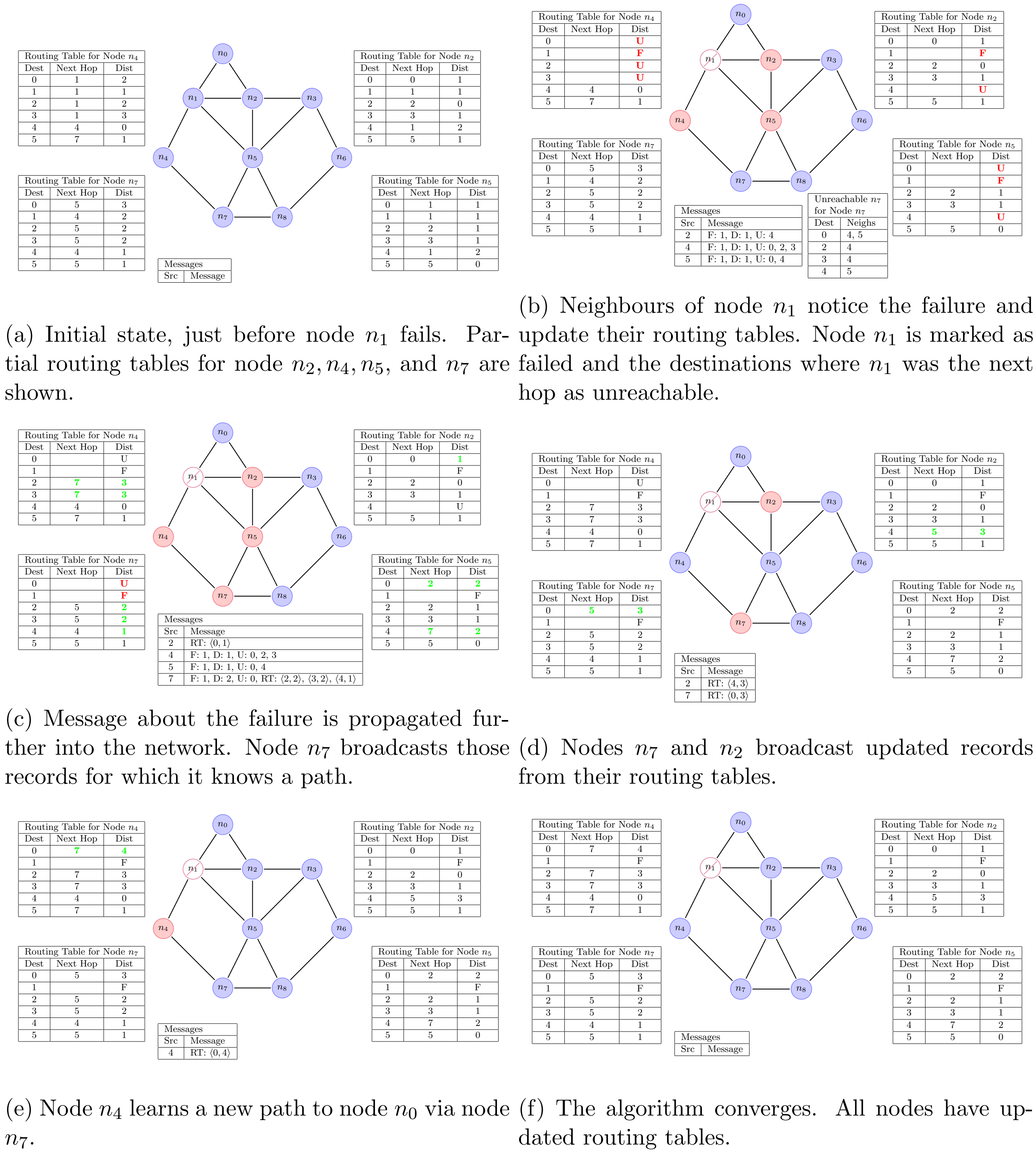
Upon receiving a message informing about a failed node, the receiving node compares its distance to the failed node with the one received from the sending node. If the receiving node's distance is lower than the sender's one the message is ignored (line 12). The message is ignored so the information is propagated further into the network and does not return back and create loops.

If the message is received from a node closer, or the same distance, to the failed node, the receiving node processes the message. 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 16). As there might be more shortest path passing through the same failed node the node has to wait for all the messages from nodes closer to the failed node to be received. For example, in Figure 3.2b node n_7 has to wait for messages from nodes n_4 and n_5 as the node n_0 can be reached via paths n_7, n_4, n_1, n_0 and n_7, n_5, n_1, n_0. Therefore, node n_7 ought not to broadcast the path to n_0 via n_5 immediately after it receives the unreachable list from node n_4. Node n_7 has to wait for the list of unreachable nodes from node n_5, too.

In Figure 3.2b we can see the list created at node n_7. It shows, that the destination node n_0 is currently not reachable through nodes n_4 and n_5, destination nodes n_2 and n_3 are not reachable through node n_4, and the destination node n_4 can not be reached via node n_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 and propagate information about the failed node further into the network (line 19). The timeout depends on the distance from the failed node - the further away the node is the more the node waits. The increasing timeout is important as with the increase of the distance from the failed node the more nodes are informed about the failed node and new alternative paths might be discovered which then decrease number of unreachable nodes. This can be seen in Figure 3.2e where the routing table of node n_7 is fully updated, without any unreachable node. In networks, where nodes have several neighbours, the unreachable list is rarely propagated more than one or two hops away.

After 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 what is the next hop for given unreachable destination. If the routing table lists as the “next hop” a node which is associated with an unreachable destination, the destination is marked as unreachable (line 26). Otherwise, it means the node has an alternative path to the destination and therefore



Routing Table for Node n_4		
Dest	Next Hop	Dist
0	1	2
1	1	1
2	1	2
3	1	3
4	4	0
5	7	1

Routing Table for Node n_2		
Dest	Next Hop	Dist
0	0	1
1	1	1
2	2	0
3	3	1
4	1	2
5	5	1

Routing Table for Node n_7		
Dest	Next Hop	Dist
0	5	3
1	4	2
2	5	2
3	5	2
4	4	1
5	5	1

Messages	
Src	Message

Routing Table for Node n_5		
Dest	Next Hop	Dist
0	1	1
1	1	1
2	2	1
3	3	1
4	1	2
5	5	0

(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.

Routing Table for Node n_4		
Dest	Next Hop	Dist
0		**U**
1		**F**
2		**U**
3		**U**
4	4	0
5	7	1

Routing Table for Node n_2		
Dest	Next Hop	Dist
0	0	1
1		**F**
2	2	0
3	3	1
4		**U**
5	5	1

Routing Table for Node n_7		
Dest	Next Hop	Dist
0	5	3
1	4	2
2	5	2
3	5	2
4	4	1
5	5	1

Messages	
Src	Message
2	F: 1, D: 1, U: 4
4	F: 1, D: 1, U: 0, 2, 3
5	F: 1, D: 1, U: 0, 4

Unreachable n_7 for Node n_7	
Dest	Neighs
0	4, 5
2	4
3	4
4	5

Routing Table for Node n_5		
Dest	Next Hop	Dist
0		**U**
1		**F**
2	2	1
3	3	1
4		**U**
5	5	0

(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.

Routing Table for Node n_4		
Dest	Next Hop	Dist
0		U
1		F
2	**7**	**3**
3	**7**	**3**
4	4	0
5	7	1

Routing Table for Node n_2		
Dest	Next Hop	Dist
0	0	**1**
1		F
2	2	0
3	3	1
4		U
5	5	1

Routing Table for Node n_7		
Dest	Next Hop	Dist
0		**U**
1		**F**
2	5	**2**
3	5	**2**
4	4	**1**
5	5	1

Messages	
Src	Message
2	RT: $\langle 0, 1 \rangle$
4	F: 1, D: 1, U: 0, 2, 3
5	F: 1, D: 1, U: 0, 4
7	F: 1, D: 2, U: 0, RT: $\langle 2, 2 \rangle$, $\langle 3, 2 \rangle$, $\langle 4, 1 \rangle$

Routing Table for Node n_5		
Dest	Next Hop	Dist
0	**2**	**2**
1		F
2	2	1
3	3	1
4	**7**	**2**
5	5	0

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

Routing Table for Node n_4		
Dest	Next Hop	Dist
0		U
1		F
2	7	3
3	7	3
4	4	0
5	7	1

Routing Table for Node n_2		
Dest	Next Hop	Dist
0	0	1
1		F
2	2	0
3	3	1
4	**5**	**3**
5	5	1

Routing Table for Node n_7		
Dest	Next Hop	Dist
0	**5**	**3**
1		F
2	5	2
3	5	2
4	4	1
5	5	1

Messages	
Src	Message
2	RT: $\langle 4, 3 \rangle$
7	RT: $\langle 0, 3 \rangle$

Routing Table for Node n_5		
Dest	Next Hop	Dist
0	2	2
1		F
2	2	1
3	3	1
4	7	2
5	5	0

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

Routing Table for Node n_4		
Dest	Next Hop	Dist
0	**7**	**4**
1		F
2	7	3
3	7	3
4	4	0
5	7	1

Routing Table for Node n_2		
Dest	Next Hop	Dist
0	0	1
1		F
2	2	0
3	3	1
4	5	3
5	5	1

Routing Table for Node n_7		
Dest	Next Hop	Dist
0	5	3
1		F
2	5	2
3	5	2
4	4	1
5	5	1

Messages	
Src	Message
4	RT: $\langle 0, 4 \rangle$

Routing Table for Node n_5		
Dest	Next Hop	Dist
0	2	2
1		F
2	2	1
3	3	1
4	7	2
5	5	0

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

Routing Table for Node n_4		
Dest	Next Hop	Dist
0	7	4
1		F
2	7	3
3	7	3
4	4	0
5	7	1

Routing Table for Node n_2		
Dest	Next Hop	Dist
0	0	1
1		F
2	2	0
3	3	1
4	5	3
5	5	1

Routing Table for Node n_7		
Dest	Next Hop	Dist
0	5	3
1		F
2	5	2
3	5	2
4	4	1
5	5	1

Messages	
Src	Message

Routing Table for Node n_5		
Dest	Next Hop	Dist
0	2	2
1		F
2	2	1
3	3	1
4	7	2
5	5	0

(f) The algorithm converges. All nodes have updated routing tables.

Figure 3.2: Routing Table Update Algorithm. The Figure depicts the process how routing tables are updated during a node failure. Only partial routing tables (RT) of four nodes: n_2, n_4, n_5, and n_7 are shown. In a RT a distance to a node marked as "F" or "U" represents a "failed" or "unreachable" node. Updated routing records (i.e. to be broadcast) are displayed in a green or red colour. Figure 3.2b also shows a list of unreachable nodes collected at node n_7. Every figure also shows a list of messages sent in given time period. Messages are marked as follows: "F" stands for "Failed Node", "D" stands for "Distance to the Failed Node", "U" stands for "Unreachable Nodes", and "RT" stands for "Routing Table Record".

the record is marked as updated so it will be broadcast in the next iteration and the neighbour can learn this alternative path (line 28).

In Figure 3.2c the timer expired at node n_7. It uses the list of unreachable nodes received from neighbours (depicted in Figure 3.2b) to find invalid records in the routing table. We can see the destination node n_0 is marked as unreachable. This is due to the fact that prior to the node failure, as the next hop was chosen node n_4. However, node n_4 has lost its connection to node n_0. On the other hand, node n_7 has alternative paths to nodes $n_2, n_3,$ and n_4, therefore these records are marked as updated and are broadcast to neighbours.

Finally, the node checks whether there are any unreachable destinations in its routing table. If there is any, the list is propagated to the neighbours (line 33). Otherwise, only the information about the failed node is broadcast (line 36).

In Figure 3.2c node n_7 propagates information about the failed node n_1 and unreachable node n_0 to its neighbours. This information is processed only by node n_8 as all other neighbours are closer to the failed node than node n_7. After two more iterations (Figure 3.2d – 3.2e) all routing tables are updated with new routes and the network converges to the stable state, as shown in Figure 3.2f.

3.4 Multi-hop Forwarding with Implicit Acknowledgements

Our multi-hop forwarding algorithm with implicit acknowledgement is built on top of a reliable unicast as defined by the IEEE 802.15.4 standard [oEI03] or used by De Couto *et al.* in their paper on expected transmission count metric (ETX) [DCABM05]. A reliable unicast of a packet amongst two neighbouring nodes requires an acknowledgement packet sent by the receiver to the sender. Therefore, sending a message to a neighbouring node leads to exchange of 2 messages, provided the communication is reliable and neither the message nor the acknowledgement is lost. If we apply the single-hop reliable unicast communication to send a message to a node

Algorithm 5 Multi-hop Forwarding Algorithm - Part 1

Preamble: A node which wants to communicate with *any* node in the network calls SendForwardedMsg procedure
packet - a message sent to another node
destination - ID of a node to whom the message is sent. It is not necessary a next-hop neighbour

```
procedure SendForwardedMsg($packet, destination$)
    $nextHop \leftarrow$ find the next hop for the $destination$ in local routing table
    $packet.destination \leftarrow destination$
    $crc \leftarrow$ compute CRC of the payload of $packet$
    store $packet$ and $crc$ in local buffer
    associate timer with $packet$
    Unicast($packet, nextHop$)
end procedure
```

Preamble: on receiving a message of type Forwarded Message **do** execute ReceiveForwardedMsg procedure
packet - contains the message and the ID of the destination node
sendedId - a neighbour from whom the message was received

```
procedure ReceiveForwardedMsg($packet, senderId$)
    $destination \leftarrow packet.destination$
    $crc \leftarrow$ compute CRC of payload of $packet$
    if $crc$ is found in message buffer then
        enqueue acknowledgement for $packet$
    end if
    if $destination$ is this node then
        process $packet$
        enqueue acknowledgement for $packet$
    else
        SendForwardedMsg($packet, destination$)
    end if
end procedure
```

which is h hops away, the overall number of messages will be at least $2h$. However, this number can be reduced by lowering the number of acknowledgement packets sent by the receivers. The forwarding algorithm described in Algorithm 5 and 6 exploits snooping (i.e. overhearing of communication) in order to decrease the number of acknowledgement packets. The algorithm achieves reliable single-hop communication of a forwarded message in a multi-hop environment while reducing the communication overhead. The algorithm does not guarantee end-to-end reliable communication between two nodes $h > 2$ hops away.

Algorithm 6 Multi-hop Forwarding Algorithm - Part 2

Preamble: **on** snooping on a neighbour and receiving a message of type Forwarded Message (sent on line 7 or 34) **do** execute SnoopForwardedMsg procedure
packet - contains the message and the ID of the destination node
senderId - a neighbour who was forwarding the message to another node

```
22: procedure SnoopForwardedMsg($packet, senderId$)
23:     $destination \leftarrow packet.destination$
24:     $crc \leftarrow$ compute CRC of payload of $packet$
25:     if $crc$ is found in message buffer then
26:         remove $packet$ from the buffer
27:         cancel timer associated with $packet$
28:     end if
29: end procedure
```

Preamble: **on** expiration of a timer associated with a packet (line 6 or 33) **do** execute TimerExpires procedure
packet - a packet in the buffer

```
30: procedure TimerExpires($packet$)
31:     $destination \leftarrow packet.destination$
32:     $nextHop \leftarrow$ find next hop for the $destination$ in local routing table
33:     restart the timer for $packet$
34:     Unicast($packet, nextHop$)
35: end procedure
```

Preamble: **on** receiving an acknowledgement packet (sent on line 13 or 17) **do** execute ReceiveAcknowledgement
packet - acknowledgement for the packet
senderId - ID of the sender of the packet

```
36: procedure ReceiveAcknowledgement($packet, senderId$)
37:     if $packet$ is found the message buffer then
38:         remove $packet$ from the buffer
39:         cancel timer associated with $packet$
40:     end if
41: end procedure
```

When a node wants to send a message to any *destination* node, it calls SendForwardedMsg procedure. This procedure finds in the node's local routing table which neighbour is the next hop for given *destination*. Node computes Cyclic Redundancy Check (CRC) of the *payload* of the message and stores it in a buffer alongside with the message. The reason, why CRC is computed of the payload only and not the whole message is that the header of the message, which contains also ID of the next hop, changes as the message is forwarded through the network, while the payload of the message remains the same. CRC is used as a unique identifier of the message. When the message is inserted into the buffer a timer is associated with the message. The duration of the timer is the time the sender waits for a confirmation from the receiver. Finally, the message is sent using unicast communication to the next hop neighbour (lines 2–7).

Upon receiving a unicast message the receiver executes the ReceiveForwardedMsg procedure. The procedure computes CRC of the payload of the message and compares it with the buffer. If the computed CRC is already in the buffer it means that the sender assumes the receiver had not received the message before, therefore the message was re-sent. In this case the receiver enqueues an acknowledgement packet into the buffer (line 13). If the receiver is the destination node of the message, the processes the message and enqueues an acknowledgement packet as it is not forwarding the message any further (line 17). Otherwise, the receiver forwards the message using SendForwardedMsg function described above.

When a message is forwarded all neighbours snoops this message and execute the SnoopForwardedMsg procedure. Upon snooping a message the node computes CRC of the payload and compares it with messages stored in the buffer. If CRC matches with a message stored in the buffer it means the message was successfully received and is now being forwarded towards its destination. The message is removed from the buffer and the associated timer is cancelled (line 39).

Once a node receives an acknowledgement packet it executes the ReceiveAcknowledgement procedure (line 36) which acts similarly to the SnoopForwardedMsg procedure described above. The procedure removes the message from the buffer and cancels the timer associated with this message.

If a timer expires it means that either the receiving node has not received the message or the sender has not snooped the message being forwarded. As the sender cannot distinguish between these two possibilities, the message is re-sent (line 34).

As it was stated above, sending a message to a node h hops away in a multi-hop forwarding environment while using a reliable single-hop unicast leads to the exchange of at least $2h$ number of messages. The algorithm described above can, under ideal conditions, decrease this number to $h + 1$ (h messages sent by the forwarding nodes plus acknowledgement packet sent by the *destination*). Under "ideal conditions" we assume that no single-hop unicast message is lost and each node on a path between two endpoints can snoop on their neighbours.

The algorithm does not increase the delay when compared with the traditional reliable single-hop way of sending confirmations, provided the timeout is set to the following cycle.

3.4.1 Packet Merging

When a node receives two independent messages from two neighbours with a common destination node, the node may decide to merge these two messages into one. The decision depends on the size of the payload. If the sum of two payloads is less than the maximum size of the packet's payload, two messages are merged.

The problem with merging two messages together is that the payload of the forwarded message differs from the two individual messages, hence the CRC is also different. Therefore the merging node has to send explicit confirmations for the original messages received. However, it is not necessary to send two individual confirmations, as they can be merged into one message.

Another problem with merging two messages is that both messages have to arrive within the same cycle. If the merging node, for some reason, waits for the second message to arrive, the sender of the first message will think, as it has not received the confirmation, that his message was lost and it would re-send the message. Re-sending the message will increase the network traffic unnecessary. The solution to this problem would be to increase the timeout of a timer, so the node would re-send the message later, giving more time to the merging node to confirm reception of the message.

3.5 Evaluation

We evaluate DRAGON's Routing Table Discovery algorithm in our testing environment described in Chapter 2. The evaluation is split into two separate parts. First we compare two version of our algorithm: the *proactive* and the *reactive* version. Next we compare our routing algorithm to other routing algorithms.

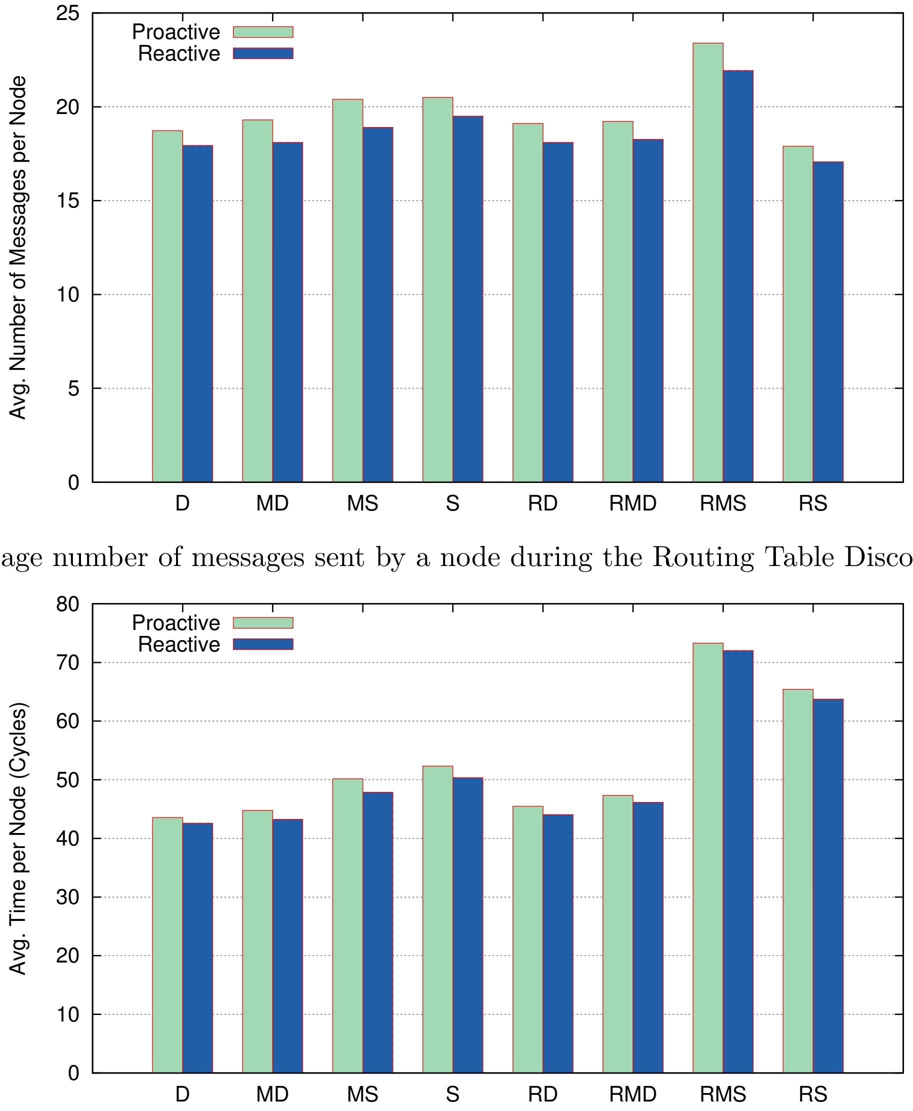
There is a subtle difference, from the algorithmic point of view, between the *proactive* and the *reactive* version of our Routing Table Discovery algorithm. The only difference is that the *reactive* version does not contain lines 21 and 22 in Algorithm 2. In practice it means that a node running the *reactive* version of the algorithm does not broadcast shorter paths to destinations if a node discovers that its neighbour learned a sub-optimal path. If we relate back to Figure 3.1, in the case of the *reactive* version the steps depicted in Figure 3.1e and 3.1f do not occur.

3.5.1 Proactive vs. Reactive Approach to Routing Table Update

In the evaluation of the Routing Table Discovery algorithm we first compare *proactive* and *reactive* version of the algorithm. In this evaluation we focus on three metrics: *i*) number of messages sent by each node, *ii*) time it takes to learn the routing table by each node, and *iii*) an error in the distances learned by nodes, i.e. the difference between the length of the optimal path and the learned one.

Figure 3.3a shows the average number of messages sent by each node during the routing table discovery. As it can be seen from the figure, the proactive version of the algorithm sends only $4 - 7\%$ (5% on average) more messages than the reactive one. This can be expected as the proactive node broadcasts a message also in cases when it overhears a sub-optimal path from a neighbour, while the reactive version does not broadcast anything.

The figure also shows an interesting trend which can be seen in the case of uniform networks. The more sparse the network is, the more messages a node has to send. This behaviour is



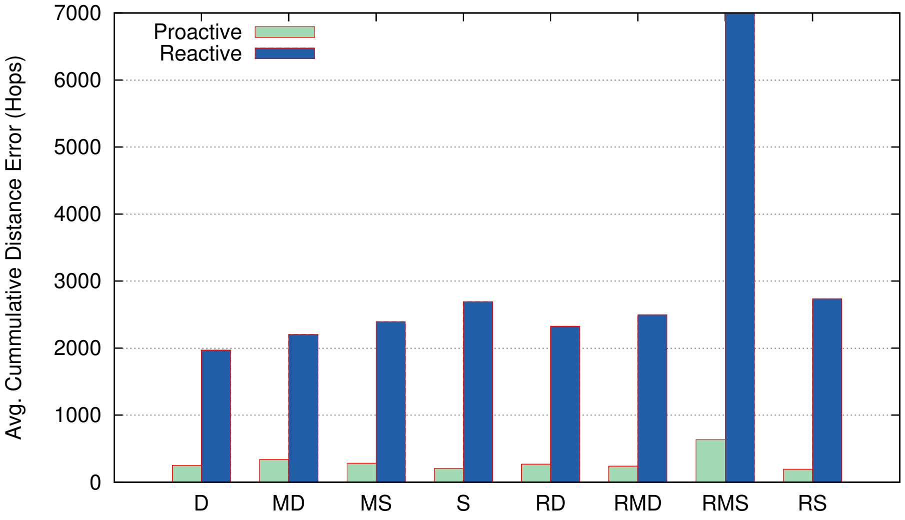
(a) The average number of messages sent by a node during the Routing Table Discovery process

(b) The average time it takes for a node to converge to the stable state.

Figure 3.3: Routing Table Discovery Algorithm. Comparison of proactive and reactive version of the Dragon algorithm. The results are grouped by network topology and network density. The figure continues on the next page. On the x-axis "D" stands for the dense, "MD" for medium dense, "MS" for medium sparse, and "S" for sparse topology. Topologies prefixed with "R" stand for random topologies, otherwise the topology is uniform.

caused by the fact that in more sparse networks the maximum distances between nodes are larger and each node has fewer neighbours. Therefore in one broadcast the number of nodes that receive an information about a certain node is lower.

Next metric we focus on is the average time it takes for a node to learn a routing table. From the comparison in Figure 3.3b it can be seen that the proactive version of the algorithm is slightly slower than the reactive one. In fact, the proactive version is only $2 - 5\%$ slower (with



(c) Cumulative error of distances discovered by nodes in terms of hops.

Figure 3.3: Routing Table Discovery Algorithm. Comparison of proactive and reactive version of the algorithm. The results are grouped by network topologies and network densities. Continuation of the figure from the previous page. On the x-axis “D” stands for the dense, “MD“ for medium dense, “MS” for medium sparse, and “S” for sparse topology. Topologies prefixed with “R” stand for random topologies, otherwise the topology is uniform.

an average of 3%). Again, this behaviour could be expected as the time is strongly correlated with the number of messages.

The last metric we focus on is the error in routing tables. Figure 3.3c shows an average of sums of differences between the optimal distance between two nodes and the learned one. After the algorithm converge we save the routing table learned by each node and compare it with the shortest distance computed externally by Dijkstra algorithm [Dij59]. For each run of an experiment we compute the sum of these differences in the distances. On the y-axis of the figure we display the average of these sums.

As can be clearly seen from the figure, the proactive version of the algorithm greatly outperforms the reactive version by reducing the error by as much as $85 - 93\%$ (with an average of 90%). This extreme improvement is achieved at the expense of only 5% more messages and 3% more time. The reason why the proactive version decreases the error so significantly is due to the unreliability of the wireless communication. Because all of the nodes are building the routing table at the same time, a lot of messages are lost due to the interference. Once a node learns a sub-optimal path, in the case of the reactive algorithm, there is a lower chance that this sub-optimal path is corrected. The node propagates this sub-optimal path further into the network and more nodes may learn the sub-optimal path. In the case of the proactive version of the algorithm, a node proactively broadcasts better paths, should the node find out that its neighbour had learned a sub-optimal path. As it can be seen in Figures 3.3b and 3.3a, this proactive broadcasting of better paths comes at the expense of only a very small network traffic and convergence delay overhead.

The reason why errors in the routing tables occur, even in the case of the proactive version of the algorithm, is again unreliability of broadcasts in WSNs. As it has been shown in Figure 3.1d, node n_4 learns a sub-optimal path to node n_1 via node n_3 because it missed the broadcast message from node n_2. In the scenario depicted in Figure 3.1f node n_4 eventually learns an optimal path via node n_2, however, it happens only if node n_2 overhears sub-optimal path broadcast by node n_4 and, subsequently, node n_4 receives broadcast from node n_2 with an optimal distance. If any of these messages is not received, node n_4 will keep the sub-optimal path.

Additionally, at this point node n_6 stores path to n_1 via node n_4 with a distance of 4 hops. If the node misses subsequent update from node n_4 it will keep and further propagate distance to node n_1 of 4 hops instead of 3 hops, even though the message sent by node n_6 to node n_1 would have travelled via three hops only.

In Figure 3.3c we can see an evident outlier represented by random medium sparse network. As we have described in Section 2.2 medium sparse networks are consist of large clusters of nodes loosely connected by very few links. If an incorrect information about a distance to a node is propagated from one cluster to another, it is highly probable that the rest of the network will learn the wrong distance.

3.5.2 Routing Stretch

The second metric we focus on in our evaluation is the *routing stretch*. Routing stretch is defined as:

$$s = \frac{d_{found}}{d_{optimal}} \tag{3.1}$$

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

Finding the shortest path between two nodes is important as it lowers the network traffic. The reason for it becomes even stronger when this path is used for long term communication, i.e. if two nodes communicate using this path for a longer period. An example of such communication is a continuous query when a node samples data at predefined rate and send them to another node for processing.

In Section 3.1 we described and categorised routing algorithms. In our evaluation we only compare our algorithm to the algorithms capable of peer-to-peer communication. Collection algorithms like CTP [GFJ+09] or Backpressure [MSKG10] are omitted as they support routing towards one node only. From the peer-to-peer group of routing algorithms we do not include algorithms based on geographic routing, e.g. GPSR [KK00] or GEAR [YGE01]. These algorithms rely on the exact geographic location of the 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. Both of these approaches are not very precise. Additionally, geographic routing algorithms are not capable of dealing with routing voids.

From evaluation we also exclude ad-hoc routing algorithms, e.g DSR [JM96] or AODV [PR99]. 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, we compare Dragon routing algorithm only to those algorithms which are capable of restricting the search space. In particular we compare against three groups of algorithms: *i*) algorithms based on one routing tree, *ii*) algorithms based on several routing trees, and *iii*) hierarchical routing.

The representative of the first group is RPL [WTC+12]. Even though RPL was designed to route data towards a base-station, it also supports peer-to-peer communication using following scheme. 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 which 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 about every node in the network. In our implementation, we assume that every single node is a routing node, therefore, the route discovered is the shortest path in a tree.

The second group of algorithms is based on several routing trees, e.g. Innet [MJIG08]. 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 the hierarchical routing. We have not implemented this type of routing but we rely on an extensive evaluation of this type of routing presented by Iwanicki and van Steen [IvS09] where the authors stated that the average routing stretch is 25%.

The results of our comparison are depicted in Figure 3.4. 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, we can claim that Dragon is able to 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 random sparse topology. The routing stretch ranges from only 3% to as much as 28%, with an average of 11%. In the case of the algorithm based on one

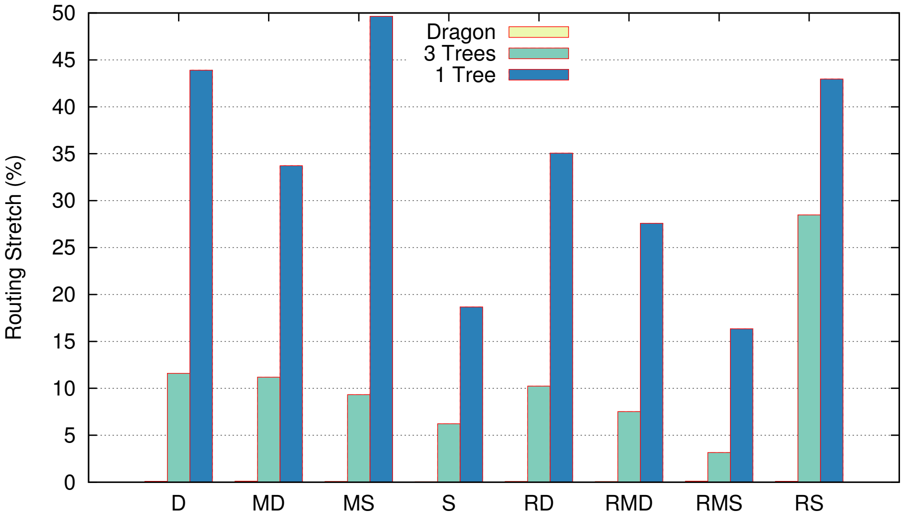


Figure 3.4: Routing stretch of Dragon, algorithms based on one routing tree and three routing trees. On the x-axis "D" stands for the dense, "MD" for medium dense, "MS" for medium sparse, and "S" for sparse topology. Topologies prefixed with "R" stand for random topologies, otherwise the topology is uniform.

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 the bootstrapping Dragon and Hierarchical Routing algorithm can start routing packets immediately, while the algorithms based on routing trees must first discover the paths between the nodes. This discovery phase requires additional messages being sent, hence these platforms are not suitable for ad-hoc communication.

3.5.3 Routing Table Update Algorithm

We tested the Routing Table Update algorithm (described in Algorithm 3 & 4) in several various densities and on different topologies. The network has always converged into a stable state while every node learned new optimal routes to all destinations. However, in order to thoroughly evaluate the algorithm, one must take into account many variables, including but not limited to *i*) network density, *ii*) network topology, *iii*) whether after the node failure the network is still connected or not, *iv*) number of nodes neighbours, *v*) how many paths passes through the node, or *vi*) how many nodes fail at the same time. Evaluation of so many different parameters is beyond the scope of this thesis and we leave it as an open research question which we would like to address in the future.

3.6 Conclusion

Routing is one of the basic part of every application running in a WSN. It could take a form of just a simple forwarding of data towards a base-station, a more sophisticated data aggregation, or inter-node communication. The objective of every routing algorithm is virtually the same - to minimise the communication while achieving the goal. Several constraints have to be taken into consideration while designing any routing protocol, e.g. a limited memory space or a sudden node failure.

Here we have presented a new routing algorithm for WSNs based on routing tables. Its idea is inspired by Tajibnapis' Netchage [Taj77] algorithm. Netchange was designed for wired distributed networks with reliable unicast communication only. This type of communication implies that every message sent by a node is reliably delivered to the node's neighbour. On the other hand, every message can be delivered to one neighbour only. In order to send a message to all neighbours the node has to send the message to each neighbour separately. Our Routing Table Discovery algorithm was adapted for WSNs which uses unreliable broadcast communication. This type of communication allows one message to be delivered to all the node's neighbours at once. On the other hand, the delivery of the message is not guaranteed and the sender cannot know which neighbours received the message. The unreliability of wireless communication was mitigated by proactive broadcasting of better paths, should a node detect its neighbour learned a sub-optimal path. We presented and evaluated two version of the algorithm: *reactive* and *proactive*. We have shown that the *proactive* version of the algorithm can decrease the errors in the routing table by 90% while sending only 5% more messages and taking 3% more time to converge. Additionally, we have also presented how the algorithm updates the routing table in a case of a node failure.

Next we compared Dragon routing algorithm with other state-of-the-art routing algorithms for WSNs. We have shown that Dragon achieves increase in routing stretch by only 0.1%.

The routing algorithm based on one routing tree (e.g. RPL [WTC^{+}12]) increases the routing stretch by $16 - 50\%$, while the routing algorithm based on three routing trees (e.g. Innet [MJIG10]) increased the routing stretch by $3 - 28\%$. Finally, the hierarchical routing [IvS09] algorithm increases the routing stretch on average by 25%.

Chapter 4

Distributed Static Attribute Table

4.1 Introduction & Related Work

Each node in a network can be characterised by a set of *static attributes*, i.e. the attributes which do not change during the lifetime of the network. Amongst them we can include node's ID, location where the sensor node is deployed (e.g. room ID, floor ID, building ID, pipe ID, or geographic coordinates), type of sensors given node has (e.g. temperature, light, accelerometer, magnetometers, etc.), or hardware specification of the node (e.g. CPU, memory size, etc.). We can assume that these attributes will not change during the operation of the node. The actual sensed data streams are *dynamic attributes* as they change over time.

Often, users want to communicate only with nodes which fulfil given static criteria. Kang in his recent survey on in-network processing in WSNs refers to inability to readily allow a node to search the network based a given static criteria as one of the three major problems of in-network processing in WSNs [Kan13]. If we relate back to our scenario, if an engineer wants to see how liquid flows through a given pipe, the system needs to find all sensors with flow meters on given pipe. Nodes on other pipes or nodes on the pipe but not having a flow sensors cannot contribute to the request submitted by the engineer. And it is not only engineers which need to target specific nodes. In actuator networks, if a leak is detected by a node, the node must be able to find an actuator which operates a valve on given pipe. The node which detects the leak informs the actuator which then closes the valve. If the leak detection depends on readings from all sensor nodes monitoring the pipe, the node processing the data needs to be able to find all the nodes with relevant data.

If we consider all static attributes of all nodes in a network we can represent it as a table where each column represents a static attribute and each row represents one node. This table is distributed amongst all nodes and each node holds only one record describing the node.

An analogy of searching for a node fulfilling given static criteria is executing an SQL query on the table of static attributes. For example, to find all flow sensor node on the pipe with id 5 we can execute a query similar to:

```
SELECT id FROM static_attributes WHERE pipe_id = 5 AND sensor_type = "flow".
```
[1]

Various systems may support different operators. Some of the system may support equality operator only while others may also support the inequality operator.

Because we assume that the static attributes do not change during the lifetime of the network, one may suggest to store these metadata in a cloud where users can easily reach it. However, in this thesis we target scenrios where a WSN is isolated with either no Internet connection or no permanent Internet connection. Our goal is to enable any node in the network to find a list of nodes which fulfil given static attributes. A node should be able to achieve it with low network overhead and low latency.

Currently, there are several basic strategies used to find all nodes with given static attributes. Below, we describe these five different approaches in more detail and list their advantages as well as disadvantages.

Flooding

The easiest way how to find all the nodes with given static attribute is to flood the network with a request. This approach was proposed by Intanagonwiwat *et al.* in *Directed Diffusion* algorithm [IGE00]. The request is periodically broadcast by a sink into the

[1] Please note, it is not our intention to create a sensor/actuator SQL-like language in this thesis.

network. During the request dissemination a gradient is used to build a tree-like structure rooted at the sink. Once the request reaches the nodes which hold relevant data for the request they use the tree to route sensed data back to the sink. Similar approach was also proposed by Ye *et al.* in *Gradient Broadcast (GRAB)* [YZLZ05] algorithm. The algorithm works on a similar principle to Directed Diffusion with a difference that instead of a tree a forwarding mesh is created and used to deliver data from sensor nodes to the sink.

The advantage of this approach is its simplicity, its data is current, local space requirements are minimal, and static attributes are not required to be collected a priori. The obvious disadvantage is scalability given the numbers of messages required to collect the data, possible network congestion, and long response times due to the need to wait for a reply from every node in the network.

Super-node

This approach assumes there is at least one super-node (usually the base-station) which has a global knowledge about whole network. This super-node is more powerful than other nodes and is equipped with bigger storage capabilities. The initiating node forwards the query to the super-node which finds all nodes fitting the static criteria and reply back to the initiating node. This approach was adapted by Stern *et al.* in their *SENS-Join* [SBB09] and *Continuous Join Filtering* [SBB10], where all the static attributes are collected at the base-station. When a request is submitted to the base-station, it is able to find the list of all the nodes that fulfil given static criteria. The request is then routed towards the correct nodes via a routing tree.

The advantage of this approach is lower message footprint when compared to flooding the network. However, the number of messages required to retrieve the result depends on the distance between the super-node and the base-station. The disadvantage of this approach is that data from the whole network needs to be collected at the super-node before it can be queried. Additionally, this node represents not only a single point of failure, but also a congestion point because the load is not distributed throughout the network but is forwarded towards one node or very few ones.

This category also includes approaches which ship all static attributes into a cloud. If a node needs to find all nodes with given static attributes it requests data from the cloud via a base-station. Users outside the WSN can send a request directly to the cloud, without relying on the base-station.

Hashing

In this case by “hashing” we mean any type of function or process (e.g. election) which leads to a mapping of a particular value to a particular node. For example, a column in a table is translated to a geographical coordinate. Then a node which is closest to given geographical coordinate stores given table column, e.g. a column storing pipe ID. In case a node needs to find all nodes deployed on given pipe, it sends a request to this particular node and the node replies with the result. We refer to the node which initiates the search as the *requesting node*.

The approaches described below were designed to distribute sensed values in a network in such a way that a node can discover these sensed values without flooding the whole network. Even though these approaches are used to periodically distribute newly sensed values within the network, the same approach could be used to distribute information about static attributes and allow any node to discover them. These approaches could be categorised into three groups: *i*) hierarchical cluster-based approaches *ii*) geometrical approaches, and *iii*) hash-based approaches [CD13].

The first group of approaches is based on *hierarchical clustering*. Demirbas and Ferhatosmanoglu proposed *Peer-to-Peer (PP)* [DF03] algorithm which creates cluster over rectangle-shaped regions. Cluster heads for each region aggregate attributes from all the nodes within the region. If a region exceeds a predefined threshold the cluster-head splits the cluster into new clusters. The requesting node first searches within the same region. If the request is not satisfied, it is propagated upward to search a larger part of the network.

Distance-Sensitive Information Brokerage (DSIB) [FGNW06] proposed by Gibbons *et al.* also falls in the same category. DSIB uses a hash function to choose a node, referred as the *information server*, on which the data are stored. The querying node visits all information server in the hierarchy until the request is satisfied.

Demirbas and Lu proposed *Distributed Quad-Tree (DQT)* [DL07] which splits the network area into grid cells. One grid cell may contain several sensor nodes which share the same data. The cells are organised in a quad-tree hierarchy where four lower-level cells form a higher-level cell. A request submitted by a requesting node is routed towards a higher level cluster-heads until the request is satisfied.

The second group of approaches use *geometrical shapes* to distribute data and to answer requests. Liu *et al.* proposed *Combs, Needles, Haystacks* [LHZ04] algorithm designed for networks with a regular grid topology. Each node broadcasts its attributes to a certain neighbourhood vertically. The shape formed by the nodes storing this attribute remind a shape of a needle. The size of the needle is tunable. The requesting node then propagates the query first vertically and then horizontally every n nodes. The path taken by the query reminds a comb. The distance between the teeth of a comb, which are perpendicular to the needle, depends on the size of the needle in order to ensure that the query and the needle intersects. A node which receives a request and stores data relevant for the query reply back to the requesting node.

Alternative approach based on geometrical shapes is an algorithm proposed by Braginsky and Estrin called *Rumor Routing* [BE02]. The algorithm is based on a probability of intersecting two independent paths in a rectangular-shaped network. An attribute is randomly disseminated into the network. Each node which receives an attribute stores it locally. A node issues a request which travels randomly for a certain amount of time through the network. If the request reaches a node which stores an attribute satisfying the request, the attribute is reported back to the requesting node.

The *hash-based* approaches is the last group of algorithms which distribute attributes in the network. The most common one is *Geographic Hash Table (GHT)* [RKY$^+$02] proposed by Ratnasamy *et al.* GHT stores $\langle key, value\rangle$ pairs in a distributed way. It uses a hash function to map a key into a geographic location. A node closest to the geographic location is chosen as the storage point. GHT relies on GPSR [KK00] routing protocol. The requesting node sends a request to the node which stores given key. If the key is very popular a node storing values for given key may become overloaded. In this case the hash function produces a list of geographic locations. A node sends the value to the closest geographic location. However, the requesting node has to send the request to all of geographic locations from the list.

Greenstein *et al.* proposed *A Distributed Index for Features in Sensor Networks (DIFS)* [GRS$^+$03] algorithm which extends the GHT algorithm with a support for range operators. DIFS has a similar organisation to quad-trees with a difference that one child node points to several parents. At each level of the tree a node stores values for a larger part of the network. When a range request is submitted, DIFS can forward the request to a minimum set of points covering given range. Bottlenecks in DIFS are solved by replicating nodes to several locations and expanding number of parent nodes.

A disadvantage of the aforementioned algorithms is that if a node is not capable of storing whole column of the table, i.e. all association between node ID and pipe ID, several hash functions are used, each of which results into a different geographical coordinate. The table column is then split amongst several nodes. However, in this case, the initiating node has to request data from all nodes storing this information. Similarly, if a node's search is based on several attributes, the initiating node has to send a request to all nodes storing all required attributes which may result in large network traffic. Another disadvantage of this approach is that the hash function in its essence is random, therefore the nodes storing information are chosen randomly and their proximity to other nodes is not taken into consideration. Additional network traffic is generated by distributing static attributes throughout the network before the network can be queries. Last, as we have mentioned in Section 3.1.2, geographic routing has several disadvantages, e.g. it is not able to route in network with routing voids and it requires a node to know its geographic location, which is not always possible.

Approaches which do not rely on hash functions and geographical coordinates, e.g. Peer-to-Peer, Rumor Routing, or Combs, Needles, Haystacks are limited for grid topologies only. These approaches cannot operate on uniform or random topologies.

Summaries

With summaries every node stores a summary of one or more attributes for a certain part of the network. Usually summaries are used in a tree structure where each node stores a summary of attributes for all nodes in a sub-tree rooted in given node. What type of summary is used is determined by a developer and it depends on the static attribute and expected types of queries. Amongst most common types of summaries belong Bloom filters [Blo70], histograms, or R-Trees. Sometimes a node stores several different summaries for the same attribute, e.g. Bloom filter and histogram. The reason is that each summary better serves different type of queries, e.g. Bloom filters better fit queries with equality operators while histograms better serve range queries. When a node receives a query it first checks the summaries stored locally. The node decides whether there is any node in the sub-tree rooted in the node which fits the static criteria of the query. If so, the query is propagated down the tree, otherwise it is discarded. The search space can be further lowered by storing a separate summary for each direct child in the sub-tree. However, storing more summaries means larger memory requirements, especially if several summaries are stored for each attribute. Memory requirements are closely connected with the fidelity of the summaries. The more memory is dedicated for a summary, the more precise it could be and provide user with lower number of false positives.

An example of this approach can be seen in the work of Mihaylov *et al.* who proposed the *Innet* algorithm [MJIG08, MJIG10] or Madden *et al.* who proposed *TinyDB*. These algorithms build a summary tree, where each node holds a summary of a static attribute for all the nodes in a sub-tree rooted in given node.

A disadvantage of this approach is the fact that the search has to originate at the root of the summary tree. Therefore a query submitted to a node in the network has to travel up the tree to the root node and then be propagated down the tree. This fact contributes to a rather high time delays. Additionally, the root represents a single point of failure in the network. These two disadvantages, i.e. the search delay and the single point of failure, can be mitigated by building several summary trees, each rooted in a different part of the network. However, this comes at the cost of even higher memory requirements. This approach was adopted by Innet framework which uses Bloom filters, histograms, and R-Trees to store summary of attributes for each child in three separate summary trees.

Apart from large memory requirements the summary approach also suffers from long and expensive bootstrapping, i.e. it takes long time and exchange of a lot of messages to create summary trees. Summaries could be collected only after a tree is formed. Additionally, as each new root of a tree is chosen to be furthest away from all other previous roots of other summary trees, these summary trees cannot be built in parallel. It is also important to note that the tree structure remains the same and children cannot choose a new parent if the link quality decreases as it would require to rebuild all the summaries. Last, it is also not possible to remove a node's static attribute from a summary.

Store locally

The most convenient solution would be to store all static attributes about all nodes in the network, i.e. the whole static attribute table on every node. In this case each node can easily find all relevant nodes without communicating with any other node. However, the obvious disadvantage are very large memory requirements and the need to disseminate static attributes of every node to every other node. To the best of our knowledge, there is no system applying this strategy to store information about static attributes. The probable reason is the memory constraints of a typical WSN node. For example, memory size of MicaZ node is 4KB only [Croa].

Here we have described most common approaches for allowing any node in a network to identify a set of nodes with given static criteria. Each of the aforementioned approach imposed different requirements on memory, number of messages required to fetch the result and bootstrap. In the rest of this chapter we propose a new approach to distributing static attributes information about every node throughout the network.

4.2 Distributed Static Attribute Table

Our design takes a form of the last approach described in the Introduction above, i.e. it stores all static attributes locally. We prefer storing real values to storing summaries as with the real data a node can instantly find out which node fulfils given static criteria and which does not. As we assume that the static attributes will not change throughout the lifetime of the network we can assume that the higher cost of bootstrapping will be easily overcome by savings made during the lifetime of the network as we assume that searching for other nodes with given static attributes will be a common type of query submitted by users, nodes, or actuators.

However, because WSN nodes are very limited in terms of memory, fitting the whole table of static attributes on a single node may not be possible. Due to this limited memory, we took an inspiration 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. In case a user wants to execute a query on such 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 computer which issued the query. Similarly to this partitioning design we split the static attributes table into n equally sized *parts* and each node needs to store only one part which could easily fit into the node's memory. We refer to this distributed table as the *Distributed Static Attribute Table (DSAT)*.

Similarly to the design of distributed tables in traditional distributed databases, when a node receives a query instructing it to find all nodes which have a static attribute equal to x, the node first looks at its local DSAT 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. So, if a node wants to search the whole table it has to send at least $p-1$ messages and receive $p-1$ replies, assuming that the nodes containing the rest of the table are its neighbours.

DSAT introduces two challenges: *i*) into how many parts should the table be split and *ii*) how to distribute the parts amongst the nodes. The first challenge could be solved rather easily: we compute the size of the whole table in bytes and divide it by the size of the memory we are willing to dedicate for storing static attributes. 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. We assume that the developer knows how big the DSAT could be and how much memory each node can dedicate to storing part of the DSAT. Similar decision must be also done in case *hashing* or *summaries* are used to store static attributes.

However, the second challenge is much harder to tackle. In the case of distributed databases, the system is not concerned with which computers store the distributed table because the cost of communication within the same network is uniform. Also, if data replication is not used, each computer stores a different part of the distributed table. On the other hand, in the case of a WSN, nodes communicate in a multi-hop manner. Therefore the cost of communication depends on the number of hops between the endpoints. 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. Because we assume that the DSAT will not change during the lifetime of the network we allow more than one node to store the same part of the DSAT, i.e. the parts are replicated throughout the network.

In terms of communication we can define the lower bound of number of messages a node has to send in order to search in the whole DSAT. Let the DSAT be split into p parts, N be the number of nodes in the network, and $D_i = \{d_0, d_1, \ldots, d_{N-1}\}$ be the vector of distances to all nodes from node n_i ordered ascending order. The minimum cost c in terms of the number of messages a node n_i has to send is defined as:

$$c_i^{min} = \sum_{j=0}^{p-1} d_j \tag{4.1}$$

In other words, it is 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.

If we think about each part as a colour, the objective of assigning DSAT 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 with minimum number of hops.

So how can we assign DSAT parts to nodes in such a way that we minimise average c_i of the whole network? 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 which 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 we define the following metric. The real normalised cost c of a node n_i to perform a search in the DSAT is defined as:

$$c_i = \frac{1}{p} \sum_{j=0}^{p-1} d_{ij} \tag{4.2}$$

where d_{ij} is the number of hops from node 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 has to send 3 messages in order 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 has to send is 4 (one part is at least 2 hops away), therefore the normalised cost will be $c = 4/4$. Because we want to minimise number of messages in the *whole* network, the normalised cost C of the whole network is:

$$C = \frac{1}{N} \sum_{i=0}^{N} c_i \tag{4.3}$$

where N is the number of all nodes in the network.

To minimise the overall cost we propose the following distributed algorithm described in Algorithm 7. The objective of this algorithm is not only to choose the $partId$ but also to discover closest nodes which 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

Algorithm 7 Distributed Static Attribute Table

```
Preamble: on receiving a message of type DSAT Association do execute RECEIVEDSATASSOCIATION procedure

 1: procedure RECEIVEDSATASSOCIATION($packet, senderId$)
 2:     $receivedDSAT \leftarrow packet.dsat$
 3:     update $dsat$ with data from $receivedDSAT$
 4:     if message of type DSAT Association was received for the first time then
 5:         set timer with a random timeout
 6:     end if
 7:     if $dsat$ was updated then
 8:         $packet.dsat \leftarrow dsat$
 9:         BROADCAST($packet$)
10:     end if
11: end procedure

Preamble: on expiration of the timer set by RECEIVEDSATASSOCIATION procedure (line 5) do execute DSATTIMEREXPIRES

12: procedure DSATTIMEREXPIRES
13:     $partId \leftarrow$ choose part which was chosen least times by neighbours or is furthest away
14:     insert chosen $partId$ into $dsat$
15:     $packet.dsat \leftarrow dsat$
16:     BROADCAST($packet$)
17: end procedure
```

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, or in the case of a large network, several nodes may initiate the algorithm simultaneously. The initiating node calls DSATTimerExpires function in which the node chooses its *partId* randomly and broadcast 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 a random delay, after which it will choose its own *partId*. Which *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: *i*) the node chooses the *partId* which has been chosen by a node furthest away or *ii*) the node chooses the *partId* which has been chosen by least number of neighbours. Ties are resolved by choosing the *partId* randomly. We evaluate both techniques and show under what conditions is one superior to the other.

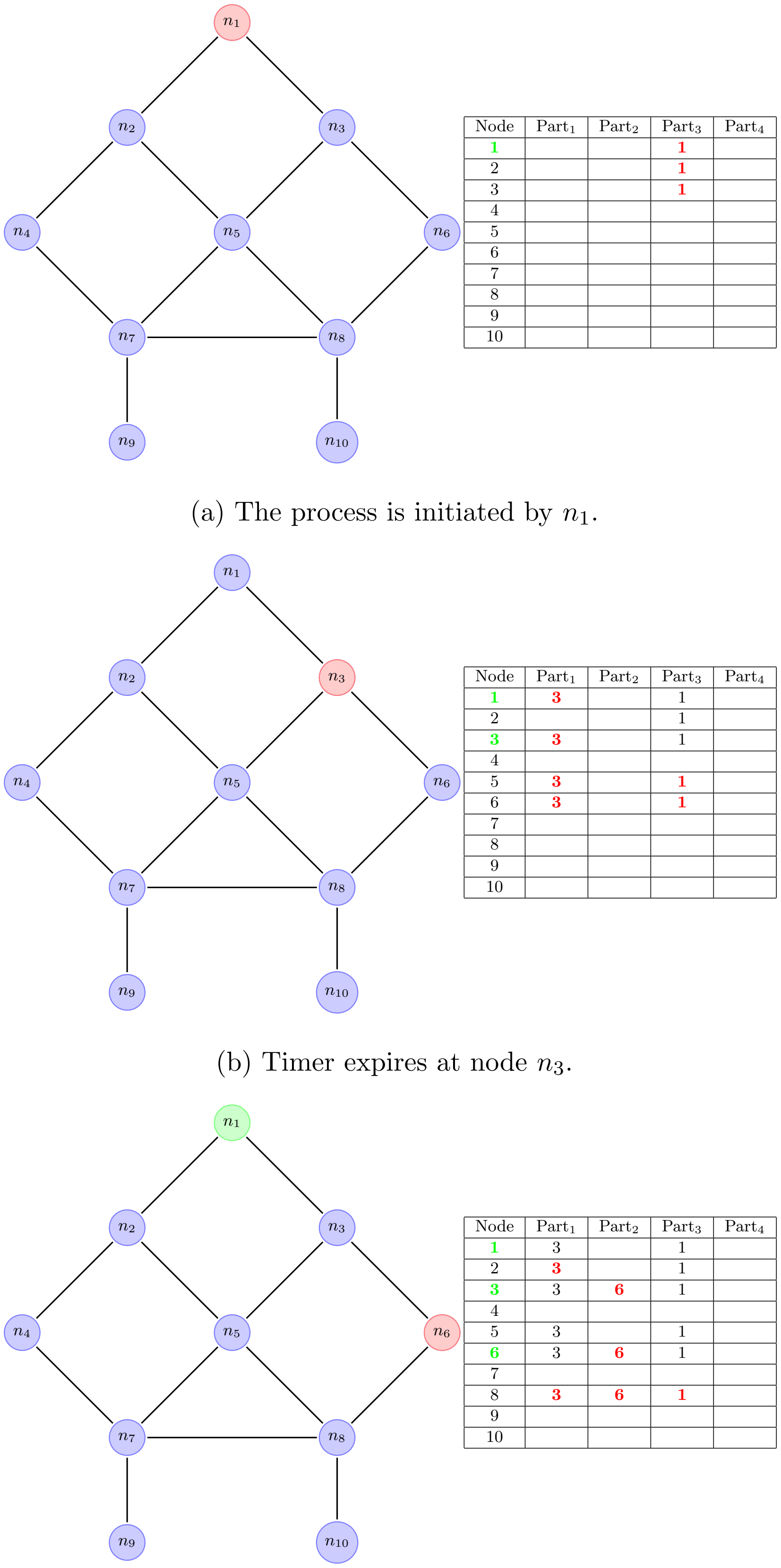
The whole process is demonstrated in Figure 4.1. In this scenario the node uses the first technique, i.e. node chooses the *partId* which is chosen by a node furthest away. The parameter $k = 1$, i.e. each node stores only one closest node for each part of the DSAT. The node which runs DSATTimerExpires function is depicted in red colour, while the node depicted in green shows a node which re-broadcast its updated *dsat* (line 9. In the table summarising all *dsat* variables, the node shown in green is the node which has already chosen its *partId*, while the value shown in red represent 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 from node n_1 is received by its neighbour nodes n_2 and n_3, both of which starts a timer with a random delay (Fig. 4.1a). Timer at node n_3 expires sooner and the node chooses to store part 2. This information is broadcast to all neighbours (Fig. 4.1b). There are two cases in which the *dsat* variable is updated: *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-broadcast it.

4.3 Static Attribute Propagation

Static Attribute (SA) which describe each node in the network are stored in the DSAT. How a node decides which part of DSAT it will store was shown in the previous section. Now, the DSAT has to be filled in with actual SA of each node. Theoretically, in order to propagate each node's list of SA to every other node, each node has to 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 could be significantly reduced using algorithm described in Algorithm 8. Prior to starting the algorithm each node learns a list of common neighbours with every other



Node	Part$_1$	Part$_2$	Part$_3$	Part$_4$
1			1	
2			1	
3			1	
4				
5				
6				
7				
8				
9				
10				

(a) The process is initiated by n_1.

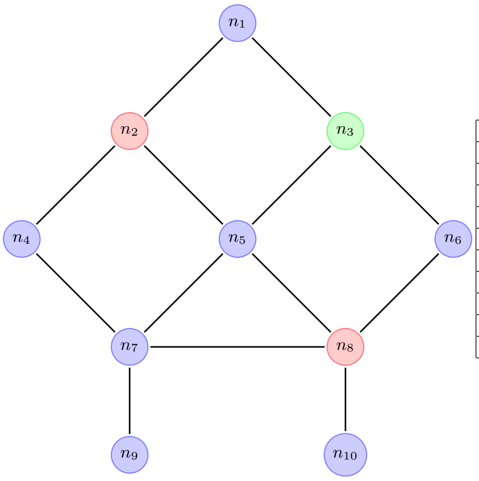
Node	Part$_1$	Part$_2$	Part$_3$	Part$_4$
1	3		1	
2			1	
3	3		1	
4				
5	3		1	
6	3		1	
7				
8				
9				
10				

(b) Timer expires at node n_3.

Node	Part$_1$	Part$_2$	Part$_3$	Part$_4$
1	3		1	
2	3		1	
3	3	6	1	
4				
5	3		1	
6	3	6	1	
7				
8	3	6	1	
9				
10				

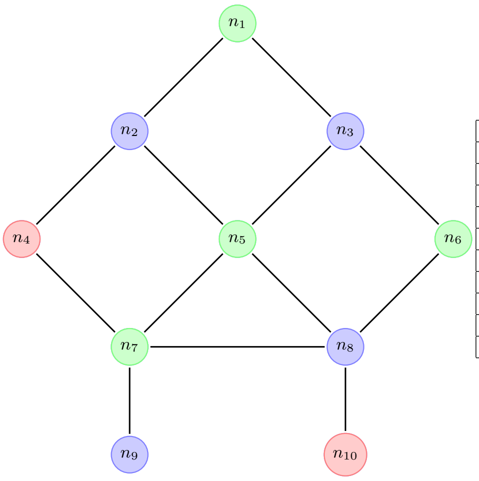
(c) Timer expires at node n_6. Node n_1 re-broadcast its *dsat* as it was updated in previous epoch.

Figure 4.1: Assigning parts to nodes in the DSAT. Part 1/3



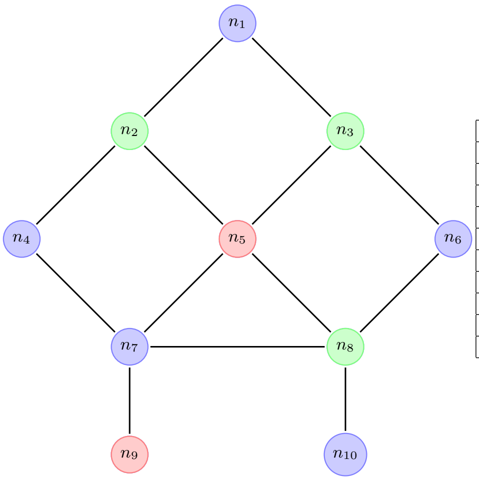
Node	Part$_1$	Part$_2$	Part$_3$	Part$_4$
1	3	2	1	
2	3	2	1	
3	3	6	1	
4	3	2	1	
5	3	2	1	8
6	3	6	1	8
7	3	6	1	8
8	3	6	1	8
9				
10	3	6	1	8

(d) Timer expires at nodes n_2 and n_8. Node n_3 re-broadcast its updated *dsat*.



Node	Part$_1$	Part$_2$	Part$_3$	Part$_4$
1	3	2	1	
2	3	2	1	4
3	3	6	1	8
4	3	2	1	4
5	3	2	1	8
6	3	6	1	8
7	3	6	1	8
8	3	6	10	8
9	3	6	1	8
10	3	6	10	8

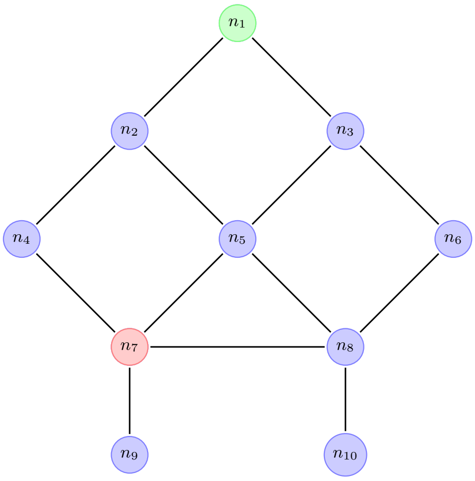
(e) Timer expires at node n_4 and n_{10}. Nodes n_1, n_5, n_6, and n_7 re-broadcast their updated *dsat*. Almost every node has its *dsat* full.



Node	Part$_1$	Part$_2$	Part$_3$	Part$_4$
1	3	2	1	4
2	3	2	1	4
3	3	6	1	8
4	3	2	1	4
5	3	2	5	8
6	3	6	1	8
7	3	9	5	8
8	3	6	10	8
9	3	9	1	8
10	3	6	10	8

(f) Timer expires at node n_5 and n_9. Nodes n_2, n_3, and n_8 re-broadcast their updated *dsat*.

Figure 4.1: Assigning parts to nodes in the DSAT. Part 2/3



Node	Part_1	Part_2	Part_3	Part_4
1	3	2	1	4
2	3	2	1	4
3	3	6	1	8
4	**7**	2	1	4
5	3	2	5	8
6	3	6	1	8
7	**7**	9	5	8
8	**7**	6	10	8
9	**7**	9	1	8
10	3	6	10	8

(g) The last node to choose its part is node n_7. After node n_1 re-broadcast its *dsat* the process is finished.

Figure 4.1: Assigning parts to nodes in the DSAT. Part 3/3

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 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 timer with a random timeout 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 (line 13). If n_r has already received the sa before, n_r just removes n_s and all n_s's common neighbours from the ln.

Once a random timeout has expired, n_r is ready to broadcast the sa using the function StaticAttributeTimerExpires. Prior to broadcasting, the node checks the ln (line 16). If the 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 18). The timeout is chosen randomly in order to avoid all nodes broadcasting at the same time.

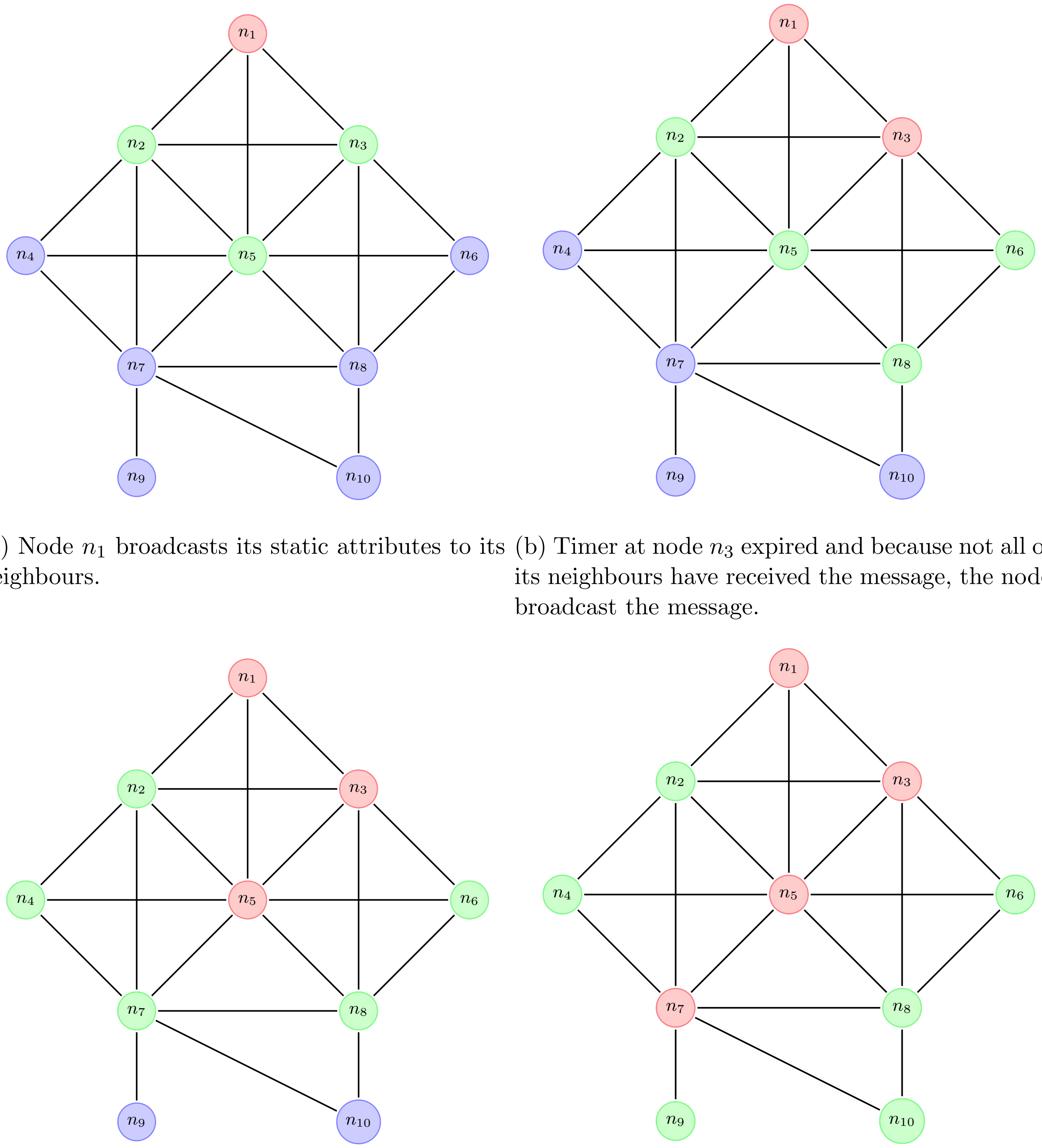
The principle of how the algorithm disseminates static attributes of node n_1 in the network is demonstrated in Figure 4.2. Nodes depicted in red colour are the nodes which broadcast static attributes to their neighbours. The nodes depicted in green are the nodes which received node's n_1 static attributes but they did not broadcast this information further as at the time when

Algorithm 8 Static Attribute Propagation

```
Preamble: on receiving a message of type Static Attribute do execute ReceiveStaticAttribute procedure

 1: procedure ReceiveStaticAttribute(packet, senderId)
 2:     staticAttribute ← packet.staticAttribute
 3:     if staticAttribute is received for the first time then
 4:         insert staticAttribute into buffer
 5:         associate timer with a random timeout with staticAttribute
 6:         associate list of neighbours with staticAttribute
 7:         if this node stores the part of DSAT to which staticAttribute belongs then
 8:             insert staticAttribute into DSAT
 9:         end if
10:     else
11:         fetch staticAttribute from the buffer
12:     end if
13:     remove senderId and all common neighbours with the senderId from the list of neighbours associated with staticAttribute
14: end procedure

Preamble: on expiration of the timer associated with a staticAttribute set by ReceiveStaticAttribute procedure (line 5) do execute StaticAttributeTimerExpires procedure

15: procedure StaticAttributeTimerExpires(staticAttribute)
16:     if list of neighbours for staticAttribute is not empty then
17:         packet.staticAttribute ← staticAttribute
18:         Broadcast(packet)
19:     end if
20:     remove staticAtrribute from buffer
21: end procedure
```

their timer expired all their neighbours had already received node's n_1 static attributes.

First, node n_1 broadcasts its static attributes to all neighbours (Fig. 4.2a). After a random delay, node n_3 broadcasts the static attributes information it received from node n_1 (Fig. 4.2b). Next, timer at node n_5 expires and the node broadcasts the attributes (Fig. 4.2c). At this moment, when the timer expires at nodes $n_2, n_4,$ or n_6, the nodes will not broadcast the attributes, as all their neighbours have already received this information. Last, the timer expires at node n_7 and the node's n_1 static attributes are disseminated in whole network (Fig. 4.2d). As it can be seen from the figure, only four nodes out of ten were needed to disseminate information to the whole network.



(a) Node n_1 broadcasts its static attributes to its neighbours.

(b) Timer at node n_3 expired and because not all of its neighbours have received the message, the node broadcast the message.

(c) Timer at node n_5 expired and the node broadcast the message. If a timer at node n_2, n_4, or n_6 expires, the message will be discarded as all of their neighbours have received the message.

(d) After node n_7 broadcast the message all nodes have received the message and no further communication is required.

Figure 4.2: Static Attribute Propagation Algorithm. Only four out of ten nodes are needed to propagate the static attributes of node n_1 throughout the network. Nodes shown in red are the nodes which broadcast the message, nodes in green are the nodes which has received the message, and the nodes in blue are the nodes which has not received the message, yet.

The algorithm finishes after a predefined Δt from the time the node has received 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 our implementation we use a simple modulo function, i.e. the node n_i stores SA of a node n_j if and only if

$$j \bmod partId = 0 \tag{4.4}$$

Due to the unreliability of 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 is first requested from the nodes storing the same part of DSAT. If these nodes also miss this information the static attributes are requested directly from node n_x.

4.4 Evaluation

In this section we separately evaluate assigning *partId* to nodes and dissemination of static attributes throughout the network. Algorithms are evaluated on the platform described in Chapter 2.

4.4.1 Distributed Static Attribute Table

We evaluate our algorithm using the cost function defined in Equation 4.3. The cost represents normalised average number of messages any node in the network has to send in order to reach all nodes storing the DSAT. We compare two versions of our algorithm with a naive random assigning of parts to nodes.

Unfortunately, we were not able to find an algorithm which can find a “perfect assignment” of parts to nodes. Because the search space is extremely large: p^N, where p is number of parts the

DSAT is split into and N is number of nodes in the network, as a reference point we provide a “theoretical minimum” which computation is based on Equation 4.1

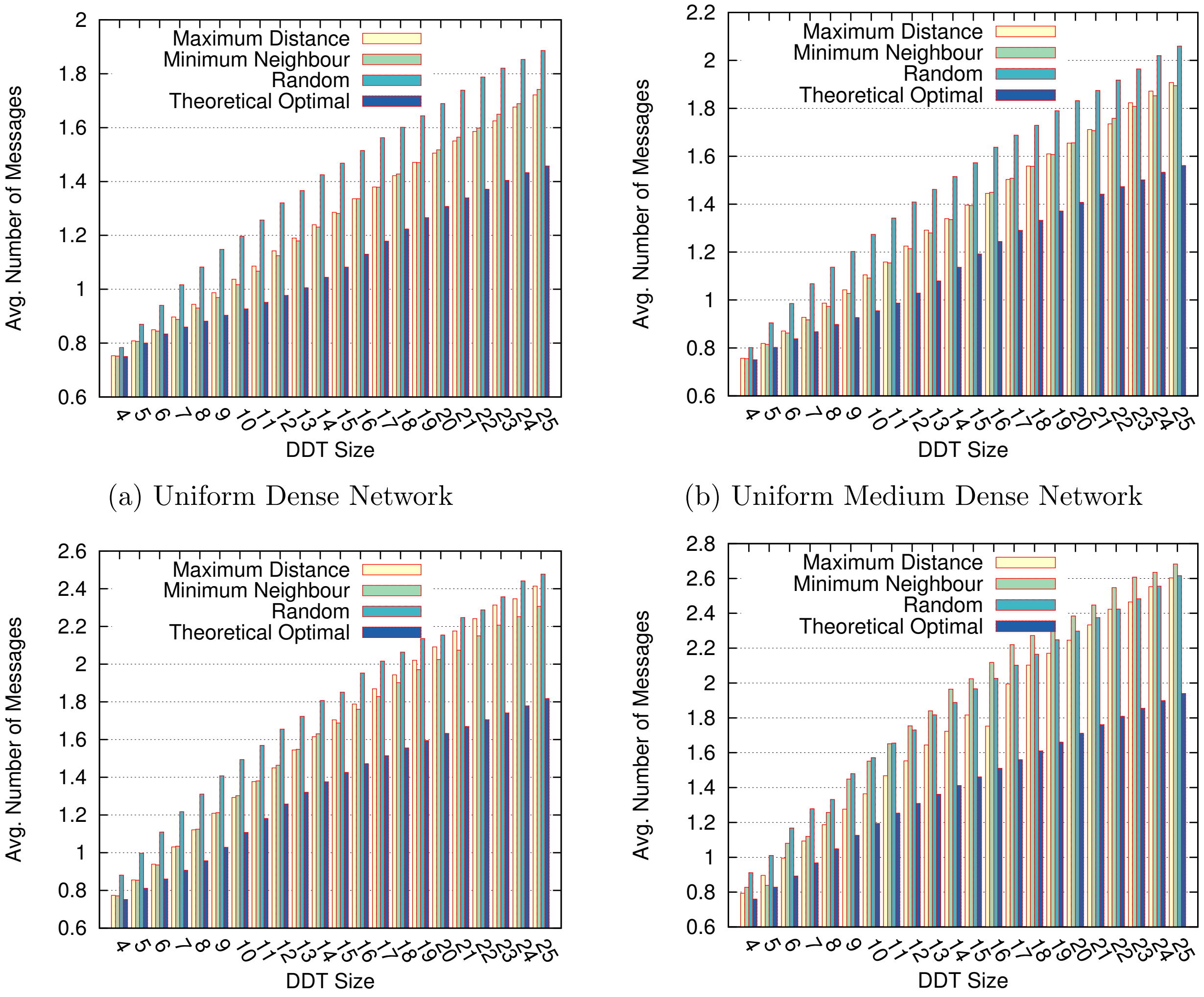
$$C^{min} = \frac{1}{N} \sum_{i=0}^{N} \frac{1}{p} c_i^{min} \tag{4.5}$$

C^{min} is the normalised cost computed by summing of distances to $p-1$ closest nodes. We assume, 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 might not be possible.

Figure 4.3 shows results of comparison of two our algorithms with a random assignment and a theoretical minimum in uniform networks of various densities. 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 4.3. Even though on average both our algorithms perform very similar (with an exception of sparse networks) when we look at the results in more detail we can see that the Maximum Distance version of our algorithm works better in more sparse networks or if the DSAT is split into more parts. On the other hand, Minimum Neighbour version of our algorithm performs slightly better in dense networks or if the DSAT is split into fewer parts. This behaviour could have been expected as in dense networks each node has more neighbours. In this case the Maximum Distance algorithm has often chosen 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 given part and chooses the part which was chosen least times.

Table 4.1 summarises the results for uniform topologies, showing minimum, maximum, and average gain for both versions of algorithm when compared to random assignment. Results are grouped by network density. It could be seen that the average gain is around 10% while the maximum gain can be as high as 17%.

Figure 4.4 shows results for random networks of various densities. As it can be seen from the figure, in random network the Maximum Distance and Minimum Neighbour version of our algorithm perform very similar. If we take a closer look at the results, similarities with



(a) Uniform Dense Network

(b) Uniform Medium Dense Network

(c) Uniform Medium Sparse Network

(d) Uniform Sparse Network

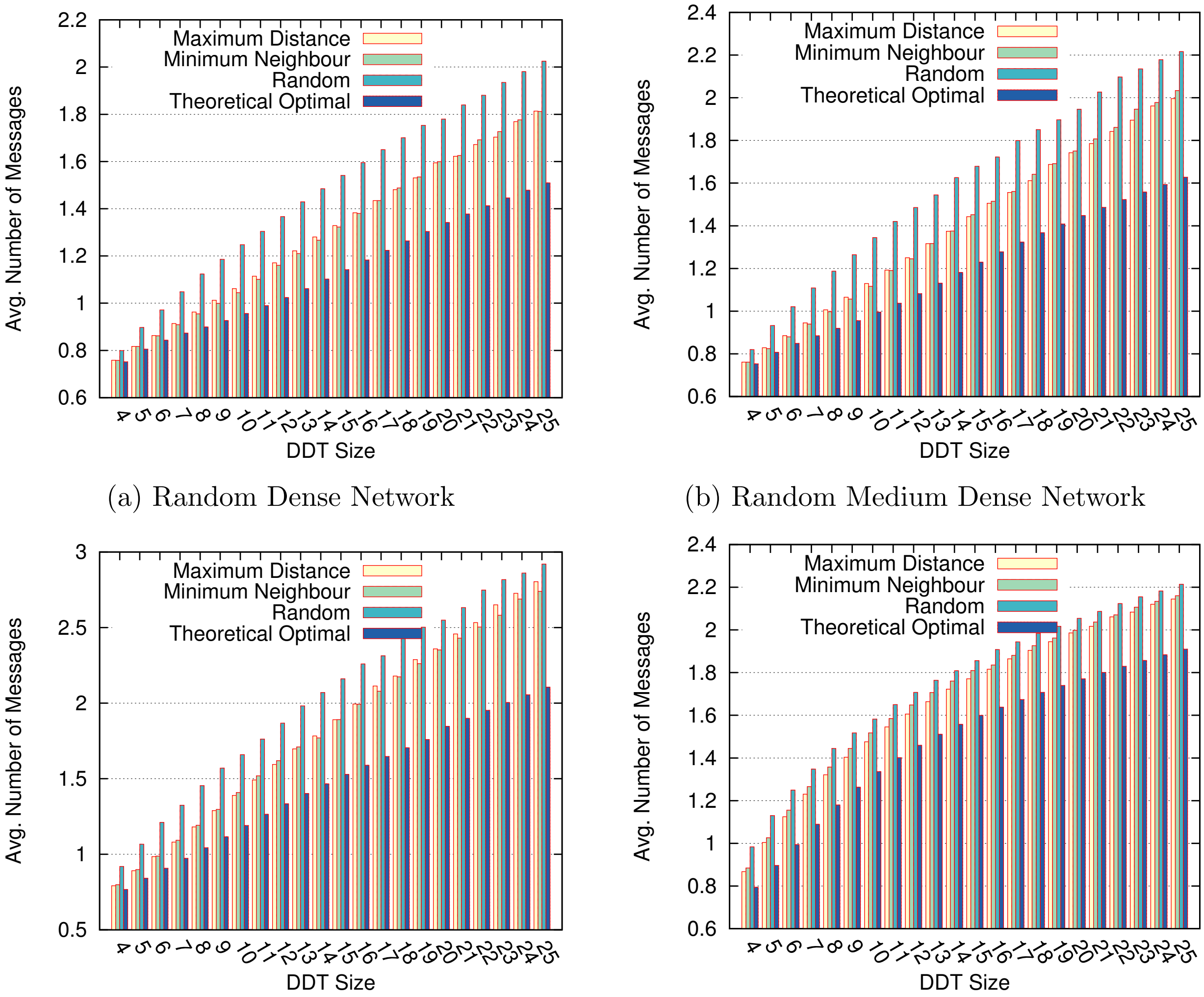
Figure 4.3: Assigning parts of DSAT to nodes in uniform networks.

the uniform results could be seen, however, the differences in performance are much smaller. Similarly to uniform networks, the Maximum Distance version performs better in more sparse networks or if the DSAT is split into more parts. The Minimum Neighbour version performs better in more dense networks or if the DSAT is split into fewer parts. However, the difference in performance is less than 3%.

The results are summarised in Table 4.2. The average gain is approximately 12% in all but sparse network. The maximum saving could be as much as 17%.

Table 4.1: Cost comparison of DSAT parts assignment in uniform networks of various densities.

Algorithm	Maximum Distance			Minimum Neighbour		
Topology/Gain	Min.	Max.	Avg.	Min.	Max.	Avg.
Dense	3.9%	14.0%	11.1%	4.1%	15.5%	11.4%
Med. Dense	5.6%	13.6%	10.6%	5.8%	14.5%	11.0%
Med. Sparse	1.8%	15.4%	8.9%	6.0%	15.7%	10.3%
Sparse	0.0%	14.8%	7.7%	-5.3%	17.0%	0.3%



(a) Random Dense Network

(b) Random Medium Dense Network

(c) Random Medium Sparse Network

(d) Random Sparse Network

Figure 4.4: Assigning parts of DSAT to nodes in random networks.

4.4.2 Static Attribute Propagation

Static Attribute Propagation problem could be seen as a dissemination problem where each node has to disseminate its static attributes to other nodes in the network. However, there are several differences between the traditional dissemination problem and the static attribute

Table 4.2: Cost comparison of DSAT parts assignment in random network of various densities.

Algorithm	Maximum Distance			Minimum Neighbour		
Topology/Gain	Min.	Max.	Avg.	Min.	Max.	Avg.
Dense	5.1%	14.9%	12.3%	5.2%	16.3%	12.5%
Med. Dense	7.1%	16.0%	13.0%	7.1%	17.0%	12.7%
Med. Sparse	4.0%	18.8%	12.2%	6.0%	18.4%	12.3%
Sparse	2.9%	11.8%	5.8%	2.2%	10.1%	4.1%

propagation problem. First, in the case of dissemination usually *one* node needs to disseminate certain value to all other nodes while in the case of static attribute propagation problem *all* nodes need to disseminate their values to other nodes. Second, the disseminated value has to reach *all* nodes, while 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 WSNs, therefore we compare SA propagation algorithm to simple dissemination protocols. Trickle [LPCS04] is the simplest dissemination protocol where each node upon receiving a disseminated value broadcasts the value several times, each subsequent broadcast occurs after longer delay from the previous broadcast. In order to lower the number of messages, in our evaluation each node broadcasts the message only once.

Other dissemination protocols lower number of messages by overhearing dissemination messages from neighbours. A node upon receiving a message waits for a random delay. During this delay the node listens to neighbours and count how many of them broadcast the message. Let x be the percentage of neighbours that have broadcast the message at time when the random delay expires. If $x > t$, where t is a threshold, the node discards the message without broadcasting it. This approach is similar to our approach with the difference that in our case the node is aware of which neighbours might or might not have received the message.

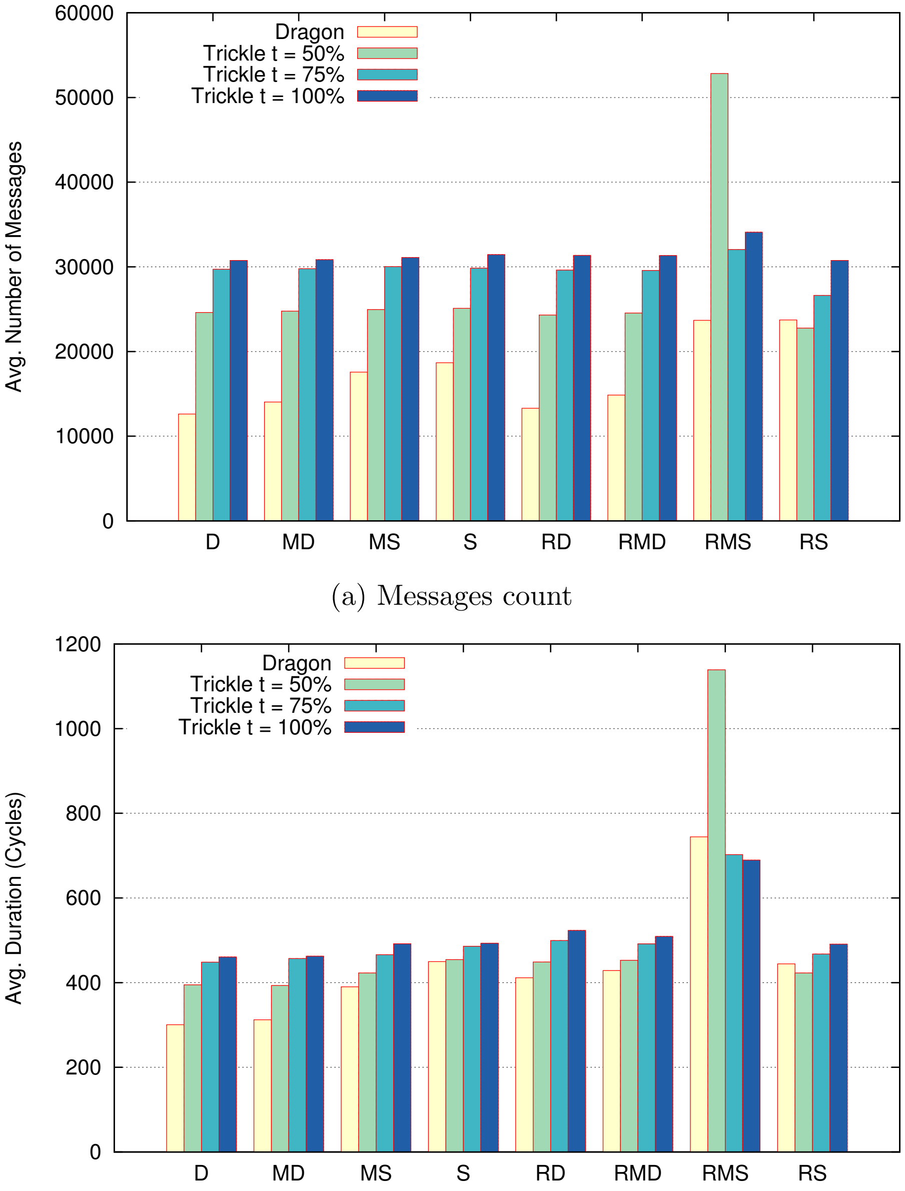
In our evaluation we vary the threshold t. The higher the threshold is, the more neighbours have to broadcast the message before the node decides to discard it. We used the following set of thresholds: $t \in \{50\%, 75\%, 100\%\}$. If $t = 100\%$ every node broadcast the dissemination message upon receiving it and therefore it mimics the Trickle algorithm. We have also tried lower thresholds $t \in \{10\%, 25\%\}$ but even though they performed rather well in dense networks, in all other network densities they were not able to converge. Algorithms with such a small thresholds were able to propagate static attributes within a small part of the network only. The nodes which have not received the data started to request the missing data directly from the source nodes which led to extremely high traffic, buffer overflows, causing the whole network to crash.

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

The results comparing number of sent messages are presented in Figure 4.5a. The results are grouped by network topology and network densities. As it can be seen from the figure, Dragon is more energy efficient than all other dissemination algorithms in all but random sparse networks, where it is slightly (by 4%) outperformed by the Trickle algorithm with $t = 50\%$. As it can be seen the gain in performance of Dragon algorithm is lower as the network become more sparse. This is understandable as in sparse network there are fewer common neighbours, therefore, more nodes are required to broadcast a message.

The comparison of Dragon and Trickle algorithm with several different thresholds is summarised in Table 4.3. As it can be seen from the table, Dragon algorithm 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 only by 4%.

By looking at the graph we can see an obvious outlier in the case of random medium sparse networks. The reason lies in the topology of these networks. The networks consist of loosely connected large clusters of nodes, at it can be seen in Figure 2.7 and 2.8 in Chapter 2. In this case, with a small threshold it is more likely to happen that static attributes about a node is



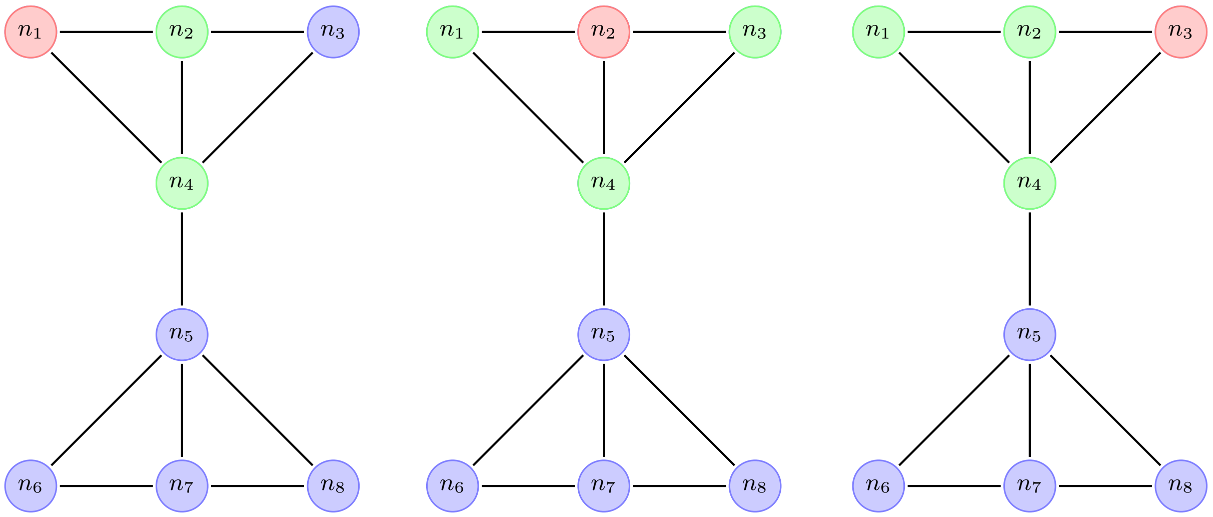
(a) Messages count

(b) Duration

Figure 4.5: Static Attribute propagation

not propagated from one cluster to another, which leads to high traffic once the nodes start to request static attributes directly from other nodes.

Figure 4.6 demonstrates how the Trickle algorithm with $t = 50\%$ may fail to propagate static attributes from one part of the network to the other. Node n_4 has four neighbours: $n_1, n_2, n_3,$ and n_5. After node n_4 receives a message from node n_1 (Figure 4.6a) it sets up a timer with a random timeout. Before the timer times out nodes n_2 and n_3 re-broadcast the message (Figure 4.6b and 4.6c). Now, when the timer at node n_4 times out, the node decides



(a) Node n_1 broadcasts its static attributes. The message is received by nodes n_2 and n_4.

(b) Node n_2 re-broadcasts the static attributes. The message is received by all its neighbours.

(c) Node n_3 re-broadcasts the static attributes. At this time 3/4 of n_4 neighbours have re-broadcast the message, therefore n_4 decides not to re-broadcast the message. Nodes n_5, n_6, n_7, and n_8 will not receive static attributes of n_1.

Figure 4.6: Problem with static attribute propagation algorithm based on neighbour threshold.

to not re-broadcast the message as three out of four of its neighbours have already broadcast the message. As a result, nodes n_5, n_6, n_7, and n_8 will never receive the message. Therefore, algorithms based on threshold are more suitable for uniform networks where most of the nodes are of the same degree.

The next metric we focus our evaluation on is the time it takes for all static attributes of each node to propagate throughout the network. As it can be seen from Figure 4.5b and the summary Table 4.4 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 $t = 50\%$. Generally, we can see the same trend 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

Table 4.3: Number of messages comparison of Dragon with Trickle algorithm for static attributes propagation.

Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
Trickle $t = 50\%$	48.7%	43.3%	29.6%	25.6%
Trickle $t = 75\%$	57.6%	52.8%	41.5%	37.4%
Trickle $t = 100\%$	59.0%	54.5%	43.5%	40.6%
Topology	Random			
Trickle $t = 50\%$	45.3%	39.4%	55.2%	-4.0%
Trickle $t = 75\%$	55.1%	49.7%	26.1%	10.9%
Trickle $t = 100\%$	57.6%	52.6%	30.6%	22.8%

Table 4.4: Time of static attributes propagation comparison of Dragon with Trickle algorithm.

Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
Trickle $t = 50\%$	23.9%	20.6%	7.7%	1.1%
Trickle $t = 75\%$	33.0%	31.7%	16.3%	7.5%
Trickle $t = 100\%$	34.8%	32.5%	20.7%	8.8%
Topology	Random			
Trickle $t = 50\%$	8.3%	5.4%	34.7%	-4.9%
Trickle $t = 75\%$	17.6%	12.8%	-5.6%	5.0%
Trickle $t = 100\%$	21.4%	15.9%	-7.4%	9.5%

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 a lot of nodes were requesting data from other nodes. This is consistent with the results of the number of sent messages.

4.5 Conclusion

Searching a WSN for nodes by static attributes is a challenging problem. Sensor nodes are very restricted in terms of memory (with only a few KB of memory) and communication capabilities (with slow low powered radio). Therefore, the nodes tend to learn only about close neighbourhood. So far, very little research attention has been paid to the solution to this problem. Researchers did not find it important to allow any node in the network to search other nodes by given static attributes. This has been also pointed out by a recent survey on in-network processing for WSNs [Kan13].

The solutions that have been proposed either rely on a central point with a greater knowledge of the whole network or the solutions did not take into account locality of the nodes and the distribution of data was not even. The central point represents a single point of failure. In case it fails, the whole network become unavailable and will not be able to accept queries. On the other hand, if data are distributed throughout the network, but the locality of the data is not taken into consideration, some nodes will have to spend more energy to perform a search than the others. Additionally, if data are not distributed evenly, hotspots may appear leading to higher network traffic and sooner battery depletion.

We argue, that this type of communication, when a user instructs a node to find all other nodes with given static criteria will become very common. Finding all flow sensors on a pipe or all temperature sensors on a floor are just two types of queries which might become common in near future. We do not see the network as a separate closed environment but rather open and interactive. And this type of queries will not be interesting only for users. As more WSNs will be equipped with actuators, a node sensing an event will want to find all actuators which fulfil given criteria. A flow sensor detecting a leak will want to send message to a valve on given pipe, a temperature sensor will want to send a message to a node controlling the HVAC, or a fire sensor will want to send a message to all sprinklers in given room. These are just a small set of examples that future WSNs will want to perform.

In this chapter we present Dragon's sub-system for distributing static attributes throughout the network. Dragon splits the table holding all static attributes about all the nodes in the networks into p parts. We refer to this table as the Distributed Static Attribute Table (DSAT). Each node is assigned to store one part of the DSAT. If a node wants to search in the table, it has to contact $p - 1$ nodes which store the rest of the table. How these parts are assigned to nodes affects how many messages a node has to send in order to retrieve data from the whole table. We proposed two algorithms for assigning parts to nodes. A node running the Maximum Distance algorithm chooses the part which is stored by a node furthest away from the node making the decision. A node running Minimum Neighbour algorithm chooses a part which is stored by least number of neighbours. We evaluated both algorithms and compared them to random assignment of the parts to the nodes. We showed that Dragon can decrease average number of messages required to search in the DSAT by as much as 18% with an average of 11%.

Once a node decides which part of DSAT it will store, it has to fill the table with static attributes of other nodes. For this purpose we proposed an algorithm for energy efficient data propagation in a WSN. Data propagation is based on overhearing broadcast of neighbours and exploiting the knowledge of the list of common neighbours. We evaluated the algorithm by comparing it with the Trickle [LPCS04] algorithm and its improved version focused on lowering the network traffic by overhearing the neighbours. We showed that Dragon lowers the network traffic by as much as 59% (with an average of 45%) in the case of uniform topologies and as much as 58% (with an average of 37%) in the case of random topologies.

Chapter 5

Snapshot Queries

5.1 Introduction

Currently WSNs are seen as single purpose platforms deployed to serve a limited group of users. These WSNs either periodically collect data or allow the limited group of users to submit specific type of queries. These queries are usually not general and the application running in the WSN support only a limited range of queries. These queries must be submitted through a more powerful node, i.e. a base-station. The base-station represents a communicating gateway by making WSN's capabilities available to outside world while translating users' requests to the WSN. The base-station usually uses a long-range communication so the users communicate with it via the Internet.

In our opinion, this type of traditional WSNs deployments will become obsolete as the WSNs will become more widespread. There will not be any need to communicate with the WSN via the Internet but users will interact directly with the notwork, e.g. using smart phones. WSNs will not be used to perform long-term monitoring tasks but will also support ad-hoc queries submitted by users, who are interested in current state of the network.

If we relate back to our scenario, an engineer may be interested in current flow readings of all sensors on a specific pipe. Alternatively, only readings from a specific range of pipe segments might be in the engineer's interest. The platform should allow any user to submit this type of ad-hoc queries and be able to response with minimal communication overhead and in timely manner. The platform should not rely on any central point or any node with a higher knowledge of the network.

In this chapter we propose such system based on top of Dragon's routing algorithm and DSAT introduced in Chapter 3 and 4 respectively. We evaluate Dragon platform using snapshot (one-time) queries and compare it with a state-of-the-art solution.

5.2 Snapshot Queries

When users are interested in the current state of the network and require the latest readings, they issue a snapshot query. We define a snapshot query as a query which is executed only *once*. The opposite of a snapshot query is a continuous query which is executed repeatedly many times (possibly indefinitely) at a predefined rate. The following Chapter 6 is dedicated to continuous queries.

When it comes to optimisation, continuous queries offer many possibilities. The query execution plan maps operators to specific nodes in the network with the aim to minimise the network traffic. However, finding the optimal placement for given operator requires time and network communication. This communication is negligible from the long-term perspective of executing the continuous query. However, in the case of a snapshot query, this optimisation could easily outweigh possible savings many times. Therefore, when a snapshot query is executed, the focus is on identifying the nodes which can contribute to the query and requesting data from these nodes with a minimal overhead.

We assume that the nodes which contribute to the query could be identified using static attributes only. We do not support searching by dynamic attributes as it would require flooding the whole network. We could say that snapshot queries test Dragon's ability to quickly and correctly identify all the nodes satisfying the query using the DSAT. Additionally, Dragon's ability to request data directly from the source nodes test its routing sub-system based on routing tables and implicit acknowledgement.

An alternative to using the DSAT are approaches based on summaries. A summary like Bloom filter [Blo70] or a histogram can point the search towards the parts of the network which contain nodes that fulfil given static requirements. Nodes in the network form a tree structure where each node holds a summary for the sub-tree rooted in given node.

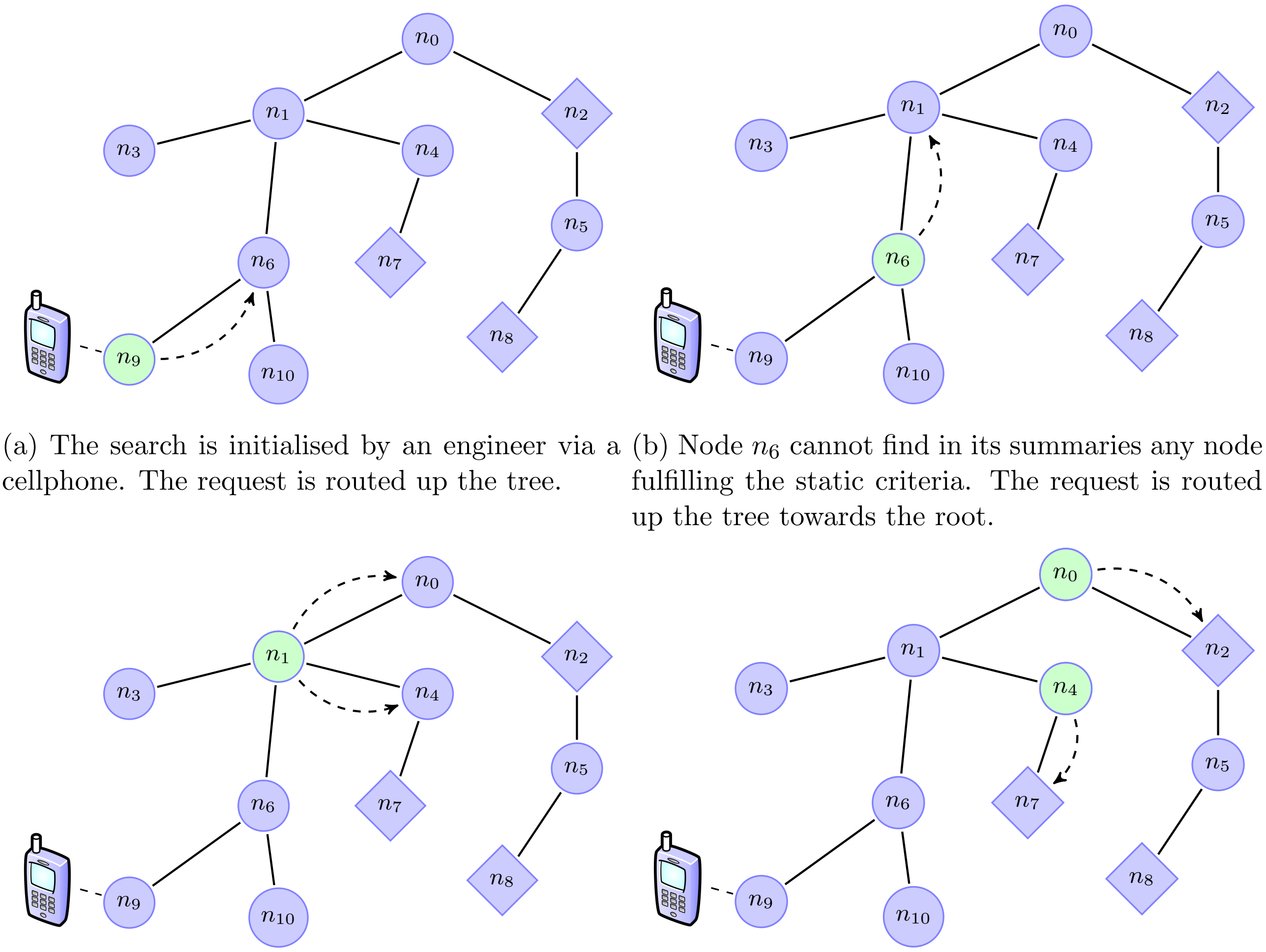
Searching for nodes with given static attributes based on summary trees is depicted in Figure 5.1. The search is initiated by a user using a cell phone and connecting to any node, in this case node n_9 (Figure 5.1a). The user sends a request which contains a list of static criteria, e.g. $pipe\_id = 25$. In the figure, nodes which fulfil given static criteria are diamond shaped (n_2, n_7, n_8). The request is forwarded up the tree until it reaches the root of the summary tree. Every node which receives a request checks the locally stored summary to find if there is any node in the sub-tree rooted in a given node which fulfils given static criteria. If there is, the request is forwarded in the given direction. This could be seen in Figure 5.1c where node n_1 forwards the request to node n_4 as well as towards the root. Similarly, in Figure 5.1d the root node n_0 forwards the request to node n_2. In order to avoid loops, the request which is forwarded downwards cannot be forwarded upwards again.

Once a node which fulfils given static criteria receives the request it replies back to the initiating node by reversing the path the request was routed through. It is possible because each forwarding node includes itself into the path vector which is a part of the request. This could be seen in Figure 5.1e where nodes n_7 and n_2 reply back to the initiating node n_9. Similarly, in Figure 5.1g node n_8 replies to the initiating node after the request was received.

There are several aspects which affect the memory requirements, the speed of the search, and the amount of traffic generated during the search for the nodes. Below we will describe these in more detail and how they affect the discovery process:

Number of summary trees

An approach based on summaries may rely on one summary tree or on several summary



(a) The search is initialised by an engineer via a cellphone. The request is routed up the tree.

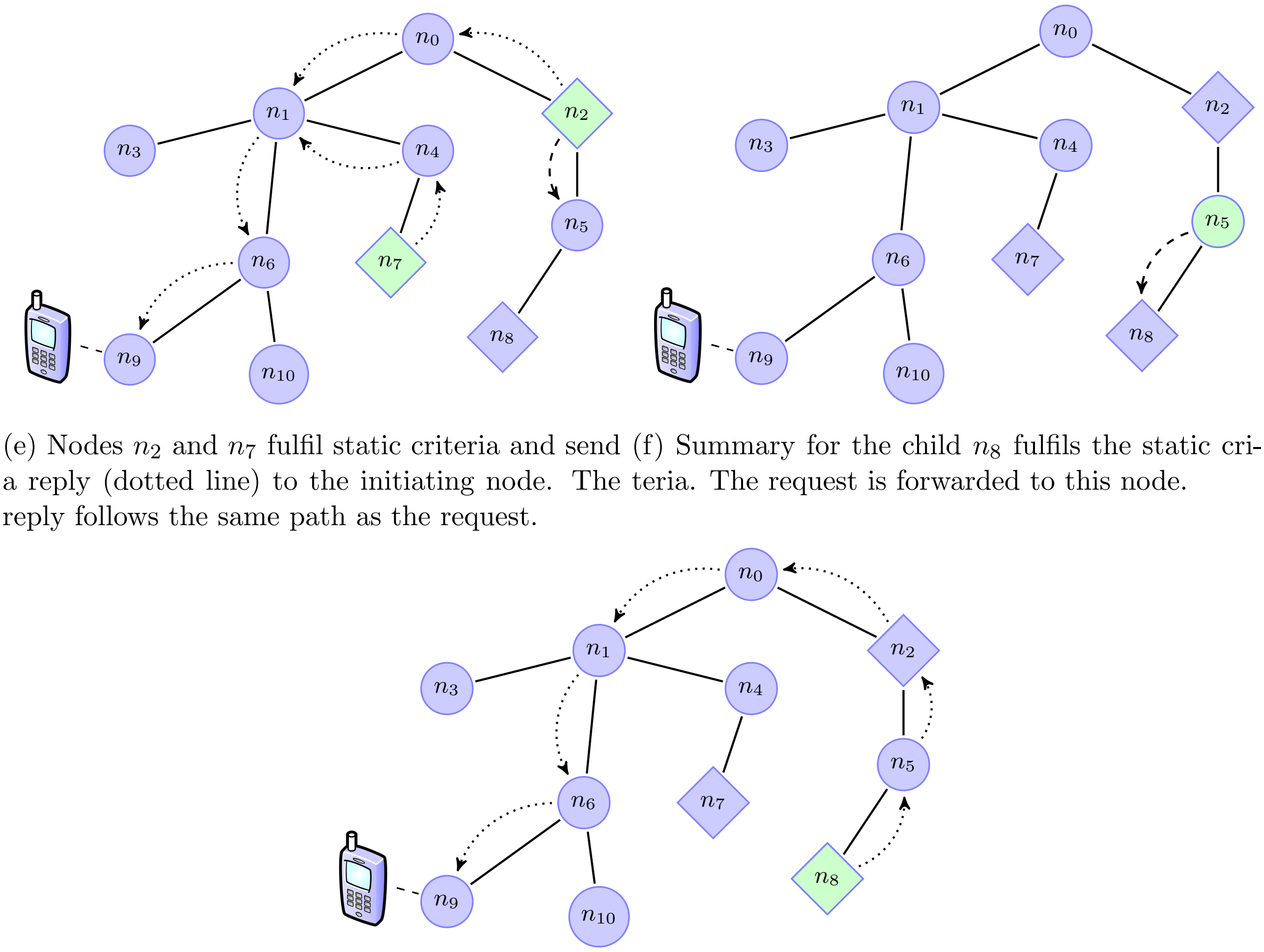
(b) Node n_6 cannot find in its summaries any node fulfilling the static criteria. The request is routed up the tree towards the root.

(c) Node n_1 found in its summary for its child n_4 a node which fulfils given static criteria. The request is forwarded to node n_4 and towards the root.

(d) Node n_4 found in its summary for the child n_7 a node which fulfils given static criteria. The request is forwarded to node n_7. The request reached the root node n_0. Summary for the child n_2 contains a node which fulfils static criteria. The request is forwarded to node n_2.

Figure 5.1: Search in tree summaries - Part 1/2. The search request (dashed line) for nodes with given static criteria is routed up the routing tree. If a node has a summary which fulfils the static criteria, the request is routed down the routing tree in given direction. Nodes fulfilling static criteria are diamond shaped. If such a node receives a request it replies via the same path as the request travelled (dotted line).

trees. If the platform uses only one summary tree, the tree is rooted at the base-station. This approach resembles the approach used in SENS-Join [SBB09] where a summary tree is used to route the query to relevant nodes. Alternatively, the platform may rely on more summary trees, with each tree rooted in a different part of the network. This approach is used in the Innet [MJIG10] platform which relies on three summary trees. Storing several summary trees has an advantage of finding the relevant nodes faster as the search is done



(e) Nodes n_2 and n_7 fulfil static criteria and send a reply (dotted line) to the initiating node. The reply follows the same path as the request.

(f) Summary for the child n_8 fulfils the static criteria. The request is forwarded to this node.

(g) Node n_8 fulfils the static criteria. The reply is sent via the same nodes as the request. Request is not forwarded as node n_8 is the leaf node.

Figure 5.1: Search in tree summaries - Part 2/2. The search request (dashed line) for nodes with given static criteria is routed up the routing tree. If a node has a summary which fulfils the static criteria, the request is routed down the routing tree in given direction. Nodes fulfilling static criteria are diamond shaped. If such a node receives a request it replies via the same path as the request travelled (dotted line).

over several trees in parallel. However, this comes at the expense of higher network traffic as three different searches are executed. Additionally, storing multiple summary trees require more memory, which is very precious in WSN nodes.

In Figure 5.1 only one summary tree is used. If the second summary was rooted in node n_9 the search for the nodes with given static attributes would be faster in the second tree as the request would have to be forwarded only downwards. It is important to note, that requests in each summary tree are routed independently.

Summary types

There are many different types of summaries, and each has its pros and cons. They differ by memory requirements, supported operations (e.g. Bloom filter does not support removing a value from the summary), precision of the summary (which is correlated with memory requirements), etc. The summary type also affects what kind of queries the platform can support. For example, Bloom filters [Blo70] can only check whether a given value is or is not in the filter (with given probability). Therefore, Bloom filters are suitable for queries based on equality operator. On the other hand, histograms can also answer range queries with inequality operator. Therefore, some platforms store several different summaries for the same static attribute.

Summary size

Summary size directly influences how many false positives a given summary will produce. With more memory dedicated to the Bloom filter or histogram it is possible to make them more fine grained and therefore point the search in correct direction.

In Figure 5.1c node n_1 forwards the request to node n_4 but not to node n_3. If the Bloom filter summary used for the sub-tree routed in node n_3 was very small, it could lead to a false positive and node n_1 would forward the request to node n_3, too.

Number of nodes in the summary

A summary may store static attributes of all the nodes in the sub-tree, or a separate summary could be held for each of the node's child. Storing one summary per a child leads to lower traffic during the search, as the search could be pushed into correct direction. However, this comes at the expense of higher memory requirements, especially if the node has many children, has to store a lot of summaries, or both.

This could be demonstrated in Figure 5.1c where node n_1 can store one summary for all the nodes in the sub-tree rooted in itself or for each child separately. If there is a summary for each child, the request is sent only to node n_4 as in its sub-tree there is a node which fulfils static criteria. However, if only one summary is used for all children, then node n_1 has to forward the request to n_3 and n_4 as it cannot determine which node is on the correct path to the nodes fulfilling the static criteria. Node n_1 does not forward the request to node n_6 as the request was received from this node. Should the request arrive from node n_0 then all three children nodes would have received the request.

In the next section we evaluate Dragon and we compare it to platforms based on summaries. These platforms have been chosen as they represent state-of-the-art algorithms for WSNs and they support search in the network without one orchestration super-node.

5.3 Evaluation

We evaluate Dragon's ability to find a list of nodes with certain static attributes and request data from them with low network overhead. We compare Dragon to approaches based on summaries. We study the influence of number of summary trees on the network traffic as well as the number of summaries hold by each node. Particularly, we evaluate the network traffic in a platform with one and three summary trees (in the figures and tables marked as "1T" and "3T" respectively). 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 each child (referred to as "child summary" (CHS)).

Number of trees and number of summaries have a large impact on the memory requirements. In our evaluation, we assigned six static attributes to every node in the network: 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. 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 as well as range queries. For each summary 16 bytes of memory is allocated. If we assume, that a node has six children on average then the cost to to store all summaries could be computed as:

$$c = \text{trees} \times \text{children} \times \text{summaries} \times \text{summary length} \tag{5.1}$$

$$c_{1TCHS} = 1 \times 6 \times 9 \times 16 = 864\text{B} \tag{5.2}$$

$$c_{3TCHS} = 3 \times 6 \times 9 \times 16 = 2592\text{B} \tag{5.3}$$

$$c_{1TTS} = 1 \times 1 \times 9 \times 16 = 144\text{B} \tag{5.4}$$

$$c_{3TTS} = 3 \times 1 \times 9 \times 16 = 432\text{B} \tag{5.5}$$

In the case of Dragon we distinguish between the memory required to store one part of the DSAT and to store the routing table:

$$c_{DSAT} = 6 \text{ static attributes} \times 25 \text{ attributes per part} = 150\text{B} \tag{5.6}$$

$$\begin{aligned} c_{RT} &= 250 \text{ number of nodes} \times 2 \text{ destination and distance} + 6 \text{ neighbours} \\ &= 506\text{B} \end{aligned} \tag{5.7}$$

$$c_{\textsc{Dragon}} = c_{DSAT} + c_{RT} = 150 + 506 = 556\text{B} \tag{5.8}$$

We evaluate Dragon abilities to answer snapshot queries by executing two queries based on our scenario. In the first case, an engineer wants to retrieve a minimum, a maximum, and an average flow from sensors on the specific pipe. We assume that every node on that pipe has a flow sensor and can provide queried for a `flow` dynamic attribute. This can be achieved by issuing a query similar to the following one:

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.

In our evaluation we focus on two metrics: *i*) the number of messages sent and *ii*) the time it takes to receive the result. The results for the Query 1 are presented in Figure 5.2. Because

Table 5.1: Number of messages comparison of Dragon with algorithms based on summaries for Query 1.

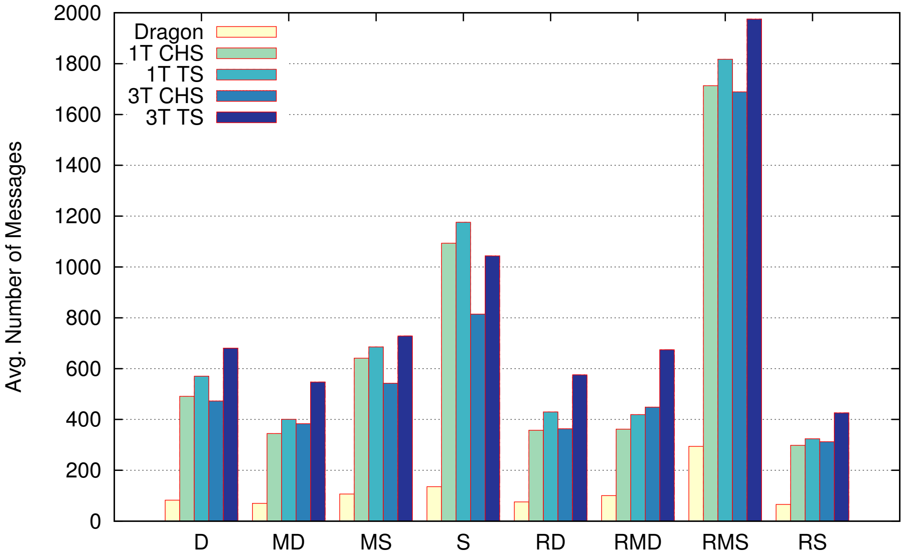
Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
1T CHS	83.2%	79.8%	83.4%	87.6%
1T TS	85.5%	82.6%	84.5%	88.5%
3T CHS	82.6%	81.8%	80.4%	83.3%
3T TS	87.9%	87.3%	85.4%	87.0%
Topology	Random			
1T CHS	78.9%	72.4%	82.8%	77.9%
1T TS	82.4%	76.1%	83.8%	79.7%
3T CHS	79.2%	77.7%	82.6%	78.9%
3T TS	86.9%	85.2%	85.1%	84.5%

Query 1 uses equality operator, the algorithms based on summaries use Bloom filters to direct the search to the correct parts of the network. The query resulted into 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.

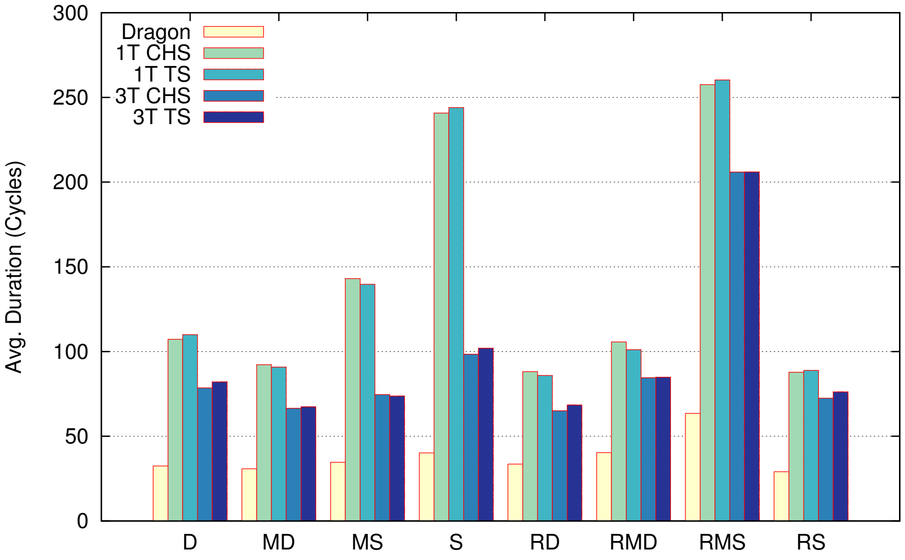
As it can be seen from Figure 5.2a (and the summary in Table 5.1) Dragon significantly outperforms all other approaches based on summaries, in terms of network traffic. Dragon can decrease the network traffic by as much as 88% with an average over 80%, depending on the network density and the approach the Dragon is compared with. Surprisingly, there is no much difference between algorithms using various numbers of trees and various numbers of summaries. This fact suggests that the Bloom filter is effective in finding correct source nodes and there are very few false positives.

The comparison of the time it took for the initiating node to retrieve the result from the network is depicted in Figure 5.2b and summarised in Table 5.2. Similarly to the 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 the detected event.

After the engineer received the results they may want to check the average flow only on segments



(a) Messages count



(b) Duration

Figure 5.2: Network traffic and execution time comparison of DRAGON to other approaches based on tree summaries for Query 1.

from their current position downstream. In order to do that a query similar to the following one may be submitted:

Query 2:

```
SELECT AVG(S.flow) FROM Sensor S WHERE S.z > @val
```

where `@val` is a random number.

The query finds all flow sensors whose `z` attribute is higher than given value `@val`. In the

Table 5.2: Time comparison of Dragon with algorithms based on summaries for Query 1.

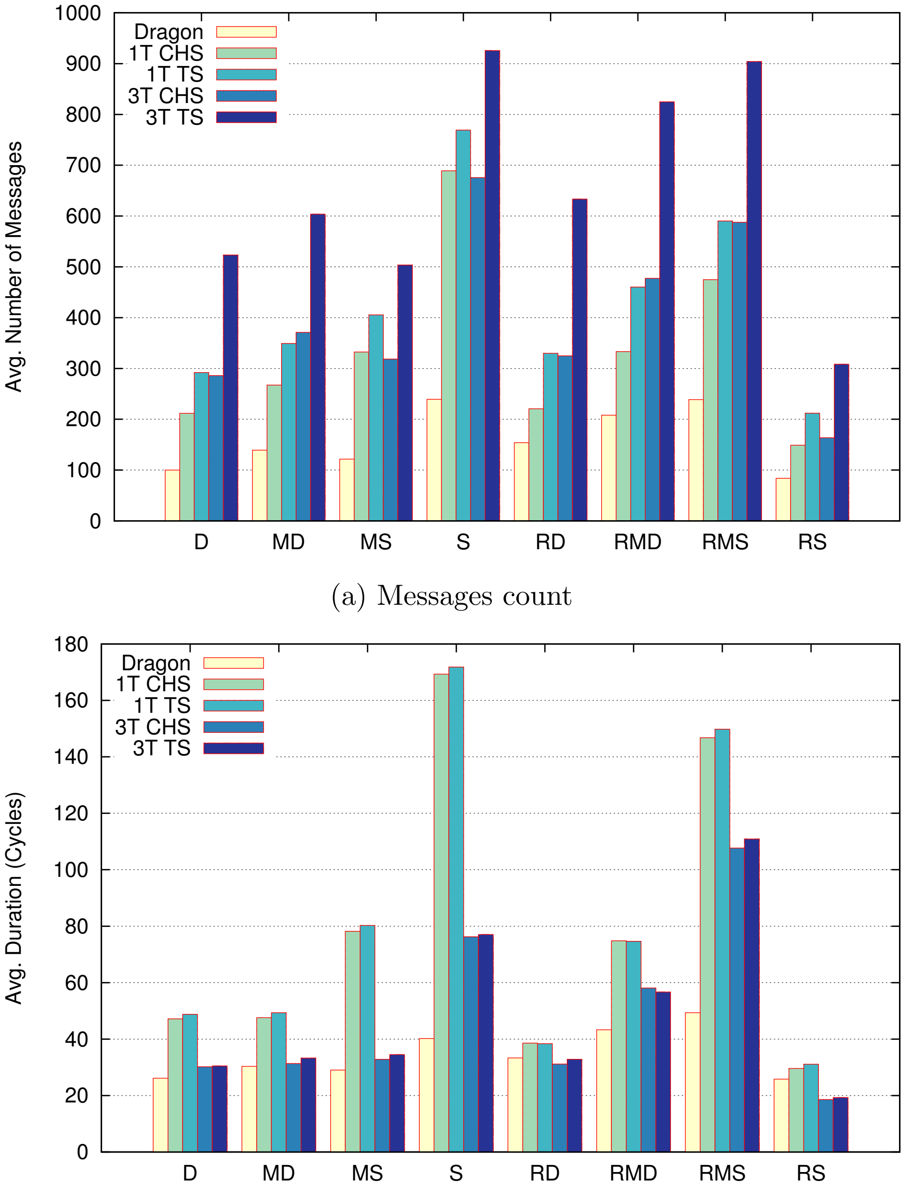
Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
1T CHS	69.7%	66.6%	75.8%	83.3%
1T TS	70.5%	66.1%	75.2%	83.5%
3T CHS	58.7%	53.7%	53.6%	59.1%
3T TS	60.5%	54.3%	53.0%	60.6%
Topology	Random			
1T CHS	62.0%	61.9%	75.3%	66.9%
1T TS	60.9%	60.1%	75.6%	67.3%
3T CHS	48.5%	52.3%	69.1%	59.9%
3T TS	51.0%	52.5%	69.2%	61.8%

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 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 given value was or was not added previously to the filter. Therefore, for this query the count histogram is used as it can identify whether in given sub-tree there are any nodes with an attribute larger than given value.

The results for the Query 2 are shown in Figure 5.3. This query results into requesting data from $2 - 18$ nodes, depending on the network. The results are grouped by network topology and density.

As it can be seen from Figure 5.3a and its summary in Table 5.3, Dragon outperforms all other approaches in terms of network traffic. The saving ranges between $30 - 81\%$ with an average of 61%. It could be seen 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 given by histograms.

In the terms of time, Dragon outperforms all other approaches almost all the time, as it can be seen in Figure 5.3b and Table 5.4. The only time 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 in DSAT takes



(a) Messages count

(b) Duration

Figure 5.3: Network traffic and execution time comparison of Dragon to other approaches based on tree summaries for Query 2.

longer than searching via summaries in a close neighbourhood. The maximum saving could be as high as 77%, while on average Dragon is 31% faster.

5.4 Conclusion

Snapshot queries are used to find out about the current state of a WSN. They are, by their nature, ad-hoc and the users are usually interested in the precise state of the network. We

Table 5.3: Number of messages comparison of Dragon with algorithms based on summaries for Query 2.

Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
1T CHS	52.8%	47.9%	63.4%	65.3%
1T TS	65.8%	60.1%	70.0%	68.9%
3T CHS	65.0%	62.5%	61.8%	64.6%
3T TS	80.9%	76.9%	75.8%	74.2%
Topology	Random			
1T CHS	30.3%	37.6%	49.7%	43.8%
1T TS	53.4%	54.8%	59.6%	60.5%
3T CHS	52.6%	56.5%	59.4%	48.9%
3T TS	75.7%	74.8%	73.6%	72.9%

Table 5.4: Number of messages comparison of Dragon with algorithms based on summaries for Query 2.

Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
1T CHS	44.6%	36.3%	62.9%	76.3%
1T TS	46.4%	38.6%	63.9%	76.6%
3T CHS	13.5%	3.2%	11.6%	47.3%
3T TS	14.3%	8.9%	15.9%	47.8%
Topology	Random			
1T CHS	13.6%	42.1%	66.4%	12.8%
1T TS	13.2%	42.0%	67.0%	17.0%
3T CHS	-6.5%	25.5%	54.2%	-27.9%
3T TS	-1.4%	23.6%	55.5%	-25.2%

believe, that as WSNs will become more popular and widespread, these types of queries will be responsible for a significant part of the queries submitted to a WSN.

Answering a snapshot query has been shown to be a challenging problem. The challenge lies in identifying all the nodes that fulfil given static requirements of the query and requesting data from these nodes. The most challenging part is to retrieve the result with low network overhead and in timely manner. In order to do so, every node in the network must be able to identify these nodes and to request data from these nodes with low communication overhead.

Various approaches have been proposed to solve this problem either by involving a base-station (e.g. SENS-Join [SBB09]) or by using in-network summaries (e.g. Innet [MJIG10]). These approaches use a tree structure where each node holds a summary for the sub-tree rooted in given node. In order to speed up the search, some approaches use more than one summary tree. On the other hand, network traffic can be lowered by using a more types of summaries for the same attribute, separate summary for each child in the sub-tree, or both.

In this chapter we evaluated abilities of the DRAGON routing algorithm and its Distributed Static Attribute Table (DSAT) sub-systems to answer snapshot queries. We compared it with state-of-the-art protocols focusing on snapshot queries. In our evaluation we used two queries, the first one uses an equality operator while the second one an inequality operator. In the case of the first query, the platforms based on summaries rely on the Bloom filter summary, while in the case of the second query they rely on count histograms.

Our evaluation focused on two metrics: *i*) network traffic and *ii*) network delay. DRAGON decreases the network traffic by as much as 80% on average for the first query and 61% for the second query. In terms of the network delay, DRAGON fetched the data from the relevant nodes on average 64% faster for the first query and 31% for the second one.

Chapter 6

Continuous Queries

6.1 Introduction

In the previous chapter we have shown how any node in the network can provide a user with current readings from all the sensor nodes satisfying user's criteria, while imposing only a very low communication overhead. It has been achieved by allowing any node in the network to find a list of all the nodes satisfying static criteria by looking them up in the DSAT and requesting data from these nodes via near-optimal paths.

Requesting data directly from the relevant sensor nodes is energy efficient in case when a user is interested only in the current readings. However, when a user wants to be continuously updated about given phenomenon for a longer period of time (possibly indefinitely), it is highly probable that processing data inside the network will lead to lower network traffic, as opposed to repeatedly retrieving data directly from the source nodes. Processing data inside the network could also be energy efficient if the network is used to detect a rare phenomenon. For example, we are interested only in data, from which a pipe burst can be detected. Because we assume that the pipe burst will occur very rarely, a lot of nodes' energy is wasted on sending data that show normal operation.

To better demonstrate what kind of continuous processing could be done on data streams from

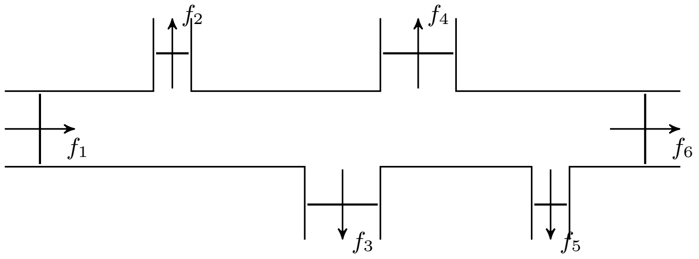


Figure 6.1: An example of a pipe segment with flow sensors. The arrow shows the direction of the flow of liquid in the pipe. Bars perpendicular to the arrows denote flow sensors.

a WSN, lets revisit our scenario. The company has already deployed sensors which monitor flow of oil through the pipes. Each pipe is split into segments. An example of a pipe segment could be seen in Figure 6.1. In the figure we can see a main trunk pipe with four branch pipes, each of which has its own flow meter. During the normal operation, i.e. without a pipe leak, the volume of oil passing through meter f_1 must be exactly the same as the sum of volumes of oil passing through all other flow meters $f_2, \ldots, f_6$.

The simplest solution to detecting a pipe leak is to periodically collect all data from the network at a base-station, transfer these data to a cloud, where an application groups data by segments and computes the differences of volumes in each segment. If a leak is detected, a message to a control centre is sent. The control centre then dispatches an engineer to fix the problem.

However, this approach has two major disadvantages. The first one is the large amount of unnecessary traffic. Transferring all data from a network into a cloud is rather expensive in terms of the energy spent on communication. And because we assume that pipe bursts are rare events, most of these sensor readings are transferred unnecessary.

Another disadvantage of collecting all data from the network is the *processing latency.* First collecting data via a multi-hop link, then sending them to a cloud and waiting for an application to process the data may incur long delay. Especially, if, for example, the connection between the base-station and the cloud fails.

However, it is not necessary to periodically ship all data to a cloud to perform a simple sum of several values. When a node is installed it is associated with a specific pipe and a specific segment. This information is part of its static attributes. All sensor nodes from the same segment can send data to a common node which can process them. We refer to this common node as the *processing node.* Once the processing node detects a burst, it can either report the burst back to the user via a base-station, or send an actuation message to an actuation node controlling a valve upstream. The actuation node will close the valve and stop the leak.

However, the decision of which node will be chosen as the processing node is not trivial and has to be done carefully with the objective to lower the network traffic. Choosing the wrong node can easily lead to an increase in the network traffic as well as the processing latency.

When a node is chosen to process data streams it has to dedicate some of its computational power and memory to perform the computation on the data streams. How much CPU and memory is needed depends on the algorithm processing the data, the rate at which data are arriving, and the number of data streams. A network may be *homogeneous*, i.e. it consists of the same type of node, or it may be *heterogeneous*, i.e. it consists of several different types of node. The heterogeneity of the network may be caused by the design of the network, when a person responsible for the design of the network chose different types of nodes on purpose, or it may be caused by extending the network with a newer type of the sensor nodes after the initial WSN deployment. This heterogeneity of a WSN brings another challenge in finding the processing node, as only a small subset of the nodes may be capable of processing the query.

In this chapter under *processing* we understand it to mean any type of operation applied to any number of data streams. Processing includes, but is not limited to: *i*) aggregation functions like maximum, minimum, average, or sum of several numbers, *ii*) more complex functions like compression, or *iii*) join operation as known from traditional databases.

We do not compare Dragon to algorithms which focus on aggregation functions only as they support only a specific type of query. Additionally, these approaches usually exploit historical summaries to answer the queries (e.g. M2 [UTK13]) as opposed to exact readings. We also do not compete with approaches which heavily rely on a base-station, e.g. Continuous Join Filtering (CJF) [SBB10], as we propose a fully decentralised approach with no single point of failure. Additionally, we also do not compare to approaches which require the nodes in the network to be reprogrammed upon receiving a query, e.g. SNEE [GBG$^+$11], as we propose a platform where users can communicate with a network via any node and can receive a reply instantly without the need to update the programme running on the nodes. Finally, we also exclude the whole family of inter-region based algorithms, e.g. Distribute-Broadcast Join [CG05], Mediated Join [CNS07], or Distributed Index-Join [PG06], as these approaches are limited to joining data from two non-overlapping regions only.

In the rest of this chapter we describe and evaluate our Processing Node Discovery Algorithm for homogeneous and heterogeneous networks. We present one algorithm for homogeneous networks and three algorithms for heterogeneous networks. We compare their performance in terms of energy spent on discovering the processing node and the optimality of the discovered processing node. We compare our solution with the state-of-the-art frameworks for WSNs.

6.2 Related Work

The first approaches to perform in-network data stream processing focused on joining of a data stream generated by a sensor and a *fixed set of values*, i.e. a small relational table. Madden *et al.* considered joining of a data stream and a *storage point*, i.e. a single node which holds a list of values the data stream should be joined with. The storage point can be seen as a materialised view known from relational databases. Data from a sensor are routed up the *routing tree* and as they reach the storage point the data are joined with a fixed set of values. This type of joining was later used also in Madden's *et al.* later work, called *TinyDB* [MFHH03], where they introduced the idea of seeing WSNs as relational databases. TinyDB accepts SQL queries which are pushed to the relevant sensors in the network and data are periodically retrieved at the base-station. TinyDB lowers the network traffic by retrieving data only from sensor nodes which can contribute to the query and only if the sensed value fulfils the condition specified in the query. However, the decision whether to send or not to send the data must the node be able to do without communicating with other nodes.

Similar approach was adapted by Adabi *et al.* in their REED [AML05] framework which joins sensed data with an *external relation*. This external relation represents a set of events the network was deployed to detect. REED pushes this external relation into the network. Depending on how big the external relation is, REED distinguishes between three scenarios: *i*) the external relation fits into a memory of a single node, *ii*) the external relation can be partitioned amongst a small set of neighbouring nodes, and *iii*) the external relation is too large and has to be partitioned across larger part of the network. REED uses various techniques, e.g. group formation, relation partition, Bloom filter [Blo70], partial join, etc., in order to lower the network traffic.

The approaches described above are able to operate on single data stream only. The data stream is joined with a pre-defined set of values, not a dynamic stream of data. If the query operates on two or more data streams, these approaches are not able to perform in-network computation and ship all data to the base-station where final processing takes place. This is the reason why other approaches for in-network data stream processing were proposed. The research work presented below focuses on frameworks capable of processing *joins over several sensor data streams*, which are well known from relational databases.

There are many various ways how the in-network join of sensor data stream algorithms could be categorised, e.g. according to join types, filtering approach, base-station involvement, adaptation to changes, cost-based optimisation, query dissemination, join initiation, routing protocol, collection of statistics, failure handling, or duration of join execution [Kan13]. Here, we briefly characterise the categorisation according to join types and then we describe various joining algorithms categorised according to the filtering approach.

Initially, large interest was dedicated to joining data streams on *spatial predicates*, i.e. the *location* (geographic coordinates) is the joining condition. The most common type of join is *inter-region join* where two disjoint sets of sensors from two non-overlapping areas join their data streams. Because every node is aware of its geographic position it can decide whether it participates in the query or not without communicating with other nodes.

Later, the interest of researchers was pointed towards joining data streams on *temporal predicates*, i.e. on *time-based windows*. Each sensor reading is timestamped and the joining node keeps a history (i.e. window) of n readings for each sensor participating in a query. Join is performed either on *fixed, sliding,* or *jumping* window of sensor readings.

Snapshot queries may have specified *fixed* window of sensor readings. The join operates only on fixed number of values which are not updated. The most common is the *sliding* window which is either specified by its size or by its time. As the new tuples are arriving they replace the old ones. *Jumping* window is similar to the sliding one, however, the window is replaced as a whole, not continuously as in the case of the sliding window.

According to *sensor tuples filtering* we can split joining algorithms into two groups: *i*) algorithm without filtering of non-joinable tuples before the final join and *ii*) algorithms which filter non-joinable tuples before the final join. Next, for each group we list and describe several in-network joining algorithms.

6.2.1 Algorithms without Filtering of Non-Joinable Tuples

Algorithms presented in this section do not support the filtering of non-joinable tuples before the final join, i.e. if the joining condition is based on *dynamic attribute* a node cannot decide apriori whether the sensed tuple will or will not join with other tuples. Algorithms from this group focus on the optimal location of the join nodes, lowering the cost of join initiation, or indexing of join tuples.

Chowdhary and Gupta proposed *Distribute-Broadcast Join* [CG05] which focuses on inter-region join R $\bowtie$ S of continuous join query. All data from region R and S are shipped to the join area P which is within a triangle formed by the region R, S, and the base-station B. The location of the join region P depends on the size of individual regions and the size of the output of R $\bowtie$ S. Distribute-Broadcast Join relies on geographic routing protocols such as GPSR [KK00] and TBF [NN03].

While the previous approach focused on continuous queries, Coman *et al.* proposed the *Mediated Join* [CNS07, CN07] which executes snapshot (one-shot) queries submitted through a base-station. It works on a similar principle as the Distribute-Broadcast Join algorithm, i.e. it collects data from regions R and S in a region P, which is in the middle of the triangle formed by R, S, and the base-station. Data are first requested from one region and are distributed amongst the nodes in the join region P. Next, data are requested from the second region and are distributed amongst the nodes holding the data from the first region. The size of the join region, hence the amount of communication, depends on the size of the first region. The mediated join takes the size of the region into consideration and chooses the best strategy for joining the data.

Another approach for region-based queries, this time for continuous queries with a *range* joining predicate, was presented by Pandit and Gupta. The *Distributed Index-Join* [PG06] algorithm is based on a *distributed B^+ tree* implemented in a WSN. Tuples from both regions are stored in a distributed B^+ tree where pointers to the new branches in the tree are geographic locations. Therefore Distributed Index-Join relies on geographic routing. Since the join predicate is a range, it is easy to find the lowest tuple which satisfy the predicate and then follow the siblings until the largest tuple satisfying the predicate is found.

In addition to Distributed Index-Join, Pandit and Gupta also proposed *Distributed Hash-Join* [PG06] to solve the same problem, i.e. continuous join queries with *range* join predicate. The algorithm partitions sensed values from both regions R and S on the joining attribute using the same hash function. If the joining operand is equality (i.e. an equipping) the output of the hash function is a random geographic coordinate. In case the joining operand is inequality (i.e. a range query) a *locality preserving* hash function is used, which hashes similar values to the close proximity locations.

So far, all of the approaches presented in this section rely on geographic location and use geographic routing protocols to route data from a source to a destination. Next, we will present an in-network join algorithm which is based on *routing trees.*

Pair-wise Join [MJIG08, MJIG10] proposed by Mihaylov *et al.* is a framework for long-running continuous queries execution. Unlike other approaches, it is not region-based, therefore it supports other than location based joining conditions. The framework is built on top of *several routing trees* where every node in a network stores summaries of static attributes for all nodes in a sub-tree rooted in given node. Summaries include Bloom filters [Blo70], histograms, R-

Trees, etc. These summaries allow any node in the network to initialise a search for another node(s) based on static attributes, i.e. find other sensor nodes with whom the node should join its data stream. It also allows a node to find out that there are no other node participating in the query, hence it should ignore the query. The decision, whether a node should or should not participate in the query cannot be made by a node locally and it requires to do a search in the network.

After an initiating node finds a set of target nodes with which it should join data, one *join node* is chosen on each path between the initiating node and every target node, i.e. there are so many join nodes as there are target nodes. When choosing the location of the join node, the distance to the nodes participating in the query as well as the distance to the base-station is taken into account. Additionally, the selectivity of the initiating node, the target node, and the join has an impact on the location of the join node. Each join node joins only one *pair* of values - one from the initiating node and one from one of the target node. If the pair fulfils the join condition it is sent to the base-station which then performs the final join of all of the partial pair-wise joins.

Abrams and Liu proposed *Greedy is Good (GIG)* [AL06] algorithm which answers continuous queries. The goal of the algorithm is to find *one join node* which is in the *centre* of all nodes participating in the query. As a metric, GIG uses the number of hops between the nodes. This join node collects data from all source nodes, computes the join, and the result is sent to the base-station. GIG finds the join node by flooding the network from every node participating in the query. First, every source node broadcast a discovery message. The discovery message is broadcast one hop further from the source node every *round*. A node which first receives a message from every node participating in the query declares itself as the join node. The setup phase of this algorithm is rather expensive as the whole network is flooded n times, where n is number of nodes participating in the query. Additionally, the selectivity of the source nodes is not taken into account. Finally, the algorithm assumes that all the source nodes are located within a small region of the network.

Inspired by GIG, Chatzimilioudis *et al.* proposed a distributed framework [CCGM13] which tackles some of the problems experienced by GIG. While they still assume that all the source nodes are located within a small region of a network, they avoid flooding the network. Flooding is avoided by first computing maximum distance of the join node from a source node. The maximum distance *d* depends on the distance between the source nodes, which researchers assume that every node knows. When a source node starts a search for the join node it includes the maximum distance *d* into the discovery message. Once the message has been broadcast *d* times, it is discarded. This way flooding of the whole network is avoided. Additionally, when choosing the join node, selectivity of the source node (known apriori by the node) is taken into account. The results showed, that under those conditions the algorithm was able to find the optimal join node in more than 90% of the cases.

6.2.2 Algorithms with Filtering of Non-Joinable Tuples

Algorithms presented in this subsection are using various techniques to filter non-joinable tuples. Usually, the base-station or other node with a better knowledge of the network status produce some kind of filter which is pushed to the sensor nodes so it can check the sensed value against the filter and decide whether to send it to the join node or not.

Synopsis join [YLZ06], proposed by Yu *et al.* , focuses on answering snapshot inter-region equijoin queries R $\bowtie_{A=B}$ S. Nodes in both regions use a hash function to transfer value of joining attribute into a geographic coordinate in a region T located between regions R and S. Part of the *synopsis* the nodes send to the join region T is a histogram of values of the joining attribute along with some other auxiliary information. The nodes in T region check the joinability of the values. If the tuples are joinable they inform nodes in R and S to send full information to a node J. The node J is chosen in the middle of the path between the joining nodes, whose exact position is sent as a part of the synopsis.

Local Semijoin [CNS07], proposed by Coman *et al.* , answers a snapshot inter-region equijoin queries R $\bowtie_{A=B}$ S. First, all joining attribute values along with the nodes' IDs from the nodes in the region R are collected and sent to the region S. The values are compared with the local values and joinable tuples are found. The IDs of the joinable tuples are sent back to the region

R. The joinable nodes from both regions send data to the base-station which performs final join.

Coman *et al.* also proposed *Mediated Semijoin* [CNS07] where another region T, between regions R and S, is chosen to receive joining attribute values and to check joinability of the nodes. Next, the result is sent back to both regions. However, as these algorithms were developed for rather small regions R and S, it has been shown that the optimal location for T is either region R or S. Therefore, the Mediated Semijoin was always outperformed by Local Semijoin.

In-Network Join strategy using Cost based optimisation in Tree routing sensor networks (INJECT) [MYC11] proposed by Min *et al.* answers continuous join queries R $\bowtie$ S. INJECT uses routing trees to collect and transfer data within the network. INJECT considers three joining strategies: *i*) Partition Join, *ii*) Synopsis Join, and *iii*) Full Synopsis Join. Statistics of node and join selectivities are collected in order to choose the correct joining strategy. In *Partition Join* tuples from region R are shipped via the base-station to the region S. Non-joinable tuples are filtered out and the result of the join is sent back to the base-station. In the case of *Synopsis* and *Full Synopsis Join* only Bloom filters of the join attributes are sent to the region S instead of tuples.

Two-Phase Self-Join (TPSJ) [YLOT07] proposed by Yang *et al.* answers time-based sliding window continuous join queries. Data are routed via routing trees. Unlike most of the previous algorithms the networks is not split into two non-interleaving regions but the nodes participating in the query are randomly placed within the network. In the first phase, the query is injected into the network and values from nodes which satisfy selection predicate are retrieved at the base-station. In the second phase the base-station injects the query along with the values collected in the first phase back to the network. Join is carried out at every node and the result is sent back to the base-station. TPJS is focused on monitoring applications where selection predicate from the first phase is highly selective and returns only a small set of values.

SENS-Join [SBB09] introduced by Stern *et al.* answers snapshot queries R $\bowtie$ S where the join predicate could be *any* attribute, including *dynamic* ones. The join computation consists of three steps. In the first step all nodes form a routing tree rooted at the base-station. During this phase the join attributes are collected. In the second step the base-station checks the joining attributes, computes *join filters*, and pushes these filters back to the network. In the third step nodes that are not included in the filter send values to the base-station which computes the final join.

In the first and second step the data and filters are sent in a compact form using *quadtrees* [Sam84] and *Z-ordering* [Mor66]. Using these summaries encoding significantly reduces amount of network traffic required to collect joining attributes from the network and distributing join filters back to the nodes. However, how many nodes will send data to the base-station in the third steps depends on the fidelity of these summaries and if the size of the summary is too small or the network is too big, it may lead to many unnecessary nodes to participate in the query.

Continuous Join Filtering (CJF) [SBB10] proposed by Stern *et al.* is an extension of their previous work *SENS-Join* to answer continuous queries. Under ideal conditions, a node in the network sends value to the base-station only if it is joinable. CJF was designed to get as close to these ideal conditions as possible. Please remember, that the joining attributes are dynamic (i.e. sensed value).

To exclude the non-joinable tuples from being sent to the base-station CJF installs *join filters* at every node in the network. The filter is in a form of intervals $\langle v_{min}, v_{max} \rangle$. If the sensed value v falls within given interval, i.e. $v_{min} \leq v \leq v_{max}$, the value is filtered out and is not sent to the base-station. Otherwise, the value is sent and the base-station performs final join. A *collision* is a state when the base-station receives a value u which might be joinable with a value that might have been filtered out. In this case the base-station requests the filtered value directly from the node.

All filters are computed by the base-station and pushed into the network. The base-station keeps all filters in order to be able to detect collisions. Filters must be non-overlapping. CJF is effective in lowering the network traffic in scenarios where the joining attributes do not change rapidly (e.g. temperature). CJF is also able to update the join filters as the sensed value is gradually changing using a linear regression model.

Progressive Energy-efficient Join Algorithm (PEJA) [LCC08] proposed by Lai *et al.* executes continuous inter-region equijoin queries R $\bowtie_{A=B}$ S. In each region a routing tree is constructed spanning all nodes within given region. The joining attribute is split into sub-ranges and each node constructs a histogram which is sent to the root node. The root node in each region merges all histograms together and send it to a common node c, which is in the middle of the path between the root nodes. The common node c then uses both histograms to find mergeable intervals. These mergeable intervals are then sent to both root nodes. Root nodes distribute the mergeable intervals within their regions. Next, the whole network is partitioned into a *grid* and nodes use geographic hashing which map each interval into particular grid. If interval I from region R is mapped to grid g then the same interval from region S is mapped into a *mirror grid $g\prime$*. Finally, nodes from grid g and $g\prime$ exchange values and the result is sent to the base-station.

Synopsis Refinement iceberg-Join Algorithm (SRJA) [LLG10] also proposed by Lai *et al.* and focus on answering snapshot inter-region equijoin *iceberg* queries R $\bowtie^{i}_{A=B}$ B. An *iceberg join* [FSGM^{+}99] for an attribute value v is a join for which number of joined tuples exceeds specific *iceberg threshold* α. The idea behind iceberg join is to find a pattern of correlation among the sensor reading.

The joining process of SRJA is similar to the one described in PEJA. First, routing trees for each region are constructed. Each root node collects a histogram which in addition to *count* for each sub-range contains also counts for the minimum and maximum value from given interval. These extended histograms are sent to the common node c located in the middle between the roots of two regions. The common node, depending on the counts for each interval, flags each interval with *PRUNE*, *JOIN*, or *DIVIDE*. The flagged histogram is sent back to the roots. The root disregards all values from the interval marked as PRUNE. Values from a interval marked as JOIN are joined between the regions and the result is sent to the base-station. The interval marked as DIVIDE is split into smaller intervals and the same process is repeated.

SNEE [GBG^{+}11] proposed by Galpin *et al.* describes SNEEql continuous declarative query language. A query submitted by user in the SNEEql language is translated into Query Execution Plan (QEP) in a form of a routing tree. The QEP is optimised and the system generates source code for each node in the network. The framework must have a knowledge about the network topology prior to generating the source code. The generated source code is compiled and uploaded to the nodes. Because the base-station has to have knowledge about the whole network and different source code is generated for each query, SNEE does not support ad-hoc queries. Additionally, SNEE does not have a mechanism to respond to node failures. If a node on a critical path fails the base-station has to generate a new QEP and disseminate it into the network.

The problem with node failures was addressed by Stokes *et al.* who proposed a Proactive Adaptation in the case of a node failure [SPF14]. The authors proposed to generate several alternative QEPs at compile time and disseminate them all together. They also studied under what conditions the QEP should be switched to an alternative one: after the node failure or prior to the node failure. The experiments suggested that proactive (i.e. prior to node failure) switching between QEPs provides better results.

6.3 Heterogeneous Networks

So far, not much attention has been paid to heterogeneous WSN. Heterogeneity of a WSN can be caused by various factors. A network can be heterogeneous due to the different amount of residual *energy* available. This is a common case of networks where some nodes are utilised more for relaying data, e.g. a node acts as a cluster-head or it is close to a base-station in a network using routing trees. Additionally, some nodes in the network might by connected to a power supply or their batteries are repeatedly changed.

Another type of heterogeneity is caused by different *hardware specifications* of the node itself, e.g. a node might have a larger memory or a more powerful CPU. This scenario might occur when a network is extended with a newer type of nodes, or, various entities are responsible for different parts of the network. For example, a water pipe network and a gas pipe network may cooperate and relay information for each other. Some networks might be deliberately designed to include more powerful nodes, capable of processing more data streams or storing more historical data.

Lastly, heterogeneity can be caused by different supporting several radio transceivers. Some nodes may be capable of long-range or high-bandwidth communication. These nodes can increase throughput of a network and decrease the end-to-end latency as a packet needs to travel through fewer hops before it reaches a base-station.

Vast majority of research of heterogeneous WSN focuses on routing protocols, more precisely routing protocols based on clustering [SJKZ11]. As we have mentioned before, routing protocol based on clustering are used to forward data to a base-station only. In each round a cluster-head is elected which during this round collects and aggregates data from other nodes in the same cluster. Aggregated data are then sent to a base-station. As the cluster head must stay awake during the whole duration of the round, its battery depletes faster. Therefore, the head election should take into consideration residual energy of the node.

Vast majority of the algorithms proposed to solve this problem focus heterogeneity of energy resources only. Amongst others *Energy efficient heterogeneous clustered scheme (EEHC)* [KAP09] proposed by Kumar *et al.* , *Developed Distributed Energy-Efficient Clustering (DDEEC)* [ESEFA10] proposed by Elbhiri *et al.* , *Stochastic Distributed Energy-Efficient Clustering (SDEEC)* [ESA09] proposed by Elbhiri *et al.* , *Improved and Balanced LEACH (IB-LEACH)* [HL] proposed by Hssane and Lahcen, or *Multi-hop communication routing (MCR)* [KAP11] proposed by Kumar *et al.*

Guilherme *et al.* studied the domain of distributed storage protocol for heterogeneous WSN. In their paper they propose *ProFlex* algorithm [MGV$^+$13] which solves the problem of distributing sensed data throughout a heterogeneous WSN. The objective is to distribute sensed data in such a way that a *mobile sink* travelling randomly through the network can receive maximum of the sensed data by visiting minimum number of nodes. ProFlex assumes the network consists of many low-end sensors, referred to as *L-sensor nodes*, and a small number of high-end sensors, referred to as *H-sensor nodes.* H-sensor nodes are equipped with two radios: one which allows it to communicate with other L-sensor nodes within radius r_L, and the second radio allowing it to communicate with other H-sensor nodes within radius r_H, while $r_L \ll r_H$.

ProFlex algorithm works as follows: all L-sensor nodes form a tree around closest H-sensor node. L-sensor nodes then forward sensed data to the root, i.e. the H-sensor node, which subsequently forwards data to other H-sensor nodes. Other H-sensor nodes then distribute the sensed data from other trees within their own tree. A mobile sink is then able to retrieve large part of all sensed data by visiting only a small random part of the network.

To the best of our knowledge there no prior work on in-network processing in heterogeneous WSNs.

6.4 Processing Node Discovery Algorithm

A user can communicate with any node in the network and submit queries. The node which 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 follows the same procedure as in the case of a snapshot query described in previous chapter. In summary, the initiating node search in the DSAT for other nodes which satisfy static attributes of the query. These nodes are referred to as *sources.* Each source produce 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 temperature > 23`. Here, the senor node sends the data tuple only if the sensed temperature is higher than 23°C. Let us define *selectivity* σ as:

$$\sigma = \frac{\text{tuples sent}}{\text{tuples sampled}} \tag{6.1}$$

Because the 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 is able to compute its selectivity (e.g. using pre-stored historic data or a histogram), it reports the selectivity back 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 which can process the data streams. We refer to this node as the *processing node.*

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

$$c_i = \sigma_S r_i + \sum\nolimits_{j \in S} \sigma_j d_{ij} \tag{6.2}$$

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.

As we have described in Section 3.4, the reliable transmission of a packet requires an acknowledgement packet sent by the receiver to the sender. However, by overhearing the receiver's communication, it is possible to decrease the number of messages required to deliver a message between the two nodes h hops away while sending only $h+1$ messages, instead of $2h$. 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*:

$$rc_i = \sigma_S rr_i + \sum\nolimits_{j \in S} \sigma_j dr_{ij} \tag{6.3}$$

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 selectivity of the processing node is very low, e.g. if detecting a pipe leak, the part 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. Lets 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 the evaluation 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.

Having only one processing node for a query has a several advantages as well as disadvantages. The biggest advantage is that one node receives data from all the source nodes and can perform any type of computation on the received values. Additionally, the final result is produced and there is no need for a base-station to process partial results. However, processing a query on a single node brings also several disadvantages. Amongst the main disadvantages belong: *i*) no ability to filter out values before they reach the processing node, *ii*) possible congestion around the processing node, and *iii*) limited number of source nodes depending on the computational capabilities of the processing node. Below, we describe each of these disadvantages in more detail.

When all sensed values for a given query are delivered to one processing node, it may lead to higher network traffic. In approaches, where the sensed data are routed towards a processing node in a tree-like structure [SBB10, GBG$^+$11] or a pair of values towards one node [MJIG10], some of the sensed values might be filtered out before they reach the processing node. Filtering out the values closer to the source nodes may lead to lower overall traffic in the network. Another disadvantage of a single processing node is that it could lead to network congestion around the processing node which may lead to sooner battery depletion of the processing node and the nodes in the close neighbourhood of the processing node. Finally, the number of nodes participating in a query is limited by the memory and computational capacity of the processing node. In this thesis we focus on subset queries, where only a small subset of nodes participate in the query. Therefore, we assume that in the case of a homogeneous network every node can process a query submitted by a user. In the case of heterogeneous networks we assume that there exists at least one node capable of processing the query.

We present algorithms for two different types of networks: *i*) *homogeneous* and *ii*) *heterogeneous* network. Recall, in the case of homogeneous network each node has the same processing and memory capabilities. In the case of heterogeneous networks we assume that some of the nodes have higher computational capabilities in terms of CPU, memory, or both. Therefore while in the case of homogeneous network we assume that every node is capable of processing the data streams, i.e. the search space consists of every node $n \in N$, where N is the set of all nodes in the network, in the case of heterogeneous network only a subset of the nodes are capable of processing the data streams. Which nodes are capable of processing the data streams is given by the number and selectivity of source nodes participating in the query. The more sources participate in the query, the more memory and CPU is required in order to process the data streams. For every query q we can split the set of all nodes N into two disjunctive subsets: *i*) N_h^q - the set of *high* nodes capable of processing the query q and *ii*) N_l^q - the set of *low* nodes not capable of processing the query q. Then the search space is limited only to the subset N_h^q. It is important to note, that nodes in this subset are not necessary neighbours and there is an arbitrary distance between them.

In the rest of this section we first describe how a query is represented in Dragon and the lifetime of the query in our platform. Next we describe four different join node discovery algorithms: one for homogeneous networks and three for heterogeneous networks.

6.4.1 Query Processing Overview

In this section we describe how DRAGON processes a query submitted by a user. A query is represented by a C structure containing following information: *i*) the list of attributes the user is interested in, i.e. the `SELECT` clause, *ii*) the joining condition, i.e. the `JOIN` clause, and *iii*) the list of restrictions, i.e. the `WHERE` clause. This simple structure was chosen as it can easily represent an evaluation query. We would like to emphasise that parsing of a general SQL query is not the aim of this thesis. On the other hand the structure allows developers to easily extend it and add another functionalities, e.g. more complex processing of the sensed data.

Once a node, referred to as the *gateway node*, receives a message with the structure representing a query it assigns a unique random ID to the query. This ID uniquely identifies the query and is used for all network communication related to the query. Next, the gateway node uses the static attributes from the `WHERE` and the `JOIN` clauses to determine which nodes participate in the query by looking up the information in the DSAT. Once the gateway node retrieves the list of nodes participating in the query (i.e. the source nodes) it uses the dynamic attributes from the `WHERE` clause to retrieve selectivities from the source nodes. After selectivities are received the gateway node starts an algorithm for finding the processing node. The algorithm is described in the following sections. Once the algorithm converges and the processing node is found, the processing node sends a message all the source nodes for given query. The message contains the query ID and the ID of the processing node. Each source node uses the dynamic attribute, received previously to compute the selectivity, to filter out the sensed values which do not pass the dynamic condition. All other sensed values are sent to the processing node. After receiving all values from given epoch, the processing node performs the calculation specified in the query and if the condition is met, the result is sent to the gateway node which relays the result to the user.

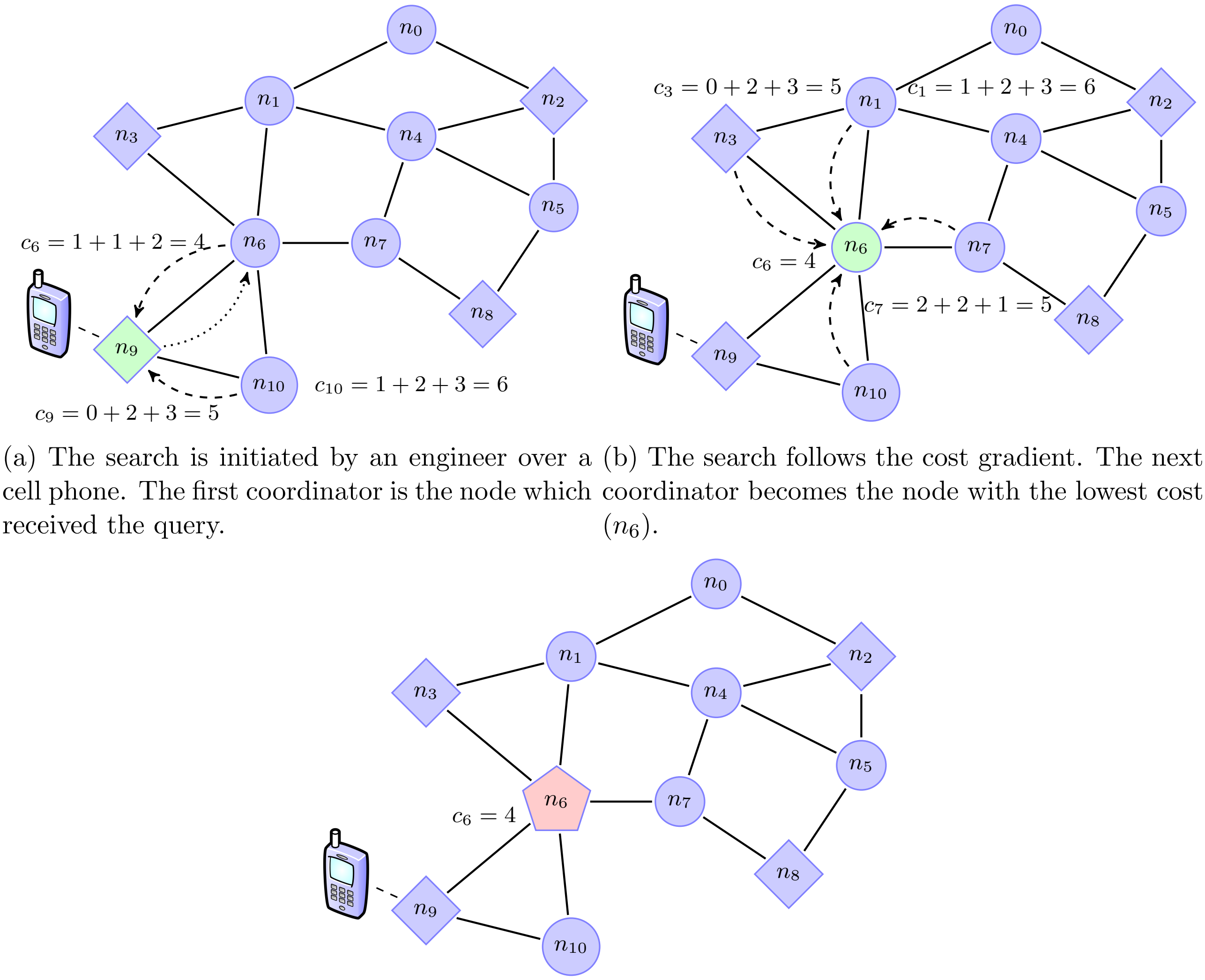
Algorithm 9 Processing Node Discovery for Homogeneous Networks

Preamble: **on** receiving a message of type Query Assignment **do** execute ReceiveQueryAssignment
The initial query assignment is send by a user via cell phone to any node

```
 1: procedure ReceiveQueryAssignment(packet, senderId)
 2:     query ← packet.query
 3:     local_cost ← compute the cost for query
 4:     request a cost from every neighbour
 5:     if local_cost is the lowest or this node was a coordinator before then
 6:         declare this node to be the processing node
 7:         inform all nodes participating in the query about the processing node
 8:     else
 9:         neighbourId ← from the list of nodes with the lowest cost randomly choose one node and send an assignment to the node
10:         packet.query ← query
11:         Unicast(packet, neighbourId, ACK)
12:     end if
13: end procedure
```

6.4.2 Homogeneous Networks

The iterative algorithm described in Algorithm 9 decreases the cost in each iteration by following the cost gradient towards the node with the lowest *cost*. How the algorithm works is depicted in Figure 6.2. In this figure the source nodes are diamond shaped, the regular nodes are shown as circles, and the processing node has a shape of a polygon. The algorithm consists of rounds, each of which is led by one *coordinator*, in the figure shown in green colour. At the beginning the node which received the query from a user, the *initiator*, becomes the first coordinator (Figure 6.2a). The coordinator computes its cost (line 3) 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 6.2a the replies from nodes n_6 and n_{10} are shown with dashed arrows. After the coordinator receives a reply from every neighbour, it compares its cost with all the received costs (line 5). If there is a node with a lower cost, the coordinator sends it an *assignment* message (line 9) and the receiver becomes a coordinator for the next round. In Figure 6.2a 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



(a) The search is initiated by an engineer over a cell phone. The first coordinator is the node which 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 6.2: 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.

coordinator is chosen randomly.

If all coordinator's neighbours' cost is higher then the coordinator declares itself as the *processing node* (line 6). In Figure 6.2b node n_6 receives costs from all neighbours. Because all received cost are higher than 4, the node declares itself as the processing node, which is depicted in Figure 6.2c.

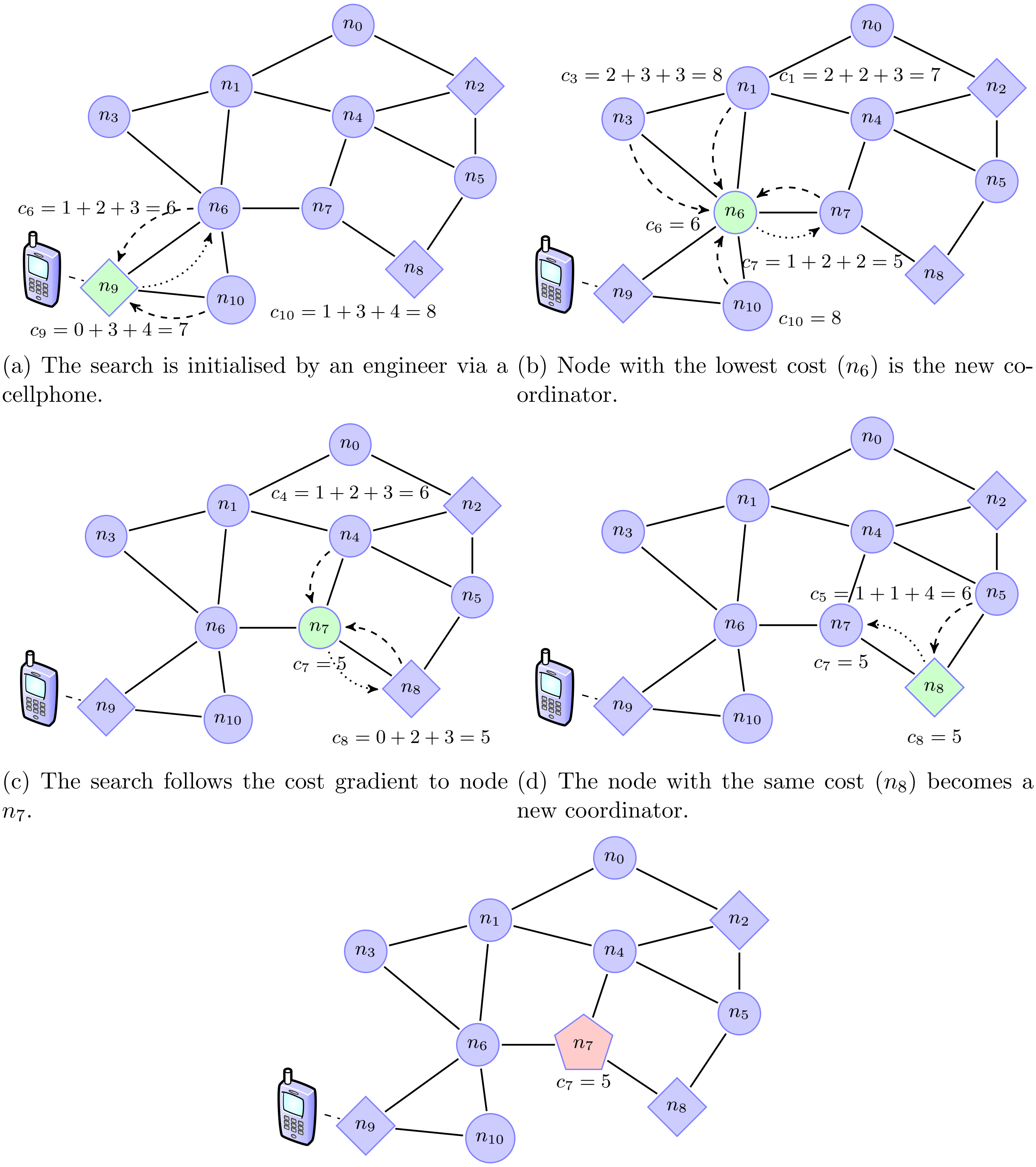
It may happen that the cost gradient is lost when the search hits an area of nodes with the same lowest cost. This situation is depicted in Figure 6.3. The search follows the cost gradient from node n_9 to node n_6 and then to node n_7. At this point, shown in Figure 6.3c, the coordinator cannot find 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 it is depicted in Figure 6.3d. Upon receiving an assignment the node checks whether it has already been a coordinator for the given query before (line 5). 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 6) in order to avoid loops in the search. This process can be seen in Figure 6.3e.

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 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 has 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 about 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 number of messages exchanged during the discovery phase mainly depends on the the number of coordinator's neighbours as the coordinator requires a reply from every neighbour. This number can be significantly decreased by snooping on neighbours as they reply back to the coordinator. Each node stores these replies and in the case it becomes a coordinator for the next round the node requests costs only from neighbours for which it is missing the cost. In Figure 6.3b node n_6 may request the cost only from nodes n_1, n_3, and n_7. Node n_6 has



(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 6.3: Processing Node Discovery 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.

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 6.3a).

6.4.3 Heterogeneous Networks

In case of heterogeneous networks we present three different algorithms: *i*) *query* algorithm, *ii*) *traverse*, and *iii*) *mixed* algorithm. We assume that a network is heterogeneous if some of the nodes have different computational and/or memory capabilities, i.e. some of the nodes are capable of processing data from more data streams. We assume that these attributes (CPU/memory) are static and can be stored in the DSAT. We also assume there is a function $f(src, sel) \rightarrow (cpu, mem)$, where (src, sel) is a list of sources and their respective selectivities and (cpu, mem) is a tuple specifying minimum CPU and memory requirements. Then, any node can retrieve a list of all possible processing nodes by running the query:

`SELECT node_id FROM dsat WHERE CPU >=` *cpu* `AND MEMORY >=` *mem*.

After a list of all *possible processing nodes* is retrieved, i.e. the list of all nodes capable of processing of all data streams of the query, the initialising node can start one of the three processing node discovery algorithms. These algorithms are, similarly to the algorithm for homogeneous networks, iterative, i.e. in every round one node is in charge and coordinates the search. The objective of each round is to find a node with a lower cost than the currently discovered one. If such node is not found in the current round, the message is *bounced back* to the coordinator. These *bounces* inform the coordinator that the search may not be going in the right direction and that the optimal node might have already been found. Additionally, they could be used to decrease the search space, hence speeding up the discovery process and lower the number of messages required to find the node.

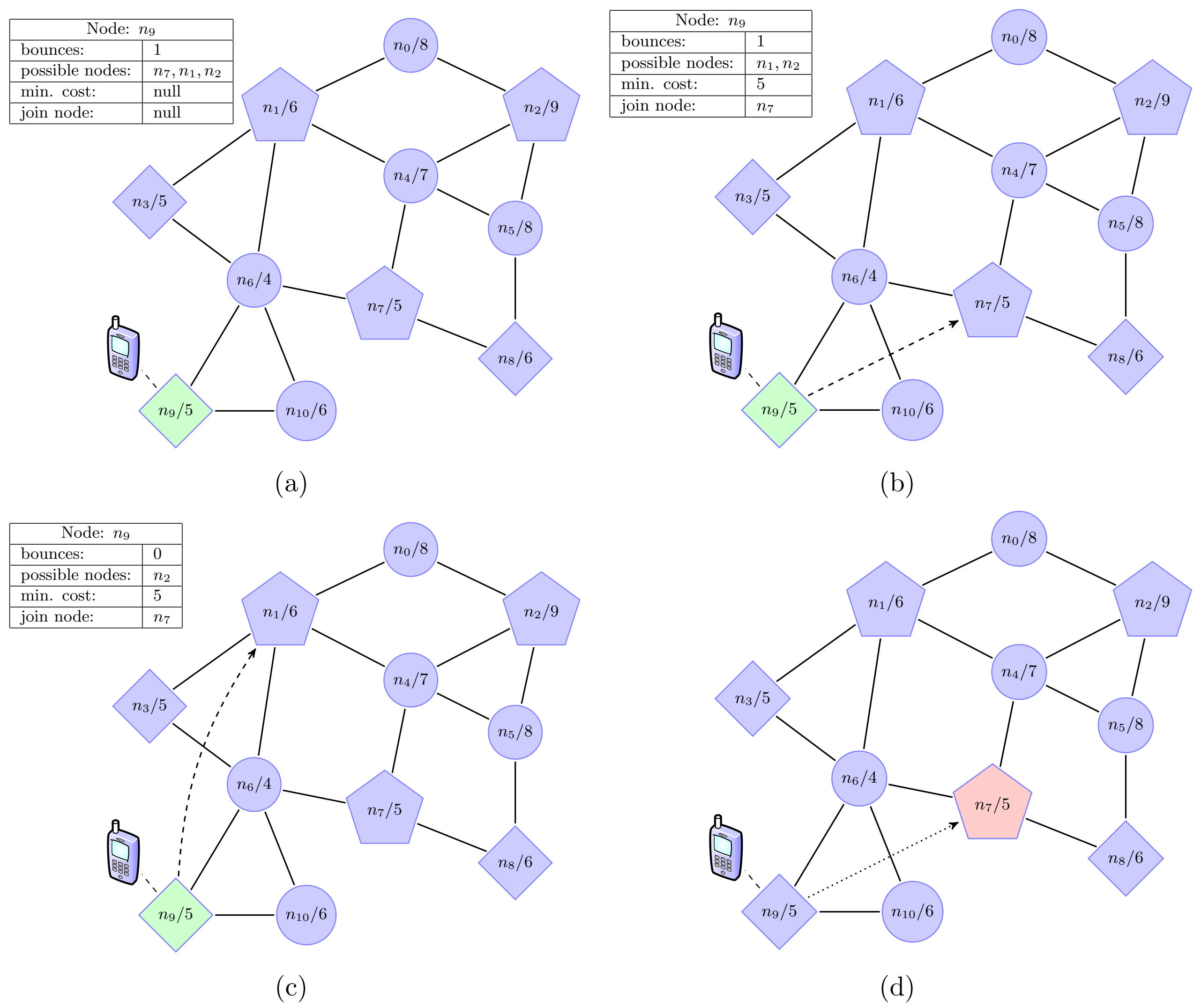
All three algorithms decrease the *bounce* variable whenever the assignment message has been bounced back. The algorithm stops if at least one of the following conditions is met: *i*) all possible processing nodes have been visited or *ii*) the number of bounces reaches 0. If the initial value of $bounces >= |possible\_processing\_nodes|$ then the *bounces* variable has no influence on the search algorithm.

Algorithm 10 Processing Node Discovery for Heterogeneous Networks - Query Algorithm

```
Preamble: on on receiving a message of type Query do execute ReceiveQuery
    query - a structure representing a query received from a user communicating with the node
    via cell phone

procedure ReceiveQuery(query)
    retrieve all possible processing nodes for query
    repeat
        retrieve the cost from the closest processing node
        if the cost < minimal_cost then
            store the processing node
        else
            $bounces \leftarrow bounces - 1$
        end if
    until $bounces = 0$ or all processing nodes have been requested
    $nodeId \leftarrow$ the node with the lowest cost node
    $packet.query \leftarrow query$
    SendForwardedMsg(packet, nodeId)  ▷ Send the query to the node with the lowest
    cost. The node will become the processing node.
end procedure
```

The main difference between algorithm for homogeneous and heterogeneous networks is that while in the case of homogeneous networks communication occurs only among neighbours, in the case of heterogeneous networks the possible processing nodes are arbitrary number of hops away. Therefore, instead of using broadcast, a reliable multi-hop forwarding algorithm, described in Section 3.4, is used for communication.

In all of the proposed algorithms for heterogeneous networks we investigate a scenario where a forwarding node is allowed to inspect the packet and, if a condition is met, act on behalf of the destination node by bouncing the message back to the sender. By allowing a message to be bounced back before it reaches the destination node it is possible to further reduce the search space and speed up the discovery process. However, if the message is bounced back under false assumptions, it may lead to discovering sub-optimal processing node. While evaluating our algorithms we investigate the influence of the bouncing conditions on the optimality of the discovered processing node.



Node: n_9	
bounces:	1
possible nodes:	n_7, n_1, n_2
min. cost:	null
join node:	null

Node: n_9	
bounces:	1
possible nodes:	n_1, n_2
min. cost:	5
join node:	n_7

Node: n_9	
bounces:	0
possible nodes:	n_2
min. cost:	5
join node:	n_7

(a) (b) (c) (d)

Figure 6.4: Query Algorithm for Heterogeneous WSNs.

Query Algorithm

The *Query* algorithm shown is in Algorithm 10. Please note, that the word “query” refers to the name of the algorithm and it does not describe overall overview of how a query is processed. The algorithm is based on a principle of the coordinating node requesting costs from the list of possible processing nodes until *bounces* = 0 or the list of possible processing nodes is empty. This process is depicted in Figure 6.4. In the figure the nodes are marked in the form of “ID/*cost*”, where the cost is the sum of distances to all the source nodes. The source nodes are diamond shaped (n_3, n_8 and n_9) and possible processing nodes are of the shape of a polygon ($n_1, n_2,$ and n_7). The coordinator is coloured in green and the processing node in red.

Algorithm 11 Processing Node Discovery for Heterogeneous Networks - Traverse Algorithm - Part 1

```
Preamble: on receiving a query from a user do execute ReceiveQuery procedure
    $query$ - a structure representing a query submitted by a user to any node

1: procedure ReceiveQuery($query$)
2:     $packet.list \leftarrow$ retrieve the list of possible processing nodes
3:     $packet.cost \leftarrow null$
4:     $packet.bounces \leftarrow$ initial value
5:     $nodeId \leftarrow$ choose the closest possible processing node from the $packet.list$
6:     SendForwardedMsg($packet, nodeId$)    ▷ Send a message of type Assignment to the
   closest node from the $packet.list$
7: end procedure
```

The process begins with the coordinator requesting the list of possible processing nodes and ordering them in ascending order according to their distance from the coordinator. The *bounces* variable is set to a predefined value, in this case $bounces = 1$ (Figure 6.4a). A request to the first node from the possible processing nodes list is sent. The request contains the list of source nodes and their selectivities only. In Figure 6.4b the request, depicted as a dashed line, is sent to node n_7, which replies with its cost ($c_7 = 5$). The coordinator removes the node from the list and saves the cost if it is lower than the one discovered so far. Otherwise, *bounces* variable is decreased. This process is repeated until either the possible processing nodes list is empty or the $bounces = 0$. In Figure 6.4c, after the coordinator receives the cost $c_1 = 6$ from node n_1, which is higher than the cost of node n_7 the *bounce* variable is decreased. Because $bounce = 0$ node n_2 is not requested for its cost but node n_7 is chosen as the processing node (line 11). In Figure 6.4d is the assignment message depicted as a dotted line.

Traverse Algorithm

The *Traverse* algorithm described in Algorithms 11, 12, and 13 is based on traversing the possible processing nodes from one to another. The process is depicted in Figure 6.5 and uses the same notation as the *Query* algorithm. The initiating node becomes the first coordinator and retrieves the list of all possible processing nodes from the DSAT. The coordinator then sends an assignment message to the closest possible processing node (lines 2–6). The assignment contains the list of source nodes with their selectivities, the list of possible processing nodes

Algorithm 12 Processing Node Discovery for Heterogeneous Networks - Traverse Algorithm - Part 2

Preamble: **on** receiving a message of type Assignment (sent on line 6, 24, or 41) **do** execute ReceiveAssignment procedure
packet - a packet which contains a structure representing the *query*, minimal *cost*, *list* of possible processing nodes, and *bounces* left
senderId - the node which sent the message of type Assignment

```
 8: procedure ReceiveAssignment(packet, senderId)
 9:     query ← packet.query
10:     minimal_cost ← packet.cost
11:     list ← packet.list
12:     cost ← compute cost for the query
13:     remove this node from the list of possible processing nodes
14:     if the cost > minimal_cost then
15:         send a message of type Reply to senderId
16:     else if the list of possible processing nodes is empty then
17:         declare this node to be the processing node
18:         inform all nodes participating in the query about the processing node
19:     else
20:         nodeId ← choose the closest possible processing node from the list
21:         packet.query ← query
22:         packet.list ← list
23:         packet.cost ← cost
24:         SendForwardedMsg(packet, nodeId)    ▷ Send a message of type Assignment to the closest node from the list
25:     end if
26: end procedure
```

that have not been visited yet, the minimal cost discovered so far, and the *bounces* variable. Sending the first assignment message could be seen in Figure 6.5a where node n_9 sends an assignment to node n_1.

Upon receiving an assignment the node removes itself from the list of possible processing nodes and computes the *cost* for given query. If $cost <= minimal\_cost$ the node becomes a new coordinator and sends an assignment to the closest node from the possible processing node. If the list is empty, it means that all possible processing nodes have been visited, therefore the node declares itself as the processing node (line 17). Sending an assignment to node n_7 by node n_1 is depicted in Figure 6.5b. Similarly, in Figure 6.5c, node n_7 sends an assignment to node n_2.

However, if $cost > minimal\_cost$ the node bounces the assignment back to the previous node,

Algorithm 13 Processing Node Discovery for Heterogeneous Networks - Traverse Algorithm - Part 3

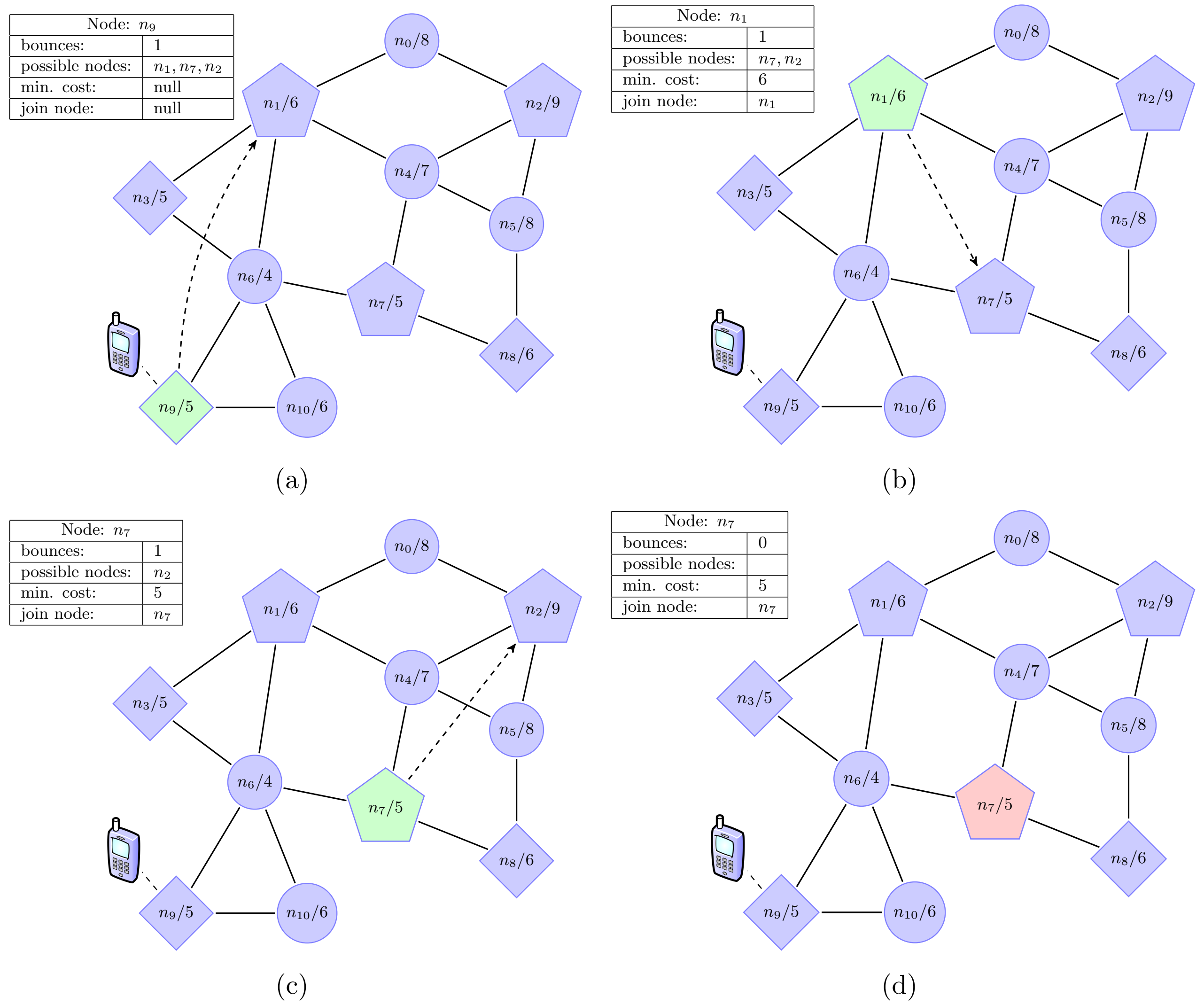
Preamble: on receiving a message of type Reply (sent on line 15) **do** execute procedure ReceiveReply
packet - a packet which contains a structure representing the *query*, minimal *cost*, *list* of possible processing nodes, and *bounces* left
senderId - the node which sent the message of type Reply

```
27: procedure ReceiveReply($packet, senderId$)
28:     $query \leftarrow packet.query$
29:     $bounces \leftarrow packet.bounces$
30:     $list \leftarrow packet.list$
31:     $bounces \leftarrow bounces - 1$
32:     if $bounces = 0$ or the $list$ of possible processing nodes is empty then
33:         declare this node to be the processing node
34:         inform all nodes participating in the query about the processing node
35:         return
36:     end if
37:     $packet.query \leftarrow query$
38:     $packet.list \leftarrow list$
39:     $packet.bounces \leftarrow bounces$
40:     $nodeId \leftarrow$ choose the closest possible processing node from the $list$
41:     SendForwardedMsg($packet, nodeId$)    ▷ Send a message of type Assignment to the
    closest node from the $list$
42: end procedure
```

i.e. the node from which it received the assignment (line 15). In Figure 6.5c node n_2 bounces the assignment back to node n_7 as its cost $c_2 = 9$ is larger than $c_7 = 5$. Upon receiving the reply the node decreases the *bounce* variable (line 31). If $bounce = 0$ or the list of possible processing nodes is empty, the node declares itself the processing node (line 33). If the list is not empty, it continues in the search and sends an assignment to the closest node from the list (line 41). In Figure 6.5d both of these conditions were met, therefore node n_7 declares itself as the processing node.

Mixed Algorithm

The *Mixed* algorithm combines the algorithm for homogeneous networks and the *Query* algorithm. How the algorithm operates is depicted in Figure 6.6. The figure uses the same notation as in the case of the *Query* algorithm. First, the algorithm for homogeneous networks finds



Node: n_9	
bounces:	1
possible nodes:	n_1, n_7, n_2
min. cost:	null
join node:	null

(a)

Node: n_1	
bounces:	1
possible nodes:	n_7, n_2
min. cost:	6
join node:	n_1

(b)

Node: n_7	
bounces:	1
possible nodes:	n_2
min. cost:	5
join node:	n_7

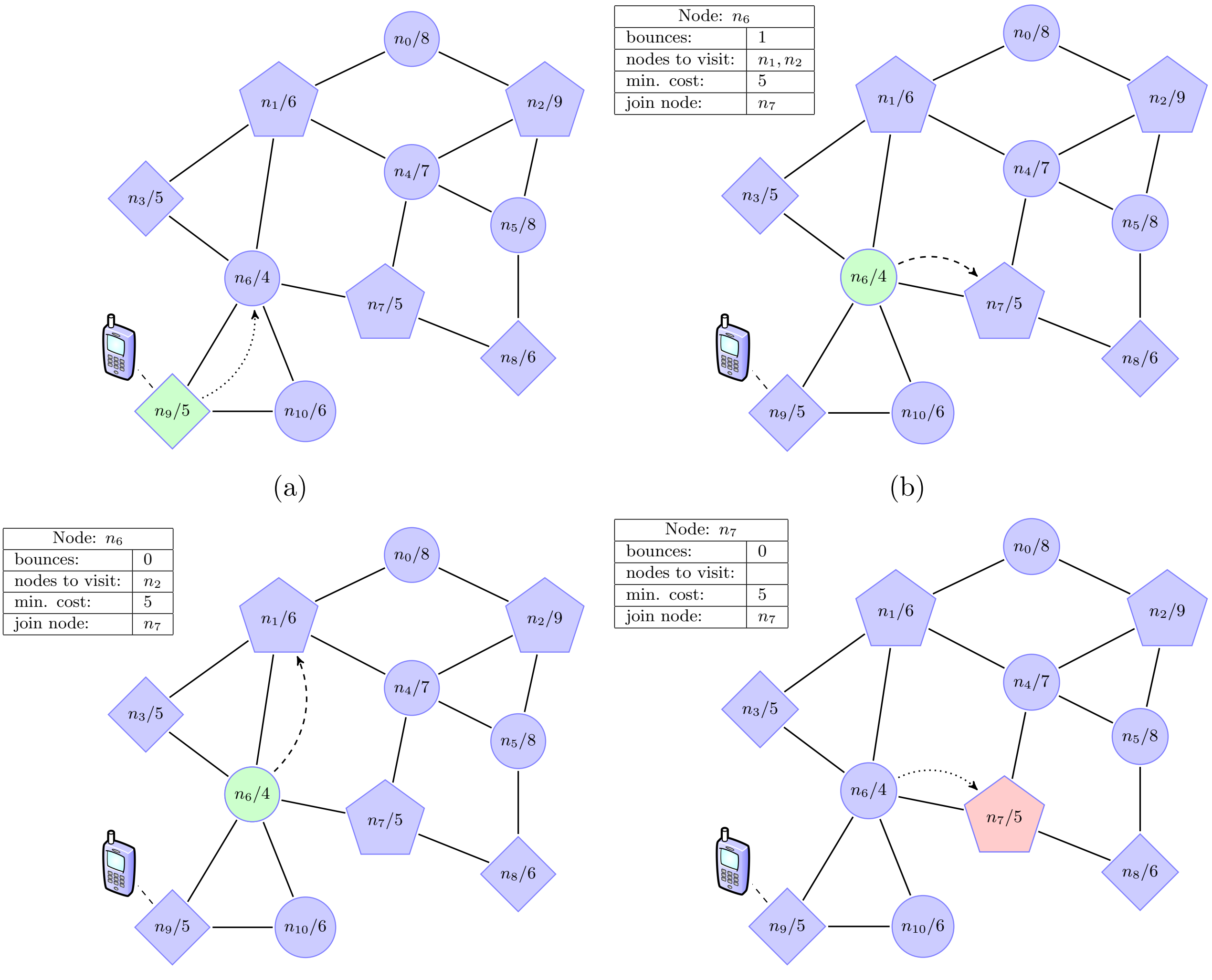
(c)

Node: n_7	
bounces:	0
possible nodes:	
min. cost:	5
join node:	n_7

(d)

Figure 6.5: Traverse Algorithm for Heterogeneous WSNs.

the optimal processing node amongst *all* nodes. In Figure 6.6a is this process shown with a dotted arrow. More in-depth description could be find in Figure 6.2. Next, this node is used as the starting point for the *Query* algorithm with very strict bouncing criteria, i.e. only a very small fraction of possible processing nodes are requested for their cost. We assume that the optimal possible processing node is in a close proximity to the optimal processing node, but not necessary the closest one. In the example in Figure 6.6 $bounce = 1$ so the coordinator requests costs from node n_7 (Figure 6.6b) and n_2 (Figure 6.6c). At the end, the node with the lowest cost is chosen as the processing node and the assignment is sent to this node (Figure 6.6d).



(a)

Node: n_6	
bounces:	1
nodes to visit:	n_1, n_2
min. cost:	5
join node:	n_7

(b)

Node: n_6	
bounces:	0
nodes to visit:	n_2
min. cost:	5
join node:	n_7

(c)

Node: n_7	
bounces:	0
nodes to visit:	
min. cost:	5
join node:	n_7

(d)

Figure 6.6: Mixed Algorithm for Heterogeneous WSNs.

6.5 Query Tuple Buffering Optimisation

Each epoch every source node senses the required value and sends the *query tuple*, in the form of $\langle source\_id, epoch, value\rangle$ to the processing node using the reliable multi-hop forwarding algorithm described in Section 3.4. As the query tuples from two different sources are forwarded towards the processing node they may pass via a common node. This node can, instead of forwarding each query tuple separately, merge two or more query tuples into a single message in order to lower the network traffic. We refer to this node as the *merging node*.

Each node, that is forwarding a query tuple towards the processing node, keeps the list of source nodes producing the query tuples. Once a forwarding node registers it is forwarding query

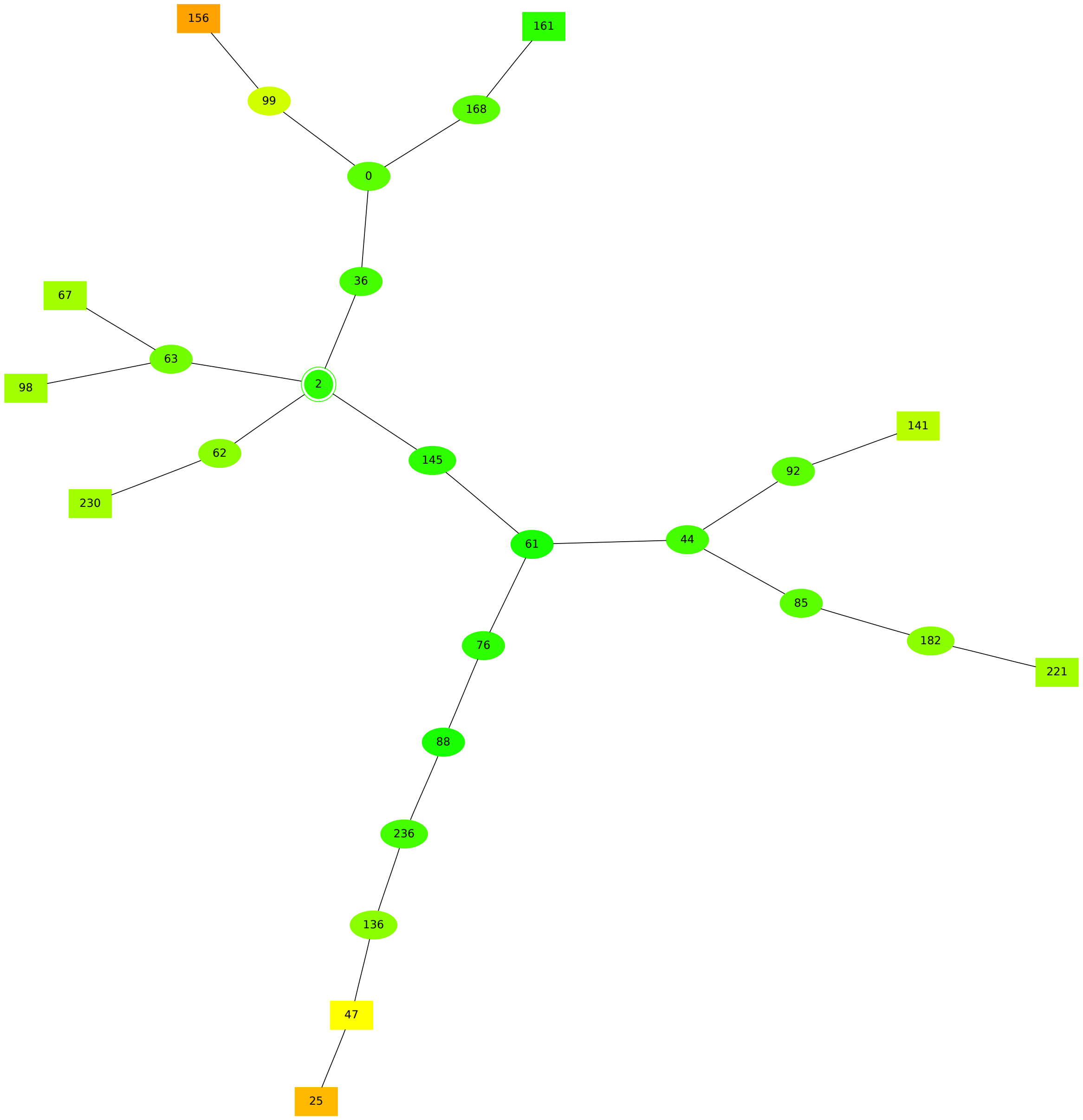


Figure 6.7: Query Tuple Buffering. The graph depicts routes travelled by the packets from the source nodes (shown as squares) to the processing node n_2 depicted as a circle. Nodes $n_0, n_{44}, n_{47}, n_{61}, n_{62}$, and n_{63} can perform QTB as packets from multiple source nodes pass through them. Nodes which do not forward any packets for given query are omitted from the graph.

tuples from more than one source node for the same query, it becomes a merging node and executes the Query Tuple Buffering algorithm described in Algorithm 14 and 15. The merging node receives query tuples from several source nodes and merges them into one message. The merged message is then forwarded towards the processing node.

Algorithm 14 Query Tuple Buffering - Part 1

```
Preamble: on receiving a packet of type Tuple do execute ReceiveTuple procedure
    packet - a packet which contains data tuple, query identifier, and the sourceId of the node
    which sensed and sent the tuple
    senderId - a neighbour from whom the packet was received
    maximum_delay - a global variable storing the maximum delay for given query
    source_list - a global variable storing a list of sources for given query
    buffer - a global variable storing tuples for given query in the same epoch

procedure ReceiveTuple(packet, senderId)
    tuple ← packet.tuple
    query ← packet.query
    sourceId ← packet.sourceId
    if this is the first tuple in this epoch then
        rcv_time ← current time
        timer ← set up a timer to expire at $rcv\_time + maximum\_delay$
    end if
    if there is only one source node in the source_list then
        forward tuple towards the processing node    ▷ Simple forwarding without merging.
        cancel timer
        return
    end if
    if tuples for current epoch have already been sent then
        forward tuple towards the processing node    ▷ Simple forwarding without merging.
        update maximum_delay    ▷ A tuple arrived later than usually therefore the maximum_delay is updated so in the next epoch the merging node waits longer for all tuples to arrive.
        return
    end if
    add tuple to the buffer
    enqueue acknowledgement packet for senderId
    if the number of tuples in buffer equals to the number of sources in the source_list then    ▷ tuples from all sources in this epoch have been received
        packet ← merge tuples in buffer
        nodeId ← processing node for the query
        SendForwardedMsg(packet, nodeId)    ▷ Send merged tuples to the processing node.
        ackPacket ← merge all pending acknowledgement packets
        Broadcast(ackPacket)
        cancel timer
    end if
end procedure
```

We have already described packets merging as a part of the routing algorithm in Section 3.4.1, however, it was limited only to packets which arrive at the same time. Obviously, this may not always be possible as the source nodes could be arbitrary number of hops away from the

Algorithm 15 Query Tuple Buffering - Part 2

```
Preamble: on expiration of the timer associated with the $buffer$ (set on line 7) do execute
    MaxDelayExpiration procedure
    $buffer$ - the buffer the timer is associated with

30: procedure MaxDelayExpiration($buffer$)
31:     $packet \leftarrow$ merge tuples in $buffer$
32:     $nodeId \leftarrow$ processing node for the $query$
33:     SendForwardedMsg($packet, nodeId$)  ▷ Send merged tuples to the processing node.
34:     $ackPacket \leftarrow$ merge all pending acknowledgement packets
35:     Broadcast($ackPacket$)
36: end procedure
```

merging node, hence the messages can arrive at various time. Figure 6.7 depicts routing paths in a network for one query. Source nodes are displayed at the edges of the graph while the processing node is node n_2 (shown as a circle). The difference in the arrival time of tuples to a merging node can be seen at the merging node n_{44} which 2 hops away from the source node n_{141} but 3 hops away from the source node n_{221}. In the case of the merging node n_{61} the difference is even larger, the closest node n_{141} is 3 hops away while the furthest away source node n_{25} is 6 hops away.

And it is not only the distance which has an influence on the arrival time of tuples to a merging node. There are many other reasons why the merging node may not receive a query tuple from a source node at some epoch, e.g. the source node may not produce the query tuple, the query tuple may be filtered by the dynamic condition of a query, the message may be lost, or a node on the path between the source and the merging node may fail.

Therefore, the merging node has to continuously monitor the *maximum delay*, Δd, i.e. the largest difference between arriving of the first and the last query tuple within the same sampling epoch. The maximum delay, obviously, mainly depends on the distance between the merging node and the source node furthest away, but it is also influenced by the link quality on given path. The merging node forwards all the query tuples in its buffer towards the processing node whenever it has received all query tuples for given sampling epoch (line 21) or Δd time later since the arrival of the first query tuple in given sampling epoch (line 30). If a query tuple arrives after the merging node has already sent all query tuples in the buffer, i.e. later than

Δd, the newly arrived query tuple is simply forwarded towards the processing node and the Δd is updated (line 14).

Recall, that in the reliable forwarding algorithm (described in Section 3.4) the sender considers the message acknowledged when the sender snoops on the receiver to forward the same message further or if the sender receives an acknowledgement packet from the receiver. However, because the merged message differs from the original messages received from the source nodes, the merging node has to act as the last hop in the reliable forwarding algorithm and send the acknowledgement packet to every node it received a query tuple from. As the merging node can also merge several acknowledgement packets into a single message it is more energy efficient to wait for all the query tuples to arrive or for Δd time before sending the acknowledgement message.

However, the forwarding algorithm waits only for *acknowledgement timeout* period before considering the message not to be delivered and re-sending the message. Therefore, the merging node has to send acknowledgement message before the timeout expires. If the timeout is very short the merging node has to send acknowledgement for every query tuple received separately. If the timeout is very long, it may have a negative effect on end-to-end delivery time. The reason is that the sending node cannot distinguish between the receiver waiting for other query tuples to arrive before acknowledging the message or not receiving the message at all. If the message was not received, the sender has to re-send the message.

If Query Tuple Buffering is used we double the normal acknowledgement timeout and experimentally show that it has very low impact on end-to-end delivery time while decreasing the overall traffic.

6.6 Evaluation

To evaluate our algorithms we used the same settings as in evaluation of other parts of the Dragon framework which was described in Chapter 2. We evaluate algorithms for homogeneous and heterogeneous networks separately. In both cases, the DSAT is split into 10 parts,

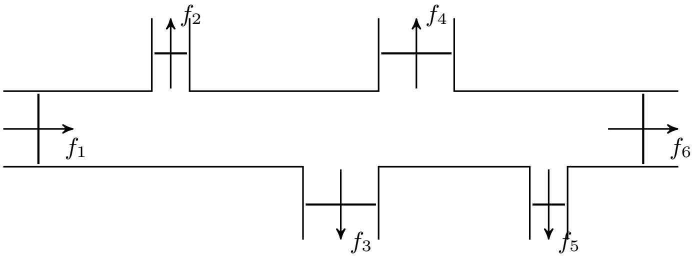


Figure 6.8: An example of a pipe segment with flow sensors. The arrow shows the direction of the flow of liquid in the pipe. Bars perpendicular to the arrows denote flow sensors.

i.e. every node is storing static attributes of 25 nodes.

6.6.1 Homogeneous Networks

In order to evaluate in-network data stream processing capabilities of Dragon platform we revisit the pipe leak scenario from the Introduction of this chapter. To refresh reader's memory, in Figure 6.8 we can see an example of a pipe segment. Each flow sensor is uniquely identified by ID. Additionally, each sensor is a member of two segments because each sensor ends one segment but also begins another one. Lets assume that the ID of a segment the node starts is saved in the `x` static attribute, while the one that the node ends is stored in the `y` static attribute. Then, the query which will retrieve all flow readings from one segment may look as follows:

```
SELECT S1.id, S1.flow, S2.id, S2.flow
FROM Sensors S1, Sensors S2
WHERE S1.x = S2.y
AND S1.flow > 0
EVERY 50 SECONDS.
```

If given query is submitted to node n_1 it will identify nodes $n_2 \dots n_6$ as the sources for given query. The processing node will retrieve flow readings from nodes $n_1 \dots n_6$ every 60 seconds and pass them to the application for leak detection.

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

We evaluate two versions of Dragon (with and without Query Tuple Buffering (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, therefore 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 node 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 [PG06, CG05]. However, it has been shown that processing data on a random node in the network leads to higher network traffic than processing at-the-base [MJIG10].

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 [MJIG08, MJIG10]. The pair-wise join, as its name suggests, joins exactly two streams of values. In the scenario from Figure 6.8, using the pair-wise join, node n_1 finds five join nodes, each of which joins data stream produced by n_1 and one other node. The location of the join node depends on selectivity of the two source nodes and the selectivity of the join node. Pair-wise join is able to lower the network traffic only if the selectivity of the join node is low. Innet framework periodically compares the cost of in-network pair-wise processing with processing at the base and chooses the one with a lower cost.

It is important to note the 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 the final processing. For comparison we used pair-wise joining with three different selectivity of the pair-wise join

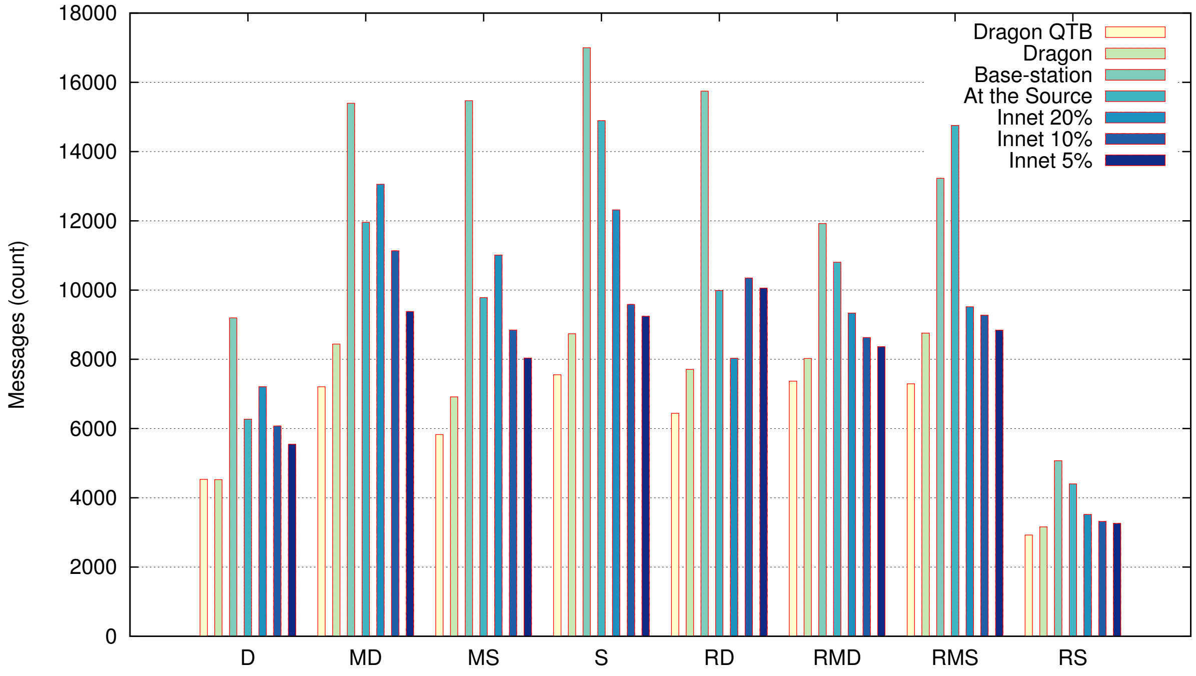


Figure 6.9: Comparison of the number of messages sent by various in-network processing algorithms.

nodes: $5\%, 10\%,$ and 20%. In case 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 of the processing node as:

$$\sigma = \sigma_p^{|S|-1} \tag{6.4}$$

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

During the evaluation, each source node sampled and sent a value every 50 seconds for the overall duration of 10000 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 the network of various topologies and densities can be seen in Figure 6.9. The comparison of Dragon with QTB optimisation is summarised in Table 6.1. Interestingly, processing data at the source outperformed processing at-the-base in all but random medium sparse topology. The savings ranged from $9 - 37\%$

Table 6.1: Number of messages comparison of Dragon QTB with various algorithms for in-network data stream processing.

Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
Dragon w/o QTB	0.2%	14.6%	15.7%	13.6%
Base-station	50.7%	53.2%	62.3%	55.6%
At the Source	27.7%	39.7%	40.4%	49.3%
Innet 20%	37.1%	44.8%	47.1%	38.6%
Innet 10%	25.3%	35.3%	34.1%	21.2%
Innet 5%	18.2%	23.2%	27.5%	18.3%
Topology	Random			
Dragon w/o QTB	16.5%	8.2%	16.7%	7.4%
Base-station	59.1%	38.2%	44.9%	42.3%
At the Source	35.5%	31.8%	50.6%	33.5%
Innet 20%	19.8%	21.0%	23.4%	16.8%
Innet 10%	37.8%	14.6%	21.4%	11.8%
Innet 5%	36.0%	11.9%	17.5%	10.3%

with an average of 23%. Despite the fact that these two techniques look similarly, 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 process them locally with its own data stream. Additionally, Chatzimilioudis *et al.* showed that under certain circumstances the optimal Fermat-Weber node is often one of the source nodes [CCGM13].

Comparing the pair-wise join with processing at-the-base shows that savings up to 49% could be achieved and on average ranges between 28% and 38%, depending on the selectivity of the join node. The lower the selectivity is, the bigger savings could be achieved. One of the reasons why pair-wise join can effectively lower the network traffic is the fact they can exploit multicast trees, because a value from one source node is delivered to several processing join nodes. A multicast tree can do 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, so 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.

However, using Dragon to find one central processing node at an optimal position (Fermat-Weber point) can further lower the traffic on average by 10% when compared with Innet with 5%

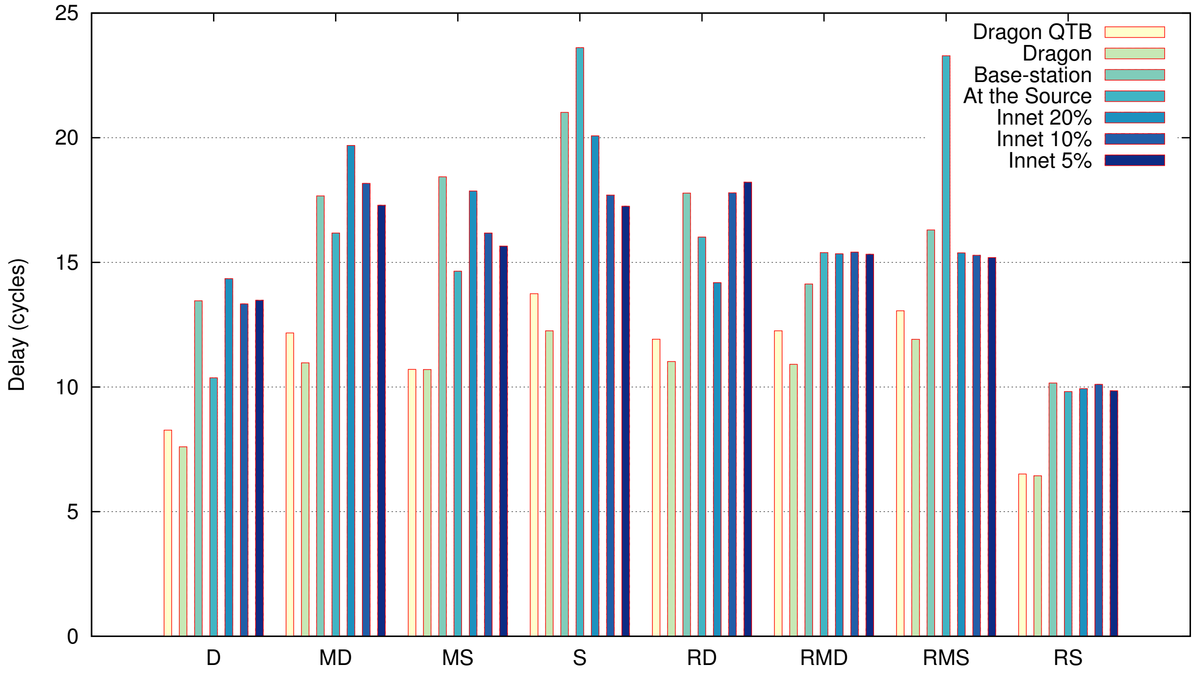


Figure 6.10: Comparison of the delay in tuple processing by various in-network processing algorithms.

Table 6.2: Delay in data processing comparison of Dragon QTB with various algorithms for in-network data stream processing.

Topology	Uniform			
Density	Dense	Med. Dense	Med. Sparse	Sparse
Dragon w/o QTB	-8.1%	-9.8%	-1.3%	-10.8%
Base-station	38.5%	31.1%	41.9%	34.6%
At the Source	20.2%	24.8%	26.9%	41.8%
Innet 20%	42.3%	38.2%	40.1%	31.5%
Innet 10%	38.0%	33.0%	33.8%	22.3%
Innet 5%	38.6%	29.6%	31.6%	20.3%
Topology	Random			
Dragon w/o QTB	-7.5%	-11.0%	-8.8%	-1.0%
Base-station	33.0%	13.3%	19.9%	35.9%
At the Source	25.6%	20.4%	43.9%	33.7%
Innet 20%	16.0%	20.2%	15.1%	34.5%
Innet 10%	33.0%	20.5%	14.6%	35.6%
Innet 5%	34.6%	20.1%	14.1%	33.9%

join node selectivity. Furthermore, if Query Tuple Buffering optimisation is used, the network traffic is lowered 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

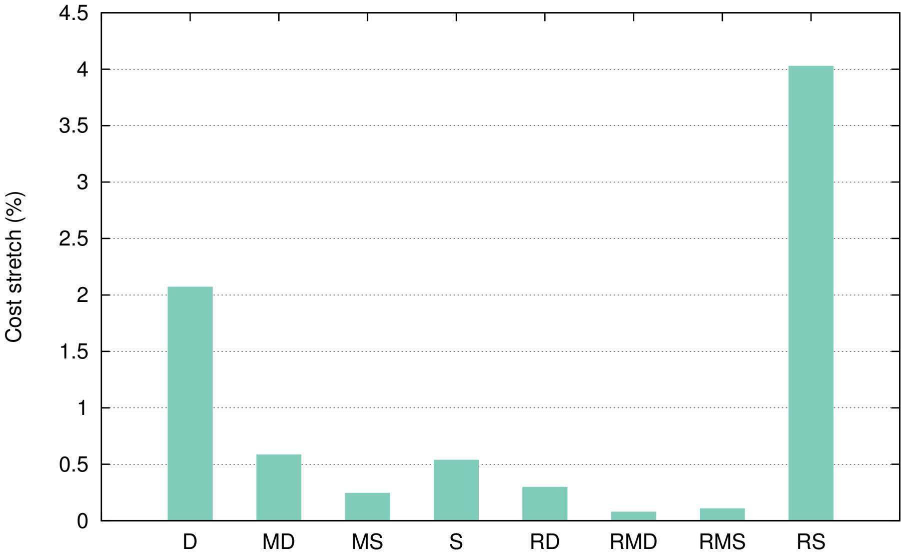
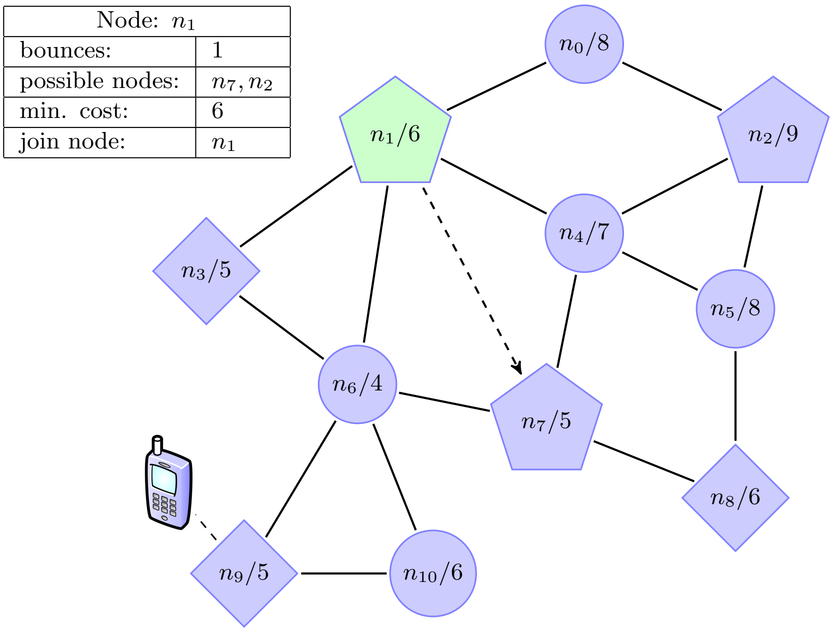


Figure 6.11: Percentage increase of cost of the discovered processing node vs. the optimal processing node.

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 actuation 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 6.10 and summarising Table 6.2 it can be clearly seen that Dragon 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%. This increase is caused by increasing the acknowledgement timeout period in the forwarding algorithm. We leave the decision which version of Dragon algorithm to use on the developer, depending on what is more important for the implemented application - whether lowering the network traffic or decreasing the network delay.

Last we evaluate the cost stretch, i.e. percentage increase in the cost of the discovered processing node vs. the optimal processing node. In Figure 6.11 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%.



Node: n_1	
bounces:	1
possible nodes:	n_7, n_2
min. cost:	6
join node:	n_1

Figure 6.12: Influence of bouncing on the Processing Node Discovery algorithms for heterogeneous networks.

6.6.2 Heterogeneous Networks

In the evaluation of algorithms for heterogeneous networks we focus on three metrics: *i*) cost stretch, i.e. percentage increase in the cost of the discovered processing node vs. the optimal processing node, *ii*) the number of messages required to discover the processing node, and *iii*) the time it takes to find the processing node. To the best of our knowledge, there is no other framework supporting in-network processing for heterogeneous networks, therefore we compare our *query* and *traverse* algorithms with the simplest solution - processing at-the-base. We assume that the base-station is the most powerful node, capable of processing any number of data streams.

In our experiments we also study the influence of *bouncing* the messages which can significantly narrow down the search space. We illustrate how a message could be bounced on Figure 6.12. Here, the coordinating node n_1 is sending an assignment message to node n_7. Cost of processing data streams at node n_1 is $c_1 = 6$ while at node n_7 it is $c_7 = 5$.

We study cases when only the destination node can bounce the message. In our example it means that only node n_7 is allowed to bounce the message back to node n_1. We also investigate cases, when any forwarding node on the path can act on behalf of the destination node. In our example it means that, depending on the path the message is taking, either node n_4 or n_6 can act on behalf of node n_7.

In our evaluation we study the influence of the *bounces* variable on the search quality. We vary the initial value of the variable and we set it to $\textit{bounces} = \{100\%, 50\%, 25\%\}$ of the number of the possible processing nodes. In practise it means that at least $100\%, 50\%$, or 25% of the possible processing nodes are queried for their cost.

Additionally, we investigate how the *cost threshold* which influences when a bounce by a forwarding node is triggered.

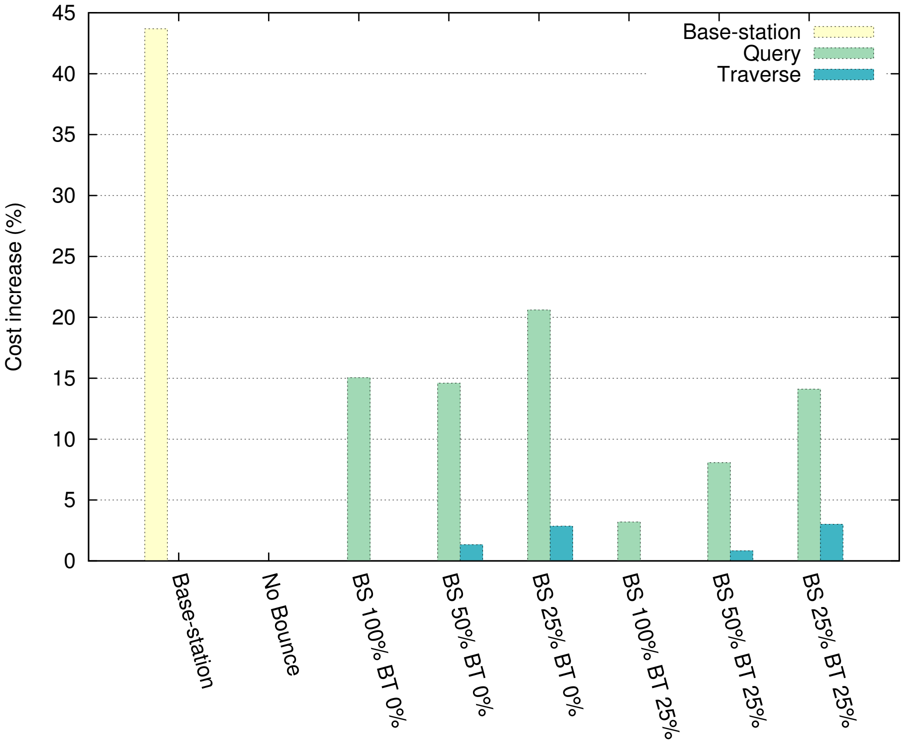
Bouncing by a forwarding node occurs when the cost of processing at the forwarding node is higher than the cost threshold c_T:

$$c_T = \text{bounce threshold} \times \text{minimal cost} \tag{6.5}$$

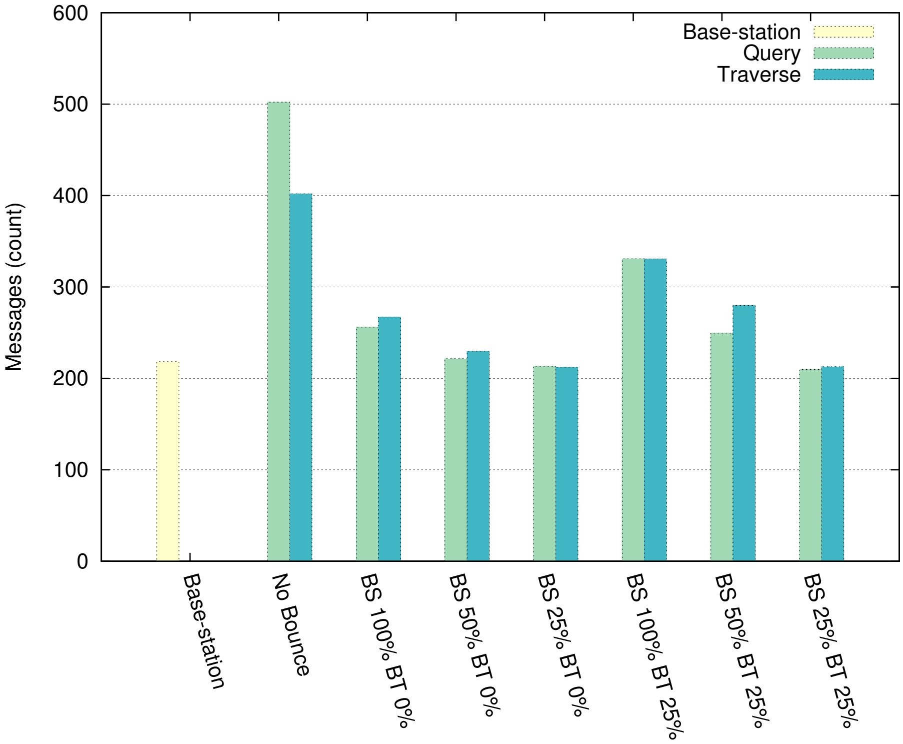
In our evaluation we use two values as the $\textit{bounce threshold} \in \{1, 1.25\}$, i.e. if the cost of the forwarding node is either higher than the minimal cost or more than 25% higher than the minimal cost. If we look back to our example in Figure 6.12 if the assignment message is sent via node n_6, it will always reach node n_7. However, if the message is sent via node n_4 and the $\textit{bounce threshold} = 1$ then the forwarding node n_4 will bounce the message back to node n_1, because $c_4 = 7$ which is more than $c_1 =$. Therefore, the optimal processing node n_7 will not be discovered. On the other hand, if the $\textit{bounce threshold} = 1.25$ then the message is not bounced by node n_4 and is forwarded to node n_7.

In our experiments we mark the result as “No Bounce” if bouncing occurs only at the *destination* node and the number of bounces is not limited. In other cases we allow any forwarding node to intercept the message and bounce it back. We use abbreviation BS for *bounces* variable and “BT” for *bounce threshold.*

To evaluate the algorithms for heterogeneous networks we use the same query as for the homo-



(a) Cost stretch



(b) Messages

Figure 6.13: Comparison of Query and Traverse algorithms with processing at the base-station. The comparison is for Query 1 which leads to selection of a smaller number of possible processing node. "BS" stands for "Bounce Size" and "BT" stands for "Bounce Threshold".

geneous network to retrieve the list of source nodes and their selectivities. After retrieving the list the initialising node issues the following query:

```
SELECT id FROM dsat WHERE y > 80.
```

This simulates retrieving the list of possible processing nodes from the DSAT. The first case, marked as *Query Q1* (Figure 6.13), results in finding $14 - 30$ possible join nodes (with an average of 22) and $2 - 12$ source nodes (with an average of 6). The number of nodes capable of processing the data streams is on average less than 9% of all the nodes in the network. The fact that only a small fraction of nodes are able to process the data streams is important for the traverse algorithm as it requires a list of nodes which need to be visited to be sent along with the message. In the case where the message has to be fragmented into many parts, the overall traffic will significantly increase. The advantage of the query algorithm is that it does not require the list of join nodes to be included in the discovery message as the initiating node orchestrates the search and only this node needs to keep the list of potential processing nodes in memory.

From the Figure 6.13a it can be clearly seen that whenever bouncing is used the traverse algorithm outperforms the query algorithm in terms of cost stretch, while both significantly outperform the processing at-the-base algorithm. The average difference between the cost of the optimal processing node and the processing node discovered by the traverse algorithm was constantly less than 3% while in case of the query algorithm it varied between $3 - 20\%$. It can be clearly seen that the more relaxed the bouncing criteria are, i.e. either more join nodes are queried (*bounce size* is higher) or the *bounce threshold* is higher, the query algorithm performs significantly better. On the other hand, the bouncing criteria do not have such a big impact on the traverse algorithm. Bouncing criteria influence the query algorithm because the search is orchestrated from a single node. If the cost of one of the initiator's neighbour is higher, e.g. due to the incorrect routing table, all possible join nodes for whom the given node is saved as the next hop, will be eliminated from the search and will not be visited.

It can also be seen that the best performance is achieved when bouncing is done only by the possible processing nodes, not the forwarding nodes. In this case both algorithms achieved 0%

cost increase, which was expected as all possible processing nodes were visited.

The second Figure 6.13b displays number of messages required to discover the processing node. As expected, most messages are sent when the messages are bounced only by the destination nodes. Additionally, the more relaxed the bouncing criteria are, the more messages are being sent. This behaviour is expected as with more relaxed bouncing criteria either more possible processing nodes are queried (if the bounce size is higher) or the discovery message travels further (if the bounce threshold is higher). It can also be seen that the difference between the query and traverse algorithm is negligible if forwarding nodes are allowed to bounce messages. The biggest difference of 20% between the query and the traverse algorithm is in the case when bouncing occurs only on destination nodes. This behaviour is also expected as the query algorithm initiates the search form a single node while in case of the traverse algorithm it traverses through all the join nodes while visiting the closest nodes first.

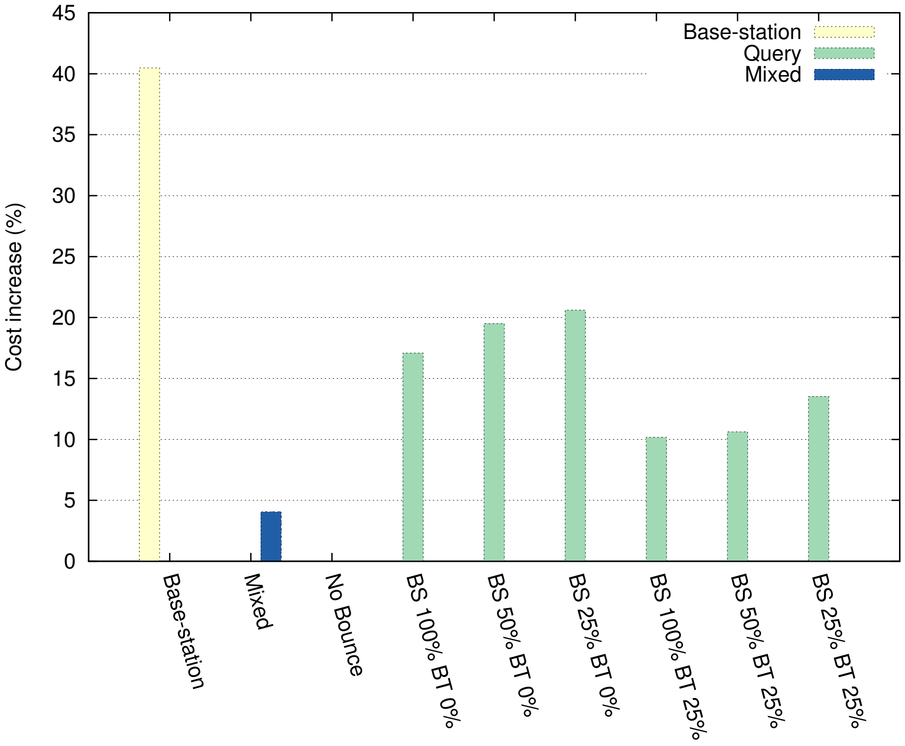
More interestingly, the cost of selecting the base-station as the processing node is not much cheaper (in terms of messages), especially if strict bouncing criteria are applied. This suggests that the cost of finding a processing node is dominated by finding the list of sources, retrieving their selectivities, notifying the processing node, and notifying the source nodes about the processing node.

We also compared the time it takes to find the processing node, however, we concluded that they are practically identical with those that compare the number of messages. Indeed, as the algorithms for heterogeneous networks rely on unicast rather than on broadcast the correlation between number of messages and time is very strong.

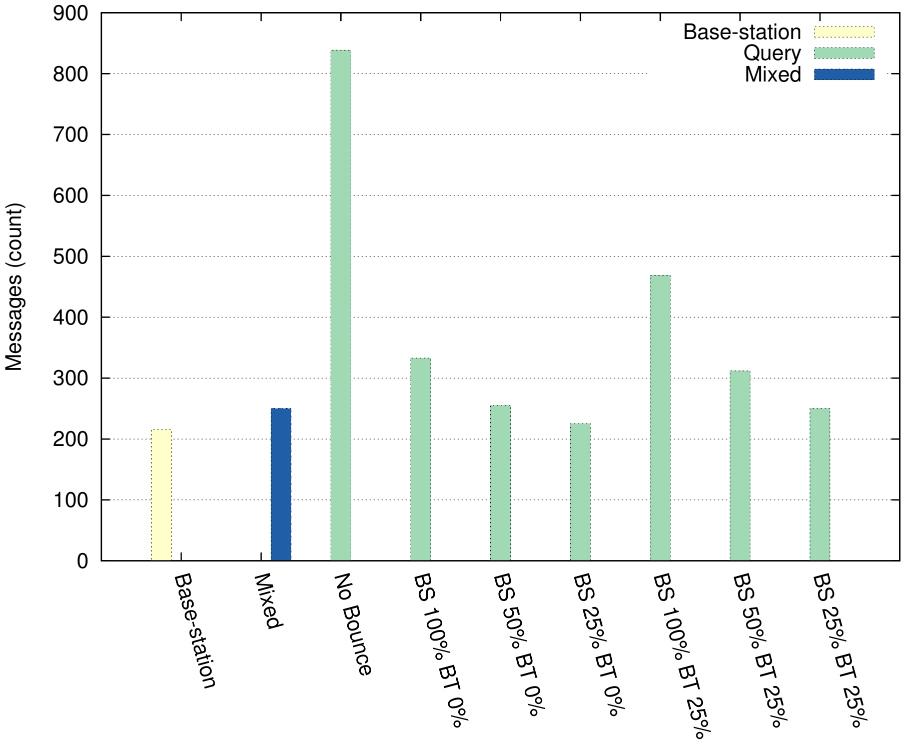
In the second case the restrictions on the possible processing node were more relaxed and the possible processing nodes were retrieved using following query:

```
SELECT id FROM dsat WHERE y > 60.
```

In this case the query Q2 (Figure 6.14) resulted in a much higher number of possible processing nodes - 55 on average - which rendered the traverse algorithm unusable due to the fact that the traverse algorithm has to pass the list of possible processing nodes from a node to a node.



(a) Cost stretch



(b) Messages

Figure 6.14: Comparison of Query and Traverse algorithms with processing at the base-station. The comparison is for Query 2 which leads to selection of a larger number of possible processing node. “BS” stands for “Bounce Size” and “BT” stands for “Bounce Threshold”.

This led to message fragmentation and a large increase in the number of messages sent. On the other side, the query algorithm does not require the list of possible processing nodes to be included in the processing node discovery message. This leads to a significant message savings as only one node orchestrates the search.

For the cases where the number of possible processing nodes is much higher than the packet size we evaluate the *mixed* algorithm. Once the optimal processing node is found using the algorithm for homogeneous networks we use the *query* algorithm with a very strict bouncing criteria - in our experiments we query only 15% of the closest possible processing nodes from the optimal processing node for their costs.

As it can be seen from the Figure 6.14a the cost increase for the *query* algorithm is very similar to the cost increase in the case of the first query in Figure 6.13a. On the other hand, the *mixed* algorithm performs very well with only 4% cost increase. Similarly to the first query, the processing at-the-base is on average 40% more expensive when compared with the optimal one.

If the number of messages is compared (Figure 6.14b), it can be seen that finding the processing node using the *mixed* algorithm is comparably cheap to processing at-the-base and cheaper than most of the *query* algorithm. We can see that the most messages are sent when bouncing is done only by processing nodes and significant number of messages can be saved by introducing bouncing by the forwarding nodes. The trend is similar to the first query. Similarly, the time is highly correlated with the number of messages; unsurprisingly.

The network is resilient to the node failure as it is handled by the routing layer described in Chapter 3. If a node on the path between the source node and the join node fails, a new path is automatically discovered. In the case of processing node failure, the processing node discovery algorithm is restarted and a new processing node is found.

6.7 Conclusion

In-network data processing has been shown to be a very challenging problem in WSNs. Choosing the right strategy can significantly decrease the number of messages transmitted within the network, hence increase its lifetime. However, current approaches assume traditional WSN where nodes are accessible only via a base-station which serves as a gateway between a user and the network. Unfortunately, this node also represents a single point of failure. Additionally, these approaches heavily rely on the base-station to perform part of the computation or to have a global knowledge about the network.

TinyDB [MFHH03] relies on a base-station to push the query down the routing tree towards the nodes participating in the query while exploiting the summaries stored at every node in the tree. However, TinyDB does not support joining of two data streams, only one data stream with a fixed size table stored at a node. Continuous Join Filtering (CJF) [SBB10] heavily rely on a base-station to first identify the nodes participating in the query and then to produce filters for each node. These filters are then pushed into the network. Subsequently, the base-station is responsible for performing the final computation on data which were not filtered out. Additionally, the base-station is also responsible for requesting the sensed data directly from the nodes if they were filtered out but may contribute to the final result due to an outlying reading from another node. In the case of SNEE [GBG$^+$11] the base-station is required to know the connectivity of the network prior to the query submission. The base-station stores metadata describing each node in the network as well as the network itself. When a query is submitted, SNEE uses these metadata to generate a Query Execution Plan. This plan is then used to generate a different binary for every node. These binaries are uploaded to the nodes which then execute one or more queries.

Innet [MJIG10] represents a framework where any node in the network can accept a query from a user and therefore is the closest to our approach. Innet uses several tree summaries to find the nodes participating in the query. However, Innet is capable to perform pair-wise joins only, where only two data streams are joined. The result of the pair-wise join is sent to a base-station where the final join is performed. Innet can significantly decrease the network traffic, however, only if the selectivity of the pair-wise join is very low. Otherwise, joining at the base-station leads to a lower network traffic.

In this chapter we presented several algorithms for discovering a processing node at or near the Fermat-Weber node, i.e. the node with the minimal weighted distance to every data source in the network. By choosing a single processing node in the network we avoid the need to have a base-station carrying out the final computation. This leads to a fully distributed design with no single point of failure and it allows every node in the network to accept and execute a query submitted by a user. However, choosing only one processing node for each query comes at the expenses of possible network congestion around the processing node and the higher computational requirements of the processing node.

The platform can operate in both homogeneous networks and heterogeneous networks, where only a subset of the nodes is capable of processing the data. The algorithm for homogeneous networks can find processing nodes whose cost is on average only 1% higher than the cost of the processing node with the lowest cost. Two algorithms presented for heterogeneous networks perform differently depending on the percentage of nodes capable of processing data streams. The *Traverse* algorithm performs better if the set of possible processing nodes is small, while the combination of algorithm for homogeneous networks and the *Query* algorithm performs better for a larger set of possible processing nodes. Depending on the query and heterogeneity of the network, algorithms can discover a node with no more than 4% higher cost than the node with the lowest cost. Performing in-network processing at the nodes discovered by the algorithms leads to a message reduction of up to 56% and decreases the processing delay by as much as 42%.

Chapter 7

Summary and Conclusion

Wireless sensor network are gradually penetrating our day-to-day life. Currently, they are used by researcher to observe and better understand the world around us, by companies to monitor their assets, or by citizen hacker communities to make their lives easier. WSNs can be a very flexible tool which could help us better understand underlying causes of a phenomenon, could lower the running costs of various equipment, and make the life easier. However, there are still many challenges that need to be addressed.

As the networks grow bigger in terms of the number of nodes deployed, it is getting more energy inefficient to retrieve the information we are interested in. As the wireless communication is usually one of the most energy hungry sub-system of a WSN node, the aim of every application is to lower the network communication to the lowest possible level. This cannot be achieved when every sensor reading is sent to a cloud where a user might eventually use it. Therefore, researchers investigate possibilities of how to push the computation into the WSN in such way, that the information a user is interested in is still delivered, but the wireless communication is minimised.

In order to do so, we must first enhance node's capabilities. Instead of periodically sensing and sending the sensed value via the same path to a base-station we need to equip the node with new abilities. First, we need to allow any node in the network to send data to any other node. This should be done with as low communication overhead as possible, i.e. the message should follow the shortest path between two nodes and there should be no need to discover this path prior to sending the message.

In Chapter 3 we presented and evaluated a new routing algorithm for WSNs based on routing tables. Each node learns a distance and the next hop neighbour to every other node in the network in a fast way with low communication overhead. The routing stretch achieved by Dragon algorithm is less than 0.1% which allows every node to communicate with any other node via optimal or near-optimal paths.

Once a node is capable of ad-hoc communication with any node in the network we should allow it to identify all other nodes in the network which fulfil given static requirements. Searching by static attributes is important in cases when a node has to identify all other nodes monitoring the same phenomenon, e.g. oil flow on the same pipe. Each node should be able to do so without flooding the whole network. Search should be fast and with a low communication overhead.

In Chapter 4 we presented a Distributed Static Attribute Table (DSAT) which stores information about all nodes in the network in a distributed way, similarly to how distributed databases split tables and store them on different computers. By distributing the table throughout the network, every node has to store only a small part of this table. Additionally, the table is distributed in such a way that it allows every node to search in whole DSAT by communicating with a close neighbourhood only.

If a node is able to identify a set of nodes with given static attributes and can communicate with them directly without a need of a central node, the whole wide range of possibilities open. Any node in the network can accept an ad-hoc query from any user and evaluate it.

In Chapter 5 we evaluated how the routing algorithm and the DSAT can be exploited to answer snapshot queries submitted by users. Users are allowed to retrieve current readings only from sensors they are interested in. Dragon supports snapshot query execution with a very low communication overhead and in timely manner.

Finally, if a network is capable of serving ad-hoc snapshot queries, it can serve continuous queries, too. Here the problem arises as which node should process the data streams arriving at predefined intervals. The problem of choosing the correct node is a very broad area of research and many solutions were proposed. Due to the broadness of this subject, many of the proposed solutions focus only on a small part of the problem and solve it for a specific criteria only.

In Chapter 6 we proposed and evaluated algorithm for finding such processing node in a distributed way, without the need for any centralised node orchestrating the search. We showed that processing data streams at the node discovered by our approach can significantly decrease the network traffic as well as the processing delay. Additionally, we investigated the problem of data stream processing in heterogeneous networks where some nodes have higher computational capabilities than other nodes.

7.1 Contributions

In this thesis we introduced Dragon framework with the following list features which could be considered as contributions to the WSN community.

1. We presented a new routing algorithm based on routing tables. Using a proactive approach and by exploiting broadcast capabilities of the wireless communication we can achieve routing stretch as low as less than 0.1%. Additionally, the routing algorithm is resilient to node failures and can adapt routing tables fast and with low communication overhead.

2. Distributing and storing static attributes throughout the network allows any node in the network to identify a set of nodes with given static attributes by communicating with close neighbourhood only. We propose an algorithm which allocates parts of the Distributed Static Attribute Table to nodes in such way that the average communication overhead is decreased.

3. We proposed an algorithm to disseminate static attributes of every node in the network to a set of nodes. We compared our dissemination protocol with other traditional dissemination protocols and showed that DRAGON can decrease the network traffic on average by 41%.

4. We proposed a framework for evaluation snapshot queries submitted by a user via any node in the network. The node can identify all the nodes that satisfy user's query and request current sensor readings from these nodes. We compared our approach with other state-of-the-art approaches and showed that DRAGON can decrease the network traffic on average by 71% and the processing delay by 48%.

5. We introduced a distributed algorithm for discovery of a processing node of continuous queries in WSNs. The algorithm follows the cost gradient to the node with the lowest cost. The cost of processing is computed as a sum of weighted distances from the processing node to all the source nodes. We compared executing query at the node discovered by DRAGON with other state-of-the-art algorithms and we showed that DRAGON can decrease the network traffic by $20 - 51\%$ depending on the algorithm the comparison is done with. Processing query at the node discovered by DRAGON also decreases the processing delay by $33 - 36\%$ depending on the algorithm DRAGON is compared to.

6. We also extended our work to the area of heterogeneous WSNs. We proposed three new distributed algorithms for processing node discovery. We assume that in heterogeneous networks only a sub-set of the nodes are able to process given query. Therefore the search has to be limited to those nodes only. We showed that our algorithms are capable of finding either optimal or near-optimal processing nodes.

7.2 Future Work

This work is a small contribution to the big area of in-network data stream processing. Now, that we have showed advantages of our approach, we can see several possible extensions of DRAGON's sub-systems.

In the routing sub-system we find the requirement to store every destination to be not very memory efficient. We would like to investigate the possibility to form groups of sensor nodes and treat them as a single entity. This approach should significantly reduce memory requirements at every node. However, group forming remains the biggest challenge. Our preliminary thought experiments suggest that grouping nodes by cliques might lead to good results. If any node knows the distance to a clique is d hops, then it can be sure that every node from that clique could be reached within at most $d + 1$ hops. Additionally, if any node fails, we can be sure that the clique will remain interconnected, as long as there are more than one node left in the clique.

Another improvement could be achieved in forwarding algorithm by allowing a pair of nodes to agree on the acknowledgement timeout period. Currently, we support only one global timeout, which does not suit nodes which merge several messages into one. By allowing a pair of nodes to agree on the acknowledgement timeout period we can achieve decrease in network traffic by merging messages together, while not increasing the end-to-end delivery time in other parts of the network.

We can also see limitations of our Distributed Static Attribute Table (DSAT) sub-system. Currently, we do not support dynamic scalability and the number of parts the DSAT is split into has to be chosen at deployment time. It cannot be changed without re-initialising the DSAT algorithm and subsequently running the Static Attribute Propagation algorithm, which fills the DSAT with data.

We also do not support updating of static attributes. We assume that once a node is deployed its static attributes will never change. We recognise, that in real world it might not always be true, therefore, we want to investigate the algorithms for updating the DSAT with new values.

Currently, the DSAT is horizontally partitioned. As the searches in the DSAT usually include only one column we would like to investigate the possibility of vertical partitioning of the DSAT. We believe that it could lead to significant savings in the preparation phase of the query execution, both snapshot and continuous.

The last improvement of Dragon we would like to implement is processing node migration in the case when selectivity of one or more source nodes changes. We believe that implementing this improvement will be straightforward as the position of the new processing node will usually be in the close proximity of the previous one.

Finally, we would like to deploy and evaluate Dragon on real nodes. We plan to do it as a part of the Information & Communication Technologies (ICT) project which we are part of.

Wireless sensor networks are slowly penetrating our life. In order to be able to exploit most of their capabilities we should allow them to do the work instead of just use them to mindlessly sense and report data. Here we have shown how a WSN can perform such computation in a fast and efficient way.

Bibliography

[ABB$^+$03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[AL06] Z. Abrams and Jie Liu. Greedy is good: On service tree placement for in-network stream processing. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 72–72, 2006.

[AML05] Daniel J. Abadi, Samuel Madden, and Wolfgang Lindner. Reed: Robust, efficient filtering and event detection in sensor networks. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 769–780. VLDB Endowment, 2005.

[BE02] David Braginsky and Deborah Estrin. Rumor routing algorthim for sensor networks. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, pages 22–31, New York, NY, USA, 2002. ACM.

[BHE00] N. Bulusu, J. Heidemann, and D. Estrin. Gps-less low-cost outdoor localization for very small devices. *Personal Communications, IEEE*, 7(5):28–34, Oct 2000.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BP00] P. Bahl and V.N. Padmanabhan. Radar: an in-building rf-based user location and tracking system. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 775–784 vol.2, 2000.

[CcC$^+$02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[CCGM13] Georgios Chatzimilioudis, Alfredo Cuzzocrea, Dimitrios Gunopulos, and Nikos Mamoulis. A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement. *Journal of Computer and System Sciences*, 79(3):349 – 368, 2013. Theoretical and Practical Aspects of Warehousing, Querying and Mining Sensor and Streaming Data.

[CD13] Zuhal Can and Murat Demirbas. A survey on in-network querying and tracking services for wireless sensor networks. *Ad Hoc Networks*, 11(1):596 – 610, 2013.

[CG05] Vishal Chowdhary and Himanshu Gupta. Communication-efficient implementation of join in sensor networks. In *In Proceedings of 10th International Conference on Database Systems for Advanced Applications*, pages 447–460, 2005.

[CN07] A. Coman and M.A. Nascimento. A distributed algorithm for joins in sensor networks. In *Scientific and Statistical Database Management, 2007. SSBDM '07. 19th International Conference on*, SSBDM '07, pages 27–27, 2007.

[CnDLR12] Eduardo CañEte, Manuel DíAz, Luis Llopis, and Bartolomé Rubio. Hero: A hierarchical, efficient and reliable routing protocol for wireless sensor and actor networks. *Comput. Commun.*, 35(11):1392–1409, June 2012.

[CNS07] A. Coman, M.A. Nascimento, and J. Sander. On join location in sensor networks. In *Mobile Data Management, 2007 International Conference on*, pages 190–197, May 2007.

[Croa] Crossbow. Micaz datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pd

[Crob] Crossbow. Telosb datasheet. http://www.willow.co.uk/TelosB_Datasheet.pdf.

[DCABM05] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. *Wirel. Netw.*, 11(4):419–434, July 2005.

[DF03] M. Demirbas and H. Ferhatosmanoglu. Peer-to-peer spatial queries in sensor networks. In *Peer-to-Peer Computing, 2003. (P2P 2003). Proceedings. Third International Conference on*, pages 32–39, Sept 2003.

[DGV04] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, Nov 2004.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DL07] M. Demirbas and Xuming Lu. Distributed quad-tree for spatial querying in wireless sensor networks. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 3325–3332, June 2007.

[DLV13] Simon Duquennoy, Olaf Landsiedel, and Thiemo Voigt. Let the tree bloom: Scalable opportunistic routing with orpl. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 2:1–2:14, New York, NY, USA, 2013. ACM.

[DpEG01] L. Doherty, K.S.J. pister, and L. El Ghaoui. Convex position estimation in wireless sensor networks. In *INFOCOM 2001. Twentieth Annual Joint Conference of the*

IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1655–1663 vol.3, 2001.

[ESA09] B Elbhiri, R Saadane, and D Aboutajdine. Stochastic distributed energy-e cient clustering (sdeec) for heterogeneous wireless sensor networks. 2009.

[ESEFA10] B. Elbhiri, R. Saadane, S. El Fkihi, and D. Aboutajdine. Developed distributed energy-efficient clustering (ddeec) for heterogeneous wireless sensor networks. In *I/V Communications and Mobile Network (ISVC), 2010 5th International Symposium on*, pages 1–4, Sept 2010.

[FGNW06] Stefan Funke, LeonidasJ. Guibas, An Nguyen, and Yusu Wang. Distance-sensitive information brokerage in sensor networks. In PhillipB. Gibbons, Tarek Abdelzaher, James Aspnes, and Ramesh Rao, editors, *Distributed Computing in Sensor Systems*, volume 4026 of *Lecture Notes in Computer Science*, pages 234–251. Springer Berlin Heidelberg, 2006.

[FSGM$^+$99] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. Technical Report 1999-67, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0121.

[GBG$^+$11] Ixent Galpin, ChristianY.A. Brenninkmeijer, AlasdairJ.G. Gray, Farhana Jabeen, AlvaroA.A. Fernandes, and NormanW. Paton. Snee: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1-2):31–85, 2011.

[GFJ$^+$09] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 1–14, New York, NY, USA, 2009. ACM.

[GRS$^+$03] Benjamin Greenstein, Sylvia Ratnasamy, Scott Shenker, Ramesh Govindan, and Deborah Estrin. Difs: a distributed index for features in sensor networks. *Ad Hoc Networks*, 1(23):333 – 349, 2003. Sensor Network Protocols and Applications.

[HCB00] W.R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*, pages 10 pp. vol.2–, Jan 2000.

[HL] Abderrahim BENI HSSANE and Moulay Lahcen. Improved and balanced leach for heterogeneous wireless sensor networks.

[IDC13] IDC. Worldwide internet of things (IoT) 2013-2020 forecast: Billions of things, trillions of dollars. *Doc 243661*, page 22, October 2013.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, pages 56–67, New York, NY, USA, 2000. ACM.

[IvS09] Konrad Iwanicki and Maarten van Steen. On hierarchical routing in wireless sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 133–144, Washington, DC, USA, 2009. IEEE Computer Society.

[JM96] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Tomasz Imielinski and HenryF. Korth, editors, *Mobile Computing*, volume 353 of *The Kluwer International Series in Engineering and Computer Science*, pages 153–181. Springer US, 1996.

[Kan13] Hyunchul Kang. In-network processing of joins in wireless sensor networks. *Sensors*, 13(3):3358–3393, 2013.

[KAP09] Dilip Kumar, Trilok C. Aseri, and R.B. Patel. Eehc: Energy efficient heterogeneous clustered scheme for wireless sensor networks. *Computer Communications*, 32(4):662 – 667, 2009.

[KAP11] Dilip Kumar, Trilok C Aseri, and RB Patel. Multi-hop communication routing (mcr) protocol for heterogeneous wireless sensor networks. *International Jour-*

nal of Information Technology, Communications and Convergence, 1(2):130–145, 2011.

[KBM15] Roman Kolcun, David Boyle, and Julie A McCann. Optimal processing node discovery algorithm for distributed computing in IoT. In *The 5th International Conference on the Internet of Things (IOT) 2015 (IoT 2015)*, Seoul, Korea, October 2015.

[KK00] Brad Karp and H. T. Kung. Gpsr: greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 243–254, New York, NY, USA, 2000. ACM.

[KLS14] Matthias Kovatsch, Martin Lanter, and Zach Shelby. Californium: Scalable cloud services for the internet of things with coap. In *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

[KM14] Roman Kolcun and Julie A McCann. Dragon: Data discovery and collection architecture for distributed IoT. In *Internet of Things 2014 - The 4th International Conference on the Internet of Things (IoT 2014)*, Cambridge, USA, Oct 2014.

[KWZZ03] Fabian Kuhn, Rogert Wattenhofer, Yan Zhang, and Aaron Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 63–72, New York, NY, USA, 2003. ACM.

[LCC08] Yong-Xuan Lai, Yi-Long Chen, and Hong Chen. Peja: Progressive energy-efficient join processing for sensor networks. *Journal of Computer Science and Technology*, 23(6):957–972, 2008.

[LG00] Sung-Ju Lee and M. Gerla. Aodv-br: backup routing in ad hoc networks. In *Wireless Communications and Networking Confernce, 2000. WCNC. 2000 IEEE*, volume 3, pages 1311–1316 vol.3, 2000.

[LHZ04] Xin Liu, Qingfeng Huang, and Ying Zhang. Combs, needles, haystacks: Balancing push and pull for discovery in large-scale sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 122–133, New York, NY, USA, 2004. ACM.

[LLG10] Yongxuan Lai, Ziyu Lin, and Xing Gao. Srja: Iceberg join processing in wireless sensor networks. In *Database Technology and Applications (DBTA), 2010 2nd International Workshop on*, pages 1–4, Nov 2010.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[LMP$^+$05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In Werner Weber, JanM. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[LR02] S. Lindsey and C.S. Raghavendra. Pegasis: Power-efficient gathering in sensor information systems. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 3, pages 3–1125–3–1130 vol.3, 2002.

[LRS01] Stehpanie Lindsay, C. S. Raghavendra, and Krishna M. Sivalingam. Data gathering in sensor networks using the energy delay metric. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium*, IPDPS '01, pages 188–, Washington, DC, USA, 2001. IEEE Computer Society.

[MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 491–502, New York, NY, USA, 2003. ACM.

[MGV$^+$13] Guilherme Maia, Daniel L. Guidoni, Aline C. Viana, Andre L.L. Aquino, Raquel A.F. Mini, and Antonio A.F. Loureiro. A distributed data storage protocol for heterogeneous wireless sensor networks with mobile sinks. *Ad Hoc Networks*, 11(5):1588 – 1602, 2013.

[MJIG08] Svilen R. Mihaylov, Marie Jacob, Zachary G. Ives, and Sudipto Guha. A substrate for in-network sensor data integration. In *Proceedings of the 5th workshop on Data management for sensor networks*, DMSN '08, pages 35–41, New York, NY, USA, 2008. ACM.

[MJIG10] Svilen R. Mihaylov, Marie Jacob, Zachary G. Ives, and Sudipto Guha. Dynamic join optimization in multi-hop wireless sensor networks. *Proc. VLDB Endow.*, 3(1-2):1279–1290, September 2010.

[Mor66] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing.* International Business Machines Company, 1966.

[MSKG10] Scott Moeller, Avinash Sridharan, Bhaskar Krishnamachari, and Omprakash Gnawali. Routing without routes: The backpressure collection protocol. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 279–290, New York, NY, USA, 2010. ACM.

[MYC11] Cun-Ki Min, Heejung Yang, and Chin-Wan Chung. Cost based in-network join strategy in tree routing sensor networks. *Information Sciences*, 181(16):3443 – 3458, 2011.

[NN03] Badri Nath and Dragoş Niculescu. Routing on a curve. *SIGCOMM Comput. Commun. Rev.*, 33(1):155–160, January 2003.

[oEI03] Institute of Electrical and Electronics Engineers Inc. IEEE Std. 802.15.4-2003 “Wireless medium access control (MAC) and physical layer (PHY) specifications for low rate wireless personal area networks (LR-WPANs)”. October 2003.

[PG06] Aditi Pandit and Himanshu Gupta. Communication-efficient implementation of range-joins in sensor networks. In *In Proceedings of 11th International Conference on Database Systems for Advanced Applications*, pages 859–869, 2006.

[PR99] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA ’99. Second IEEE Workshop on*, WMCSA ’99, pages 90–100, Feb 1999.

[Res13] ABI Research. 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.

[RKY$^+$02] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. Ght: A geographic hash table for data-centric storage. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA ’02, pages 78–87, New York, NY, USA, 2002. ACM.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[SBB09] Mirco Stern, Erik Buchmann, and Klemens Böhm. Towards efficient processing of general-purpose joins in sensor networks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE ’09, pages 126–137, Washington, DC, USA, 2009. IEEE Computer Society.

[SBB10] Mirco Stern, Klemens Böhm, and Erik Buchmann. Processing continuous join queries in sensor networks: A filtering approach. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, pages 267–278, New York, NY, USA, 2010. ACM.

[SHB14] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap), rfc 7252. 2014.

[SJKZ11] Razieh Sheikhpour, Sam Jabbehdari, and Ahmad Khadem-Zadeh. Comparison of energy efficient clustering protocols in heterogeneous wireless sensor networks. *International Journal of Advanced Science and Technology*, 36:27–40, 2011.

[SPF14] Alan B. Stokes, Norman W. Paton, and Alvaro A. A. Fernandes. Proactive adaptations in sensor network query processing. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, pages 23:1–23:12, New York, NY, USA, 2014. ACM.

[Taj77] William D. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Commun. ACM*, 20(7):477–485, July 1977.

[ULS14] Awada Uchechukwu, Keqiu Li, and Yanming Shen. Energy consumption in cloud computing data centers. *International Journal of Cloud Computing and services science*, 3(3), 2014.

[UTK13] Muhammad Umer, Egemen Tanin, and Lars Kulik. Opportunistic sampling-based query processing in wireless sensor networks. *GeoInformatica*, 17(4):567–597, 2013.

[WTC+12] T Winter, P Thubert, T Clausen, J Hui, R Kelsey, P Levis, K Pister, R Struik, and J Vasseur. Rpl: Ipv6 routing protocol for low power and lossy networks, rfc 6550. *IETF ROLL WG, Tech. Rep*, 2012.

[YF04] O. Younis and Sonia Fahmy. Distributed clustering in ad-hoc sensor networks: a hybrid, energy-efficient approach. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, pages –640, March 2004.

[YGE01] Yan Yu, Ramesh Govindan, and Deborah Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks.

Technical report, Technical report ucla/csd-tr-01-0023, UCLA Computer Science Department, 2001.

[YLOT07] Xiaoyan Yang, Hock Beng Lim, Tamer M. Özsu, and Kian Lee Tan. In-network execution of monitoring queries in sensor networks. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 521–532, New York, NY, USA, 2007. ACM.

[YLZ06] Hai Yu, Ee-Peng Lim, and Jun Zhang. On in-network synopsis join processing for sensor networks. In *Mobile Data Management, 2006. MDM 2006. 7th International Conference on*, pages 32–32, May 2006.

[YZLZ05] Fan Ye, Gary Zhong, Songwu Lu, and Lixia Zhang. Gradient broadcast: A robust data delivery protocol for large scale sensor networks. *Wirel. Netw.*, 11(3):285–298, May 2005.

[ZG04] Feng Zhao and Leonidas J Guibas. *Wireless sensor networks: an information processing approach.* Morgan Kaufmann, 2004.