Efficient and Reliable Data Dissemination and Convergecast in Wireless Sensor Networks

by

Sain Saginbekov

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

October 2014

Abstract

With the availability of cheap sensor nodes now it is possible to use hundreds of nodes in a Wireless Sensor Network (WSN) application. Since then WSN applications have been being used in a wide range of applications, including environmental, industrial, military, health-care and indoor applications. WSNs are composed of sensor nodes, also known as *motes*, that are small in size, usually battery powered, and have limited memory and computing capabilities.

As opposed to other wireless networks of more powerful nodes such as laptops, cellular phones, PDAs, etc., where communications can occur between any two nodes, in WSNs there are mainly two communication types: (i) *broadcast*, where a designated node, called a *sink*, disseminates data to all other nodes and (ii) *convergecast*, where all nodes send their generated data to the sink.

After deploying sensor nodes in an area of interest, they are usually unattended for a long time. Since motes are battery powered, the energy conservation is of great importance. Furthermore, due to limited resources such as computing, memory and energy, harsh environmental conditions and buggy programs, wireless sensors may experience a number of different types of faults.

Given the characteristics of sensor nodes and the environment they are deployed in, any WSN communication protocol and algorithm should be energy efficient and tolerant to faults. Several efficient communication protocols have been proposed so far. However, there are several aspects that has seen very little activity in the literature: (i) Handling transient faults and (ii) Dealing with two or more sinks. Therefore, in this thesis, we are proposing to address some of the issues that are still open. Specifically, we are planning to look at fault tolerance in data dissemination and the development of an infrastructure for two sinks.

In this thesis, (i) we try to make data dissemination protocols resilient to faults that can corrupt values stored in the memory and messages by presenting two algorithms that when added to fault-intolerant dissemination protocols, make the code dissemination protocols fault-tolerant, (ii) we try to minimize drawbacks of existing code update maintenance algorithms by proposing a new algorithm that efficiently maintains code updates in WSNs, and (iii) we propose an efficient data aggregation convergecast scheduling algorithm for wireless sensor networks with two sinks.

To my parents

Acknowledgements

I am very grateful to my supervisor Dr. Arshad Jhumka, for giving me the opportunity to do research at Warwick University, for his invaluable advice, guidance and encouragement during my study years. I have been fortunate to have a supervisor like Arshad. I have learned a lot from him. This thesis would have not finished without his help.

I would like to thank my advisor Dr. Nathan Griffiths and co-advisor Dr. Matthew Leeke for their advice and feedback on this thesis.

I would also like to thank all other staff members of the Computer Science department for their help and support during my study period.

Finally, I would like to thank my parents and family for their love, encouragement and support.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated.

The work in this thesis is based on the following publications:

- S.Saginbekov, A.Jhumka. Towards Energy-Efficient Collision-Free Data Aggregation Scheduling in Wireless Sensor Networks with Multiple Sinks, submitted.

- S.Saginbekov, A.Jhumka. Efficient Code Dissemination in Wireless Sensor Networks, Future Generation Computer Systems, Volume 39, October 2014.

- S.Saginbekov, A. Jhumka. Towards Efficient Stabilizing Code Dissemination in Wireless Sensor Networks, The Computer Journal, Dec 2013; doi:10.1093/comjnl/bxt110.

- S.Saginbekov, A. Jhumka, Stabilizing Information Dissemination in Wireless

Sensor Networks, Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA), 2012.

- S.Saginbekov, A. Jhumka, Fast and Efficient Information Dissemination in Event- Based Wireless Sensor Networks, Proceedings of 11th IEEE International Conference on Ubiquitous Computing and Communications, 2012.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The advances in the area of wireless communications and Micro-Electro-Mechanical Systems (MEMS) technology have made available cheaper, smaller and multifunctional sensor nodes that are capable of communicating wirelessly. These features of sensor nodes have attracted the attention of the research community over the world as sensor nodes can be used in applications in different fields.

Wireless Sensor Networks (WSNs) are composed of sensor nodes, also known as motes, equipped with a processing unit, a transceiver unit, a power unit and a sensing unit. A typical sensor node usually is operated by battery power, has a limited memory and computation capability. These characteristics of sensor nodes restrict the node's capabilities. However, combining a number of these motes into a network makes them powerful and enables their use in a wide range of applications, including environmental, industrial, structural, health, and military applications [86, 23, 62, 116, 69, 74, 14]. They are also used for many indoor applications in our daily lives: for example temperature monitoring in office buildings, fire detectors in buildings and other applications in smart homes [60, 63].

Nevertheless, WSNs brought new challenges as previous communication protocols and algorithms designed for wireless networks of more powerful devices such as laptops, cellular phones, PDAs, etc., are not suitable for WSNs. Protocols designed for WSNs should deal with the constraints of the sensor node and the environment they are deployed in.

In this chapter we first explain the communication types of WSNs. After that we explain the motivation behind our work. Then we present contributions and, finally, present the structure of this thesis.

1.1 Communications in Wireless Sensor Networks

As opposed to other wireless networks where communications can occur between any two network devices, there are mainly two types of communications in WSNs: (i) One-to-many, also called *broadcast*, where data flows from the sink to sensor nodes, and (ii) Many-to-one, also called *convergecast*, where data flows from sensor nodes towards the sink. Most WSNs involve both of these communication types.

1.1.1 Broadcast

Broadcast type of communication is mainly used by a sink to transmit messages to all sensor nodes in the network. When sensor nodes are not in the coverage range of the sink, some sensor nodes could act as relays to forward the messages further down, making the communication multi-hop. These messages could be commands that request sensor nodes to do some task, queries that request sensor nodes to send specific type of messages, or program updates that change the nodes' program, etc. [85, 115].

WSNs are deployed once and, in general, unattended for a long time. During this time we may need to retask the network by changing parameters, functions, codes, etc. Changing these manually one-by-one is tedious, time and labour consuming especially when the number of nodes in the network is very large. Moreover, depending on the deployment place, manual change could be impractical or impossible altogether. For example, in [112] motes are deployed on trees while in [116, 97, 17] motes were deployed in harsh and hazardous places. Therefore, it is preferable to retask the network wirelessly by disseminating corresponding data. This poses other challenges such as designing efficient data dissemination protocols suitable for WSNs.

1.1.2 Convergecast

Convergecast is the main type of communication in WSNs, as the main goal of a wireless sensor network is to gather data about a physical object or event. Sensor nodes, after generating data, periodically, or after detecting an event or receiving a query, send their data to a more powerful device called a *sink*. The raw data generated by a sensor node may be delivered as it is or may be *aggregated* at each node on the way towards the sink. The sink usually forwards the collected data to a monitoring center through the network so that the data can be processed and analysed into meaningful information.

In wireless communications a message collision may occur at the receiving side; when a receiver hears from more than one device at the same time, the transmitted messages collide, resulting in unintelligible data. Thus, while convergecasting, if message transmissions are not controlled somehow, messages may be lost due to message collisions. Consequently, nodes should resend their messages to make sure that the sink receives them. Therefore, the bigger the number of collisions, the more the number of retransmissions and the more energy is consumed. Further, the number of collisions increases when the network becomes denser. One of the ways of decreasing the number of collisions is to schedule the nodes' transmissions such that no colliding nodes transmit at the same time.

1.2 Motivation

One of the constraints of motes is that they have limited power as they are usually operated by two AA (double A) batteries. When a node depletes its energy, the node dies and may change the topology of the network and may make the network disconnected. So the energy depletion of one node can make the entire network disfunctional. While sometimes it is possible to replace batteries or use solar panels as an energy source, sometimes it is impractical or impossible. Therefore, when designing algorithms and protocols for WSNs, one of the most important goals is energy conservation. Among other operations, communication consumes the most energy [6]. For this reason communication is an important function in WSNs.

Several efficient communication protocols, including data dissemination and data collection, have been proposed so far. However, there are several aspects that has seen very little activity in the literature: (i) Handling transient faults and (ii) Dealing with two or more sinks. Therefore, in this thesis, we are proposing to address some of the issues that are still open. In general, we are planning to look at fault tolerance in data dissemination and the development of an infrastructure for two sinks. In particular, we try to address the following issues.

First, faults typically occur in wireless sensor networks. Due to limited resources such as computing, memory and energy, harsh environmental conditions and buggy programs, wireless sensors may experience a number of different types of faults. As mentioned in [11], these faults can be classified as node failures and hardware faults, communication faults and software faults. Some of these faults lead to transient data faults [96, 40]. Transient data faults may severely impact the efficiency of the protocols and may lead to unexpected results. Several efficient data dissemination protocols exist in the literature, however, few of them consider transient data faults. *Therefore, we try to solve data dissemination problem in the presence of transient data faults.*

Second, due to transient link failures or node mobility, some nodes may not update their data during the dissemination phase. There exists algorithms that maintain data update. However, they have drawbacks such as high latency or high transmission energy consumption. *Therefore, we try to minimize those drawbacks.*

Third, due to sink failure, data collected from sensor nodes may not reach the monitoring center. Therefore, to make WSN applications more resilient to sink failures, more than one sink should be deployed, and nodes should send data to all of them. Moreover, there exist WSN applications where more than one sink exists, and sensor nodes are required to report data to each of these sinks [5]. In other words, convergecast is done to more than one sink. Up to now several data aggregation convergecast scheduling protocols have been proposed for networks with single sink and, to our knowledge, only few consider multiple sinks. However, the existing protocols designed for multi-sink WSNs provide convergecast scheduling from *many* nodes to *one* sink. *Therefore, as an initial work in this area, we try to solve convergecast scheduling from many nodes to two sinks.*

1.3 Contributions

In this thesis, we make the following contributions:

- We formalise the concept of code dissemination in WSN, and provide three refined specifications, viz.: strong, consistent and best effort code dissemination.

- We show that (i) there is no deterministic algorithm that solves strong code dissemination in the presence of transient faults, and (ii) there is no deterministic 1-local algorithm that solves strong code dissemination in the presence of a stronger class of transient faults, called *detectable* faults.

- We present two novel f-local algorithms called (i) BestEffort-Repair and (ii) Consistent-Repair that, when added to any fault-intolerant code dissemination protocol, solve (i) BestEffort code dissemination and (ii) Consistent code dissemination, and we prove the correctness of both protocols.

- We run real-world testbed and simulation experiments on our protocol, and show their correctness and performance, especially the locality property of the protocols.

- We present a case study where we add both protocols to an existing code dissemination algorithm, namely Varuna [95]. We equip Varuna with a specific detector which triggers the protocols upon detection of an error. We show that both BestEffort-Repair and Consistent-Repair induce very little overhead on Varuna in the presence of detectable transient faults. Further, Varuna, when executed in the presence of even a single transient fault, led to all the nodes downloading the wrong code. In contrast, when running Varuna with both protocols, all the nodes eventually obtained the new code.

- We present an efficient code update maintenance algorithm called *Triva*. Triva is more energy efficient and faster than existing algorithms of this type, and works best for event-based type of Wireless Sensor Network applications. The existing code maintenance algorithms consume energy due to transmission even when there is no new code in the network or if asymmetric links exist between nodes. We perform real-world testbed and simulation experiments, and show the superiority of Triva over other algorithms.

- We present an efficient data aggregation convergecast scheduling algorithm for wireless sensor networks that have two sinks and show its correctness. The algorithm is the first in its kind and is efficient in terms of energy and latency. Further, in the algorithm every node is assigned at most 2 transmission slots and contiguous reception slots which is important in applications where packet sizes are small [7, 56]. Moreover, the algorithm decreases the number of nodes that transmit twice to a minimum. We perform simulation and real-world testbed experiments to show our algorithm's performance.

1.4 Thesis Overview

This chapter has detailed the main goals, motivations and contributions of the research presented in this thesis. The remainder of this thesis is organised as follows:

Chapter 2 presents a survey on related works. In particular, data dissemination protocols and how they maintain data consistency are presented. Then some existing data aggregation convergecast scheduling algorithms are presented. This chapter also briefly reviews reliable broadcast protocols.

Chapter 3 describes the system and fault models under which our protocols are designed.

Chapter 4 presents two algorithms, Best-Effort Repair and Consistent-Repair, which make code dissemination protocols tolerant to transient state faults. Extensive simulations on a different number of faulty nodes are performed and their performances are presented. Moreover, real-world testbed experiments on TelosB motes are performed to confirm the simulation results. Also, this chapter presents a case study where our protocols are added to an existing code dissemination protocol to show their performances.

Chapter 5 presents our code update maintenance algorithm called Triva. Before giving details on the algorithm, the chapter gives a brief description on the weaknesses of other algorithms. Simulations and real-world testbed experiments are conducted to show the performance of Triva over other algorithms.

Chapter 6 presents an efficient data aggregation convergecast scheduling algorithm for WSNs with two sinks. The chapter first gives an algorithm which builds trees and balances the number of children. Then, the data aggregation convergecast scheduling algorithm is given. To show the performance of our algorithm simulation and testbed experiment results are given.

Finally, Chapter 7 gives a conclusion and discusses future work.

CHAPTER 2

Literature Survey

Lots of protocols have been developed for wireless networks. However, because of the limitations of the sensor node, they might not be suitable for WSNs. Therefore, a number of protocols have been designed specifically for WSNs since the first use of WSNs. All of them consider the limitations of the sensor node and are thus more efficient in terms of different metrics, such as energy and latency. In this chapter, we survey protocols related to the works presented in this thesis, i.e., data dissemination protocols, code update maintenance algorithms, and data aggregation convergecast scheduling algorithms. Further, for the sake of completeness, we review protocols focused on reliable broadcasting.

2.1 Data Dissemination Protocols

In WSNs dissemination protocols are typically used for sending commands, queries and code updates. Many data dissemination protocols have been developed up to now. While some of the protocols have been designed for delivering specific type of messages like delivering tasks [94], network parameters [111], and queries [117], others have been designed specifically for delivering code updates. In the following, we will first discuss about data dissemination protocols and then discuss about algorithms that are used to maintain data consistency.

In [111], the authors propose a Sensor Network Management System (SNMS) to monitor and control the node and network status. SNMS supports two traffic patterns: Collection and Dissemination. Collection is used to gather health data from the network, and Dissemination is used to distribute management commands and queries. Collection is performed by building a higher-quality tree rooted at sink, where each node continually update the parent selection. And for disseminating queries and commands, the authors propose a dissemination protocol, called Drip. Drip uses sequence numbers (versions) to identify whether the data received is new. The protocol proposed in [57], called CodeDrip, is another dissemination protocol designed for WSNs to disseminate small values. The main idea behind this protocol is to use Network Coding [3] to decrease the number of transmitted messages and consequently save energy. Network Coding is useful when recovering lost packets. For example, consider that a sink should broadcast two packets P_1 and P_2 to two nodes n_1 and n_2. If n_1 receives only P_1 and n_2 receives only P_2, then the sink have to retransmit two packets again: P_2 to n_1 and P_1 to n_2. Instead, using Network Coding technique, the sink transmits only one packet $P_3 = P_1 \oplus P_2$. When the nodes receive P_3, n_1 can obtain $P_2 = P_3 \oplus P_1$, while n_2 can obtain $P_1 = P_3 \oplus P_2$.

Some dissemination protocols specifically have been designed for disseminating code updates. The rest of the section describes code dissemination protocols existing in the literature.

One of the earliest protocols, called XNP (Crossbow In-Network Programming) [31], is a protocol which disseminates codes to the nodes that are in the communication range of the sink node. Nodes, when downloading, store the code capsules in the external flash memory (EEPROM). After disseminating the entire code, the sink broadcasts a query asking the nodes if they have missing capsules. The nodes scan the EEPROM for the completeness of the code. If necessary, nodes can request unreceived capsules from the sink by sending a negative acknowledgement (NACK) packet. The use of NACK instead of acknowledgement (ACK) is more efficient in terms of control traffic, as NACK is sent only for undelivered packets, which are considered to be much fewer than the number of successfully delivered packets. After downloading the entire code, a node copies the new code to the program memory and reboots the system.

One of the drawbacks of XNP is that there should always exist a bi-directional link between each node and the sink. Otherwise, nodes may not request for unreceived capsules from the sink. When the size of the network gets bigger, fulfilling this requirement will be challenging as nodes are limited in transmission range. Therefore, XNP may not work in large networks as only a subset of nodes that are in the range of the sink gets the new code. Another drawback of XNP is if the link between a node and the sink fails during code dissemination, then that node will not receive the new code at all.

The protocol proposed in [105], called MOAP (Multihop Over-the-Air Programming), is a reprogramming protocol which addresses the drawbacks of XNP. MOAP disseminates code updates in a multi-hop fashion, meaning that nodes which receive a new code from the sink or other neighbours, can relay the code further down. In MOAP protocol, a code is disseminated in a *neighbourhood-by-neighbourhood* fashion: at each neighbourhood only a few nodes become sources of the code. A node can be a source if it has the entire code and has received a request message called *subscribe* message from receivers for its code advertisement message called *publish* message, which contains the version number of the code. MOAP is a NACK based protocol. Whenever a node detects a missing packet, it sends a NACK packet to the source to ask the source to retransmit the missing packet. MOAP uses the *Sliding Window* technique. In general, *Sliding Window* technique is used for controlling transmitted data packets between two nodes where reliable and sequential delivery of data packets is required.

A protocol called Deluge [50] is a protocol that is used to update codes of nodes over the radio. Like MOAP, it uses NACK and windowing to manage required segments during the download process. However, it differs from previous protocols in that it divides the new code into fixed-size pages, which are further divided into packets. The pages are delivered in a sequential order. Unlike MOAP, Deluge uses pipelining technique, which does not wait for the entire code before forwarding it further. As soon as a node receives a complete page it advertises its availability and can forward the page further upon request. The pipelining technique decreases the latency of dissemination. Another advantage of dividing the code into pages is if a page is delivered with an error, then only that page is retransmitted. In addition, unlike MOAP, Deluge deals with asymmetric links, which is very common in wireless networks, using three-way handshaking (ADV-REQ-DATA). By sending ADV and receiving REQ, two nodes understand that there is a link between them. In general, three-way handshaking method is used to establish a connection between two devices. Another protocol which uses pipelining technique, called Typhoon, has been proposed in [80]. The idea is that it is enough to have only two hops distance between concurrent transmissions if different frequency channels are used. Thus, the main difference of Typhoon is that it uses channel switching technique to decrease the dissemination latency.

A protocol called MNP (Multihop Network Reprogramming), proposed in [70], is a multi-hop network reprogramming protocol. As Deluge, before dissemination starts, MNP divides the new code into segments each having a fixed number of packets. It also uses the idea of using pipelining technique to make disseminations fast. However, unlike Deluge, MNP uses a sender selection algorithm to select a sender in a neighbourhood. It selects the sender in a way such that only one node broadcasts the code at a time in a given neighbourhood. The sender is selected after competing with other potential senders in the neighbourhood according to its number of requesters. While competing, nodes include their number of requesters in advertisement packet to inform other competing nodes in the neighbourhood about its number of requesters. After transmitting advertisement packets N times, the node with the largest number of requesters will be selected as a sender. Another property of MNP is that if a node finds that there is a node transmitting in its neighbourhood, then it goes to the *sleep* mode to conserve energy. One of the issues of using pipelining technique is that all nodes' radio in the network should be turned on during reprogramming to support pipelining. Unlike Deluge, the latter property of MNP tries to resolve this issue.

Because of the features of wireless sensor networks, e.g., transient link failures, node failures, node mobility, or when a node is in sleep mode, not all nodes receive disseminated data during the dissemination phase. To make all nodes download the disseminated data, in addition to a dissemination protocol, there should be an algorithm that ensures data consistency. The next section describes algorithms that maintain data consistency.

2.1.1 Algorithms for Maintaining Data Consistency

Data consistency maintenance algorithms should update the nodes as soon as they are able to receive messages, and this process should consume as little energy as possible.

The simplest way to ensure the delivery of new data to all nodes in the network is that any updated node should continuously check the consistency of its neighbours, by broadcasting *advertisement*(ADV) messages. Blindly broadcasting ADV messages causes the so-called "broadcast storm problem" [93], a scenario in which there is excessive number of redundant ADV broadcasts. Blindly broadcasting is expensive in terms of energy and throughput. Therefore, this way is not suitable for WSNs.

There exist algorithms which have been developed specifically to address this problem. We will describe them in the next sections and go in details of two, namely Trickle and Varuna, in Chapter 5 as they are most related to our work. One of the algorithms that address the above problem is Trickle [79], which uses a "polite gossip" policy to address the problem. In Trickle, a node suppresses its advertisement message transmissions when it hears a number of messages identical to its own. The next two algorithms, DIP and DHV, use the Trickle timer to maintain data consistency. However, instead of advertising the version of each code, they try to advertise a group of code versions and use other mechanisms to try to reduce message overhead. Another algorithm that addresses the broadcast storm problem is Varuna [95]. Varuna consumes constant energy in steady phase when the network is in a steady state, i.e., when all nodes have the same data. These algorithms are explained in the next sections.

2.1.1.1 Trickle

Trickle [79] is an algorithm that is used to suppress unnecessary transmissions. Many dissemination algorithms such as [50, 80, 81, 111, 32, 57] adopt Trickle to maintain data consistency. The idea of this algorithm is that when a node hears the same advertisement as its own several times, the node will not broadcast the advertisement message. Precisely, in Trickle, every node broadcasts advertisement messages that contain a metadata that includes the version number of the data, at most once per period t picked from the range $[\tau/2, \tau]$. If a node hears more than k identical version

Event	**Action**
τ Expires	Double τ, up to τ_h. Reset c, pick a new t
t Expires	if $c < k$, transmit
Receive same metadata	Increment c
Receive newer metadata	Set τ to τ_l. Reset c, pick a new t
Receive newer code	Set τ to τ_l. Reset c, pick a new t
Receive older metadata	Send updates

Table 2.1: Trickle Pseudocode.

numbers from its neighbours before it transmits its advertisement, it suppresses its broadcast and doubles the value of τ up to τ_h, which is an upper bound for τ. If it hears a different version number, τ is set to τ_l, which is a lower bound for τ (see Table 2.1 for a pseudocode).

By increasing the broadcast interval, τ, Trickle sends fewer advertisement messages, thus saving energy. By decreasing τ, Trickle can update nodes more quickly. So, there is a trade-off between dissemination latency and energy to be achieved when selecting τ.

In Trickle, the number of advertisement messages increases linearly as a function of time, as the dissemination is irrespective of the mission of the WSN application.

2.1.1.2 DIP and DHV

Another data consistency maintenance protocol, named DIP, has been proposed in [81]. DIP works better than Trickle in terms of message overhead when there exist several code items per node in the network. The idea of DIP is that instead of advertising the version number of each data item, DIP advertises the hash of its key and versions of all data items. Using hash enables DIP to detect a difference between versions in $O(1)$ time. Whenever a node detects a difference between hashes of size S, the node responds with two hashes of size $S/2$. By decreasing the hash size, nodes can find the data item that has changes. Therefore, if there are D data items, the search algorithm in DIP finds N new data items with $O(Nlog(D))$ packets ($N < D$), whereas Trickle finds with $O(D)$ messages [32].

A code consistency maintenance protocol, called DHV, has been proposed in [32]. The goal of DHV is to further reduce the message overhead. DHV's key contribution is its technique to efficiently determine when to perform code updates. DHV is based on the observation that when two code versions are different, their corresponding version numbers often differ in only a few least significant bits of their binary representation. Precisely, to detect and identify code changes, a node n in DHV first sends a hash of all versions of data items. If a receiving node m detects a difference, m broadcasts the checksum of all versions. Node n compares the received checksum with its own checksum, and finds the location of bits, say at index i, that are different. Then n sends the bit slice consisting of all bits of all versions at index i to m. After receiving the bit slice, node m checks to find which version number is different and sends that version number to n. Then n compares the received version with its version to check whether it is updated. Therefore, the total number of messages to identify new items is $O(N)$ for N new data items.

Nevertheless, both of the protocols, DIP and DHV, use the Trickle timer to suppress unnecessary transmissions. Therefore, the number of messages in a given period T, as noted in Varuna [95], is still linear $O(T)$ in the steady phase.

2.1.1.3 Varuna

Varuna [95] is another protocol which supports data consistency maintenance. Unlike Trickle, DIP and DHV, where there is a continuous energy consumption due to ADV broadcasts, energy consumption in Varuna is constant in the steady phase, a phase where no dissemination is being done. (Note that, if the number of data items per node is more than one, then the mechanisms used in DIP and DHV can also be applied to Varuna to further reduce message overhead.) To achieve constant energy consumption in the steady phase in Varuna, nodes send advertisement messages only when there is a change in their neighbourhood topology or metadata since their last advertisement transmissions. To learn about this change, each node stores its neighbour IDs in its *neighbourhood table*. A node stores only the IDs of neighbours that are consistent with it, i.e., neighbours that have the same data version number as it. For example, if a node n_1 sends a message to a node n_2, n_2 checks the existence of n_1's ID in its neighbourhood table. If it exists in the table, it is assumed to be consistent with the n_2, therefore, n_2 does not send an advertisement message. Otherwise, n_2 checks the consistency of its data with the node n_1 by sending an advertisement message. If the version numbers are equal, n_2 stores the ID of n_1 in its table. If they are different, the node which has the bigger version number sends an advertisement message to let other nodes request and download the data with the bigger version. After receiving the new data, the node resets its table (as it has to detect new consistent nodes). When all nodes receive the newest data, every node's table will contain the IDs of all neighbouring nodes. This state stops sending advertisement messages, which makes energy drain constant in steady phase after some time.

In Varuna, a node detects inconsistency only if it receives a message from a node which has a smaller version number. It means that, a node will not update its data unless it communicates with a node with a new version number. This makes the *update latency*, the time from the injection of new data to the time when all nodes receive the new data, dependent to a communication rate of a node. Therefore, update latency in Varuna increases linearly with data communication rate or with the event time if the application is event based.

2.1.2 Discussion

To handle the temporary node failures and disconnections problem during the data dissemination phase, i.e., to maintain data/code consistency, MOAP uses the *Late Joiner* mechanism where nodes should periodically send *publish* messages to advertise their version numbers. This mechanism may make the message overhead too large in steady phase and the latency too long when there is a new code in the network. The authors noted that the version number could be included in data packets instead of sending the version number alone as one packet. XNP does not mention how they maintain code updates. However, as it is designed for one-hop networks, it is enough for the sink to broadcast the new code periodically. Drip, CodeDrip, DHV, Deluge and Typhoon adopt Trickle algorithm to maintain code updates. MNP uses the same idea that nodes advertise every random interval and this random interval increases exponentially when the node does not receive any requests from its neighbours, which can make the latency too long.

2.2 Data Aggregation Scheduling

In WSNs, packets generated by sensor nodes can be delivered to a sink in two ways: 1) using aggregation technique, where packets are aggregated at each parent node, or 2) each generated packet is equally important and is delivered without aggregation. In our work, which will be presented in Chapter 6, we assume that data aggregation is used.

Data aggregation is a well-known technique and used to reduce the number of packets to be transmitted by excluding redundancy, thereby conserving energy [68, 4]. A parent node, after receiving packets from its children, aggregates them according to a function such as MIN, MAX, AVERAGE, etc., or the function could be just removing duplicates.

During convergecast, transmitted packets could collide if nodes' transmissions are not scheduled accordingly. Therefore, scheduling of packet transmissions influences the overall performance, including energy consumption, latency and throughput.

Although the scheduling problem has been proven to be NP-complete [39, 28], a number of collision-free convergecast scheduling protocols have been proposed so far. Many of them are designed with different perspectives in mind. For example, while some of them use data aggregation, multi-channels or load balancing, others do not. In some of them the sink computes the scheduling by collecting all information about the network while in some scheduling is done in a distributed manner, requiring only local information.

In [75], a centralized energy-efficient collision-free scheduling is proposed. The idea is to construct a set of trees in advance and use them dynamically at different rounds keeping the load balance of the nodes.

Authors of [104] propose TreeMAC, a TDMA based MAC protocol designed for real-time high-data-rate applications. They address possible horizontal collisions, i.e., collisions of nodes that are equidistant from the root, by assigning different frames (a set of slots). And address possible vertical collisions, collisions of nodes that are on the same path from the root (i.e., when one node is a descendant of another), by assigning different slots. The protocol achieves 1/3 of the maximum throughput. In [9], to reduce the latency, authors propose a collision-free scheduling algorithm by first constructing a convergecast tree and then allocating different codes (DSSS/FHSS) to nodes.

As some hardware, such as Micaz and Telos, provides multiple frequencies, some works done so far use multi-channel communication to improve communication performance. In [119], the authors present a protocol which allocates different channels to vertex-disjoint trees rooted at the sink and exploits parallel transmissions among trees. The protocol improves network throughput and reduces collisions and is centralized. Other works that use multiple frequencies can be found in [51, 64].

A work which considers, like our protocol in Chapter 6, the number of active-sleep transitions is presented in [56]. Their idea is to assign contiguous time slots to the children of a node. However, they assume several powerful gateways and it is a cluster based protocol.

There are works on data aggregation scheduling which mainly focus on data aggregation latency and give the latency bounds for their protocols [120, 49, 26]. A work in [53] proposes a crash-tolerant data aggregation scheduling protocol and shows some impossibility results in the presence of crash failures. All these protocols consider a *single sink* in a network which means they will not work properly in a network with more than one sink.

Protocols that have been developed on data communication for networks with multiple sinks can be found in [90, 108, 21, 59]. A scheme proposed in [90] performs data collection from many nodes to many sinks. In [108], the authors propose an algorithm that builds two node-disjoint paths from every node to two different (drains) sinks to collect data to two sinks. The goal of the algorithm was to address the problem of any single node failure. These algorithms address the problem at routing level, i.e., neither of these two schemes generate a collision-free aggregation schedule for a wireless sensor network with multiple sinks.

The work presented in [59, 21] are the most relevant to our work presented in Chapter 6. In [59], in addition to an algorithm that forms shortest path trees rooted at each sink in a network with multiple sinks, authors propose a scheduling algorithm that use a graph coloring algorithm. The authors of [21], propose two algorithms for scheduling data aggregation in multiple-sink sensor networks. The first of the algorithm is Voronoi-based scheduling where the sensing area is divided into regions forming k forests, one for each sink. Then the algorithm makes schedules for nodes. The second algorithm is Independent scheduling which differs from the first in forest construction. As can be observed, both of the works propose data aggregation scheduling algorithms where different portions of sensor nodes aggregate their data to a single sink, whereas we consider the case where many nodes aggregate their data to two sinks.

2.3 Reliable Broadcast

Reliable communication is preferred in most communication systems. Moreover, for some systems reliability is of great importance. In wireless communication, achieving reliable broadcast is more challenging than its wired counterpart as the wireless channel is prone to failures such as collision and interference.

In data dissemination, it is important that the new data is delivered to nodes in its entirety and without any bit changes. Since we focus on data dissemination protocols, the problem of reliable broadcast is relevant. We provide a brief survey here although the works assume permanent or Byzantine failures, where a node can fail in arbitrary ways and behave unusually [72].

The work proposed in [67] is one of the earliest work that deal with the broadcast problem in multihop radio networks. The authors propose fault-tolerant broadcasting algorithms and give algorithms' asymptotic bounds on completion time. They assume permanent faulty nodes of unknown locations that do not receive and send messages.

In [65], the author shows that it is possible to obtain a reliable broadcast whenever the number of Byzantine nodes, nodes which may behave arbitrarily, f, is no more than some value. This f is defined in terms of a communication range r. Moreover, the author shows that it is impossible to obtain reliable broadcast when f is bigger than some threshold value. The work assumes the existence of a prefixed time-slotted transmission schedule and everyone follows this schedule to avoid collisions.

In [20], the authors improve on [65] by making possibility bounds tighter. In particular, it has been shown that it is possible to achieve reliable broadcast when the number of faulty(Byzantine) nodes is strictly less than the threshold value for which in [65] it is shown that it is impossible to achieve reliable broadcast.

In [66], unlike the previous work where there is no address spoofing and collision, the authors relaxed this assumption and showed that reliable broadcast is possible even in the presence of collisions and address spoofing as long as they are bounded and the number of faulty nodes is less than some threshold value.

In [18], the authors address the broadcast problem in the presence of Byzantine faults with faulty nodes having bounded number of messages m_f. They show the possibility of reliable broadcast whenever the number of messages, m, of the correct node is lower bounded by some value defined in terms of m_f. They assume the existence of a prefixed time-slotted schedule, but faulty nodes may not follow the schedule, thereby making collisions.

In [87], the authors propose a protocol which is *safe*, i.e., correct nodes do not download an incorrect message. The protocol guarantees this property whenever $D \geq H+2$, where D is the shortest distance between two Byzantine nodes and H is a protocol parameter which is assumed to be known by all correct nodes. The paper also discusses the possibility of reliable broadcast in the *torus* network whenever $D \geq 5$ and $H = 2$. The same authors generalized this result to planar graphs in [88]. In particular, the authors show that for $D > Z$, where Z is the maximal number of edges per polygon, it is possible to achieve reliable broadcast.

CHAPTER 3

System and Fault Model

In this chapter, we present the models under which our protocols are designed. In particular, we present what network types and faults we assume in our protocols.

3.1 Graphs and Networks

We define a wireless sensor node as a computing device equipped with a wireless interface and associated with a unique identifier. Communication in wireless networks is typically modelled with a circular communication range centred on the node. With this model, a node is thought as able to exchange data with all devices within its communication range.

A wireless sensor network is a collection of wireless sensor nodes and is modelled as a directed graph $G = (V, A)$, where V is the set of wireless sensor nodes of size $|V|$, and A is a set of arcs or directed links. Each directed link is an ordered pair of distinct nodes (m, n), meaning node m can communicate with node n. For a directed link (m, n), we call n a *downstream neighbour* of m, and m an *upstream neighbour* of n. We denote by M_d and M_u, the set of m's downstream and upstream neighbours, respectively. We also assume that, for every node m, $M_d, M_u \neq \emptyset$. Whenever we say a node n sends a message, we mean n sends the message to its downstream neighbours, and when we say n receives a message, we mean n receives the message from its upstream neighbours.

The d-hop neighbourhood of a node m, denoted by M^d, is a set of nodes such that the length of the shortest path from m to a node in the set is at most d. We say that two nodes m and n can collide at node p if $(m, p), (n, p) \in A$[1].

3.2 Distributed Programs

We model the processing on a WSN node as a process containing non-empty sets of variables and actions. A distributed program P is then a finite set of communicating processes. We represent the communication network of a distributed program by a directed connected graph $G = (V, A)$, where V is the set of processes and A is a set of directed links. A link $(m, n) \in A$ means that a process m can communicate with a process n.

A variable v_i takes values from a fixed and finite domain D_i. We denote a variable v_i of process n by $n.v_i$. Each process n has a special buffer variable, denoted by $n.b$, modelling a FIFO queue of incoming data sent by other nodes. This variable is defined over the set of (possibly infinite) message sequences. Every variable of every process, including the buffer variable, has a set of initial values. The *state* of a program P is an assignment to variables of values from their respective domains. The set of *initial states* is the set of all possible assignments of initial values to variables of the program. A state is called *initial* if it is in the set of initial states. The *state*

[1]We will say two nodes m and n can collide if such a node p exists.

space of the program is the set of all possible value assignments to variables. An action a at process n updates one or more variables of n atomically.

3.3 Semantics

3.3.1 Program

We model a distributed program as a transition system $P = (\Sigma, I, \Delta)$, where Σ is the state space, $I \subseteq \Sigma$ the set of initial states, and $\Delta \subseteq \Sigma \times \Sigma$ the set of state transitions (or steps). A computation of P is a maximal sequence of states $s_0 \cdot s_1 \ldots$ such that $\forall i > 0, (s_{i-1}, s_i) \in \Delta$. If the computation is finite, then it terminates in a *final* state. A state s of a computation is final if there is no state s' such that $(s, s') \in \Delta$.

In a given state s, several processes may be ready to execute, and a decision is needed to decide which one(s) execute. A *scheduler* is a predicate over the set of computations. In any computation, each step (s, s') is obtained by the fact that a non-empty subset of enabled processes atomically execute an action. This subset is chosen according to the scheduler. A scheduler is said to be *central* [35] if it chooses only one ready process to execute an action in any step. A scheduler is said *distributed* [22] if it chooses at least one ready process to execute an action in any execution step. A scheduler may also have some fairness properties [37]. A scheduler is *strongly fair* if every process that is ready infinitely often is chosen infinitely often to execute an action in a step. A scheduler is *weakly fair* if every continuously ready process is eventually chosen to execute an action in a step. A *synchronous* scheduler is a distributed scheduler where *all* ready processes are chosen to execute an action in a step.

In this thesis, we assume a synchronous scheduler, capturing a synchronous system where an upper bound exists on the time for a process to execute an action.

This assumption is not unreasonable as WSNs are often time-synchronized to correlate sensor readings from different devices. Overall, in this thesis, we assume a *synchronous system model.*

3.3.2 Specification

A specification is a set of computations. A program P satisfies a specification Φ if every computation of P is in Φ. Alpern and Schneider [8] stated that every computation-based specification can be described as the conjunction of a safety and liveness property. Intuitively, a safety specification states that something bad should not happen, i.e., the safety specification defines a set of computation prefixes that should not appear in any computation. On the other hand, a liveness specification states that something good will eventually happen, i.e., the liveness specification specifies a set of state sequences such that every computation has a suffix in the set.

We assume the specification to be fusion-closed and suffix-closed. A specification is fusion-closed if two computations $\alpha \cdot s \cdot \beta$ and $\lambda \cdot s \cdot \gamma$ are allowed by the specification, then so are the computations $\alpha \cdot s \cdot \gamma$ and $\lambda \cdot s \cdot \beta$, where α and λ are finite prefixes of computations, γ and β are suffixes of computations, and s is a program state. Thus, fusion closure means that the next step of a program depends on the current state and not on the previous history of the execution. A specification is suffix-closed if a computation is allowed by the specification, then all suffixes of the computation are allowed by the specification. Suffix closure enables us to discuss the correctness of the program on the basis of its current state rather than potentially arbitrary long program history [92]. The assumption of fusion closure is, in general, reasonable given that most, if not all, protocol specifications in WSNs are inherently fusion closed. Fusion closure basically implies that history information is available in every state of the program. And usually history information is local in WSNs specifications. For example, when assigning slot in TDMA, the collision freedom is an important property and it states that no two nodes within 2 hops of each other will have the same slot. Thus, if a node n is choosing a slot in a state s, then n should have enough history information to determine whether any node within 2 hops have chosen the same slot. Often, this history information is piggybacked onto application messages, or specific control messages. Fusion closure is important in our work because, like mentioned in the example above, it enables nodes to choose their collision-free time slots. Moreover, if a data fault occurs, fusion closure enables recovery to take place. Further, any non-fusion closed specification can be transformed into an equivalent fusion closed one, through the addition of history information to the program. And that history information will be part of the state of the program. For example, the specification "$x = 4$ implies that previously $x = 2$" is not fusion closed. However, it can be transformed by adding a history variable h that is used to record the previous value of x. Thus, the above non-fusion closed specification can be transformed to the fusion closed specification "never ($x = 4$ and $(x = 2) \notin h$)" [41].

3.3.3 Communication

We model synchronous communication as follows: after a process m broadcasts a message in state s_i, a downstream neighbour process n executes the corresponding receive in state s_{i+1}, i.e., the corresponding receive is executed before any other enabled actions of process n, such that, in some sense, message deliveries take higher priority. It is reasonable because most of the WSNs applications are written on interrupt-driven operating system called TinyOS [1]. Typically there are two kinds of interrupts in TinyOS - clock/timer and radio. And a radio interrupt occurs when the radio receives a message [106].

3.3.4 Faults

A fault model stipulates the way programs may fail. We consider *transient data faults* that corrupt the state of the data dissemination program by artificially corrupting the values held by the variables and messages. These faults are also known as soft errors [96, 40]. There are other types of faults that can occur in the network: (i) crash faults that may occur, for example, due to energy depletion, (ii) message losses that may occur, for example, due to message collisions or link failures and (iii) Byzantine faults that may occur, for example, due to hardware faults. As one of the goals of the data consistency maintenance algorithm is to address crashes and message losses (see Section 2.1.1), the data dissemination algorithm can tolerate the first two faults by itself. The third fault type, Byzantine faults, has been theoretically studied (see Section 2.3). However, to our knowledge, there is very little evidence of the actual occurrences of Byzantine faults in WSNs.

Formally, our fault model is a set F of faulty actions [12]. These are similar to program actions, as they may modify the variables of programs and thus alter the program state. We say that a fault occurs if a fault action is executed. Fault actions can interleave program actions and they might or might not be executed when enabled. We say a computation is F-affected if the computation contains program transitions and transitions from fault model F. We also assume that the sink is able to retrieve an uncorrupted version of the data and version number (with the sink acting as a gateway), though the sink itself can be corrupted.

CHAPTER 4

Repair: Making Code Dissemination Protocols Fault-tolerant

4.1 Introduction

Due to limited resources such as computing, memory and energy, harsh environmental conditions and buggy programs, wireless sensors may experience a number of different faults [19]. As mentioned in [11], these faults can be classified as node failures and hardware faults, communication faults, and software faults. Some of these faults may cause non-deterministic bit-flips in the main memory and lead to memory corruptions [40, 33]. Moreover, because of the erroneous nature of wireless communication transmitted messages may be corrupted [96]. For example, according to the data obtained from the deployment at the outskirts of Uppsala, Sweden, the ratio of corrupt packets to correctly received packets was at least 0.5 [47].

Due to these faults nodes may behave arbitrarily and negatively impact the entire network. For example, in a deployment at Reventador [116], a software error led to a 3-day network outage. In [16], due to temperature differences, network communication failed during mornings and evenings, but worked during day and night. Recently, as reported in [55], transient faults have occurred with a probability of approximately 0.1% in a large-scale deployment and such transient faults severely impact on the efficiency of the protocols.

There have been several works done that are tolerant to transient memory faults or present memory protection mechanisms from some actions that lead to this type of faults, and [27, 30, 61, 71, 33] are among others. Also, there exists a tool that is designed specifically for wireless sensors to emulate memory faults to check the reactions of software to these faults [29].

As mentioned in Chapter 2, several data dissemination protocols have been proposed thus far [111, 81, 32, 57, 70, 79, 95, 89]. However, to the best of our knowledge, none of them tolerates transient data faults, i.e., data faults that corrupt the state of the dissemination protocol due to memory and message corruptions.

Given that several data dissemination protocols work by advertising the metadata, viz. version number[1], of the new data, e.g., [111, 32, 81, 50, 80, 70, 105, 79, 95, 15], any corruption of the version number in the advertisement messages (message corruption) or those stored at the nodes (memory corruption) can, in the worst case, lead to the network nodes having stale data, thereby reducing the ability of the network to perform properly. Thus, it is important to make these data dissemination protocols tolerate these transient data faults.

As most of the dissemination protocols have been designed for disseminating code updates, we say *code dissemination* instead of *data dissemination* in the remaining part of this chapter.

When a node decides that another node has an updated code fragment, generally, it makes a request to the updated node which, subsequently, sends (i.e., broadcasts) the update to the requesting node. The process of advertising code updates, sending

[1]In this chapter, whenever we say metadata, we mean version number.

code requests and downloads is energy consuming. As the communication part consumes a large portion of energy and the significant amount of energy per transmitted bit used [99, 100], and given the size of codes, which may vary from 20 bytes up to tens of kilobytes [31, 77, 45, 85, 70, 50], code dissemination consumes a significant amount of energy. This is exacerbated when transient faults occur, as nodes may mistakenly request and download stale code. In the worst case, the whole network may download the stale code, at great energy expense.

There exists a hierarchy of fault tolerance properties namely fail-safe fault tolerance (which ensures that a program always satisfies its safety specification), non-masking fault tolerance (which ensures that a program always satisfy its liveness specification, even if safety is temporarily compromised) and masking fault tolerance (which ensures that both safety and liveness are satisfied, even in the presence of transient faults) [12]. These fault tolerance types are shown in Table 4.1. As a simple example, consider traffic light systems. A safety property of these systems could be that at any point in time no two traffic lights that are situated on the perpendicular roads turn on green. In the presence of faults if two traffic lights show green simultaneously, then the system is not fail-safe tolerant. In this example, as safety is much more important than liveness, the system should be at least fail-safe. So, the system should never violate the safety property. However, if it violates the safety property but then eventually corrects the fault, i.e., if the system works properly again after some time, it is said to be non-masking fault tolerant. While masking fault tolerance does not allow traffic lights to turn on green simultaneously, and continues working properly in the presence of faults.

In the context of code dissemination, fail-safe fault tolerance amounts to a node not downloading any code if it believes the code to be old. Masking fault tolerance means that all nodes will only download the new code (as if no fault has occurred), and only once. On the other hand, non-masking fault tolerance allows some erro-

	Live	Not Live
Safe	Masking	Fail-safe
Not Safe	Non-masking	

Table 4.1: Fault tolerance types

neous downloads (i.e., downloads of old code) before eventually all nodes download the updated code. Such erroneous downloads are only allowed to occur finitely, though.

Given that code update dissemination is energy consuming, it is thus preferable to reduce the number of erroneous downloads while ensuring that all nodes eventually download the updated code. Fail-safe fault tolerance is not suitable as it means that some nodes may not update (which will impact of the usefulness of the network). Masking fault-tolerant code dissemination protocols would minimise the number of erroneous downloads but, given the nature of the WSNs and of the dissemination process, masking fault tolerance is not practical. Because it is expensive or impossible to design such complex protocol for resource constrained sensor nodes that allows to download only new codes. For example, it is difficult to handle such a case: if the version number of a new code, say 5, in a node n has been corrupted to 4, while the version number of the old code of a node m has been changed from 4 to 5, then n will download the old code of m. On the other hand, non-masking fault tolerance means that nodes may erroneously download old code only finitely, but they will eventually download the updated code. However, a small number of erroneous downloads can be tolerated if this means that the network state is consistent, allowing the proper dissemination of the updated code.

To design non-masking fault tolerance, it is both necessary and sufficient for a program to contain a specific type of fault tolerance component called a *corrector*. A corrector is a class of program component that enforces a predicate on the execution of a program. There exists different correctors that can guarantee non masking fault tolerance for a given program, however they may differ in their efficiency.

In this chapter, instead of proposing a specific non-masking fault-tolerant code dissemination protocol[2] for WSNs, we address the problem in a different way: we first provide an abstract specification of the code dissemination problem, and based on the definition, we propose (i) a definition of a corrector protocol and (ii) two *corrector* protocols, called *BestEffort-Repair* and *Consistent-Repair*. Each can be added to any existing (fault-intolerant) code dissemination protocol to transform it into a fault-tolerant code dissemination protocol. Specifically, since the corrector protocol is designed based on the code dissemination specification, rather than on an actual implementation, if the corrector is added to any code dissemination implementation that satisfies the dissemination specification, then the resulting protocol is non-masking fault tolerant [10]. Further, to detect state corruption, a detector component, which detects the validity of a predicate in a given state, is designed based on the protocol implementation.

The two corrector protocols developed has enabled us to observe a tradeoff during recovery: Consistent-Repair results in a lesser number of erroneous downloads than BestEffort-Repair. However, BestEffort-Repair has a shorter completion time in that Consistent-Repair needs more time to make better update decision. A shorter recovery time means that the network state becomes consistent faster, and can perform useful work faster.

Overall, our approach is as follows: *given a fault-intolerant code dissemination protocol, we design a protocol-specific detector together with a generic corrector component to obtain a corresponding non-masking fault-tolerant code dissemination protocol.*

This chapter is structured as follows: We present a definition and specifications for code dissemination in Section 4.2. We present some theoretical results in Sec-

[2]Henceforth, whenever we refer to fault tolerance in code dissemination, we mean non-masking fault tolerance.

tion 4.3. In Section 4.4, we present two f-local corrector algorithms that stabilise the code dissemination of code updates. In Section 4.5, we present simulation results to evaluate the performance of the proposed algorithms. Further, in this section, to confirm the results obtained from the simulations we present testbed results. We conclude the chapter in Section 4.6.

4.2 Specifications

In this section, we formally define the problem of code dissemination. We then provide two refined specifications for solving the code dissemination problem: (i) deterministic code dissemination, and (ii) stabilising code dissemination.

4.2.1 Abstract Specification of Code Dissemination

Before we provide problem definitions, we introduce some notations we use in the rest of the chapter. We denote a code fragment by (π, v_π), with π being the code and v_π being the version number of the code. We denote by v'_π a possibly corrupted version number for code π, i.e., if $v'_\pi = v_\pi$, then the version number is not corrupted, corrupted otherwise. We say that a code has code fragment (π^n, v'_{π^n}) to mean that a node n has the code fragment (π, v_π) but has version number v'_π associated with it instead. Thus, unless stated otherwise, whenever we say a node n has code fragment π, we mean a node n has code fragment (π, v'_π).

We assume the version number to be a scalar quantity and that the version number can grow arbitrarily large. We also assume that the entire network use the same version number for the same code and that the version number is incremented by one as in [32, 81, 111], i.e., the version number of a new code should be one more than that of the old code. Thus, for a network to be consistent, there must be at most two version numbers in the network with a difference of 1. We assume only *detectable faults*, which rules out fault cases such as a fault that corrupts version numbers in a way that the difference of them is at most 1 (see Section 4.3.1 for more details on detectable faults).

Definition 1 (Consistent State in Code Dissemination) *Given a network $G = (V, A)$ and a code dissemination program Ψ for G, the state of a neighbourhood G' of G is said to be* consistent in *s, if there is at most 2 distinct version numbers, with a difference of at most 1, in that neighbourhood in s, where s is a state of Ψ. A process is said to have a* consistent state in *s if its code's version number is the same as that of at least one other neighbour process in s. A state of Ψ is* consistent in *s if all neighbourhoods of G are consistent in s.*

The above definition says that the state of a neighbourhood/node is consistent if the neighbourhood contains at most two version numbers, i.e., the version number of an old code and/or a new code. For example, if the version number of an old code is 0 then, for a neighbourhood to be consistent, in the neighbourhood there should be at most one more version number which is 1. In this case, for a node to have a consistent state, its version number should be 0 or 1. Similarly, the state of a program is consistent if all nodes in the network have either 0 or 1.

A node whose version number is neither 0 nor 1 is not consistent (if the version number of old and new codes were 0 and 1, respectively), and we call such a node *f-affected* node. A set of *f-affected* nodes is called *f-affected area.*

Definition 2 (F-affected node and F-affected area) *Given a network $G = (V, A)$ and a fault F that corrupts the program state, a process n is F-affected in a state if n will need to change its state to make the program state consistent. An area $G' = (V', A')$, with G' being a subgraph of G, is F-affected in a state s iff $\forall n \in V'$, n is F-affected in s.*

When a node changes its state to make the program state consistent, we say that the node *corrects its state*. As in most cases WSNs are deployed and unattended for a long time, it is preferable to have algorithms that correct program states themselves. A type of such algorithm is called a stabilizing algorithm.

Definition 3 (Stabilizing algorithm) *Given a network $G = (V, A)$, a problem specification Φ for G, and an algorithm Ψ. Algorithm Ψ is said to be* stabilizing *to Φ iff every computation of Ψ has a suffix which is a suffix of a computation of Φ that starts in an initial state.*

According to the definition above, in the context of code dissemination, an algorithm is called stabilizing if it can eventually correct the state of an *f-affected* node.

Stabilizing algorithms may need additional information, such as information about the network or neighbourhood, to work correctly. Due to stringent resource constraints of sensor nodes, stabilizing algorithms that need less auxiliary information from other nodes are more desirable. It may be the case that stabilizing algorithm may need more information than information about 1-hop neighbours. For example, instead of 1-hop neighbourhood information, it may need d-hop neighbourhood information. Thus, we define d-local stabilizing algorithm as follows:

Definition 4 (d-local stabilizing algorithm) *Given a network $G = (V, A)$, a problem specification Φ for G, and a stabilizing algorithm Ψ to Φ. Algorithm Ψ is said to be d-*local stabilizing *to Φ iff the cost of correcting the state of a node is bounded by functions of d.*

According to this definition an algorithm is called d-local stabilizing, if the complexity of correcting the state is $O(d)$. Here d can be any metric, including the number of transmitted messages, the number of nodes involved or the number of hops. For example, if the algorithm corrects the state of a node with information from nodes that are at most d-hop away, then the algorithm is called d-local stabilizing. It is ideal if the information a node needs to correct its state depends on itself. However, sometimes it may not be possible to correct the state using its own state information. Therefore, the node may need to involve other nodes in the network and acquire necessary information.

Definition 5 (Code Update) *Given two code fragments* (π, v_π) *and* (Π, v_Π)*,* Π *is said to be an* updated code over π *if* $v_\Pi > v_\pi$*. If a node* n *changes its code from* π *to* Π *and* Π *is a updated code over* π*, then we say that a* node n *updates* its code to Π*. Otherwise, if a node* n *changes its code from* Π *to* π*, then we say that a* node n *outdates* its code to π*.*

We will say that Π is an updated code to mean that Π is an updated over code π, whenever π is clear from the context. We will also say that a node n updates/outdates its code to Π/π if Π and π is obvious from the context.

An updated code Π is a new code for a network.

Definition 6 (New Code for G) *Given a code fragment* (Π, v_Π) *and a network* G*, we say that* Π is a new code for G *if all nodes in* G *have code fragment* π *and* $v_\Pi > v_\pi$*.*

Definition 7 (Version View) *Given a network* $G = (V, A)$*, the* version view *of a node* $n \in V$ *is a function that maps* n *to a sequence of multiset of values.*

Basically, the version view of a node n returns the distribution of version numbers in the network, i.e., the distribution of version numbers at every hop in the network centred on n. For example, if a node n has 3 1-hop neighbours with version numbers $1, 1$ and 2 and 5 2-hop neighbours with version numbers $2, 2, 1, 2$ and 3, then the version view of n is $\{1, 1, 1, 2, 2, 2, 2, 3\}$.

We now provide an abstract definition of code dissemination.

Definition 8 (Code Dissemination (CD)) *Given a network $G = (V, A)$, with a dedicated node called a sink $S \in V$, and an updated code (Π, v_Π) to be disseminated. Then, a code dissemination for Π is a sequence of sets of receivers $\langle R_0 \cdot R_1 \ldots R_{k_\Pi} \rangle$ such that*

1. $R_0^\Pi = \{S\}$
2. $\forall\ i, 0 \leq i \leq (k_\Pi - 1) : \forall r \in R_{i+1}^\Pi, \exists s \in R_i^\Pi \cdot (s, r) \in A$
3. $\bigcup_{0 \leq i \leq k_\Pi} R_i^\Pi = V$

Given a network G and an updated code Π, the code dissemination process starts with the sink (condition 1). Then, the code update process propagates forward (condition 2), i.e., all updated nodes forward the updated code to their neighbours, one hop at a time, until all the nodes have received the updated code (condition 3). The sequence represents the sequence in which the nodes updates their code.

In the above definition, we have made three assumptions: (i) when a code dissemination process starts, all the nodes have the same code base, i.e., they all have the same code, (ii) all nodes need to get the updated code (however, we can easily adapt the definition to the case where only a subset of nodes require the code update), and (iii) only one code dissemination can take place at a time, i.e., a code update can only occur once a previous one has completed. We call k_Π, the *dissemination latency* for Π. Observe that condition 2 *does not warrant that all the downstream neighbours* of an updated node to receive the code update in the next round. Due to issues such as message collisions and duty cycling, a downstream neighbour node may not receive the update in the next round, but sometime later from, possibly, another upstream neighbour.

4.2.2 Local Specifications for Code Dissemination

The specification given in Definition 8 is a global specification in the sense that it specifies the expected behaviour at the network level. In a distributed system, the verification that a program satisfies the global specification is challenging, given that global state is not instantaneously available. Thus, it is preferable to develop node-level specifications, which we call *local specifications*, which are more amenable to verification. The combination of local specifications (one for each process) result in the global specification.

We now present three increasingly weaker local specifications through which code dissemination could be achieved, which we call (i) strong code dissemination (CD), (ii) consistent CD and (iii) best effort CD. The first specification, strong CD, represents a gold standard and is satisfied by current dissemination protocols, such as [95, 79, 70]. The weaker specifications become important especially when transient faults occur in the network. We will define both specifications in terms of safety and liveness [8].

4.2.2.1 Strong Code Dissemination

Intuition In the fault-free case, a node n, having code π, will only download a code from a neighbor node, having code Π, if Π is an updated code. Further, n will not download Π again. Current code dissemination protocols also guarantee that, eventually, every node will download the updated code (even if some nodes are temporarily disconnected from the network, due to duty-cycling, link failures etc).

Thus, we define strong CD as follows (Definition 9):

Definition 9 (Strong CD) *Given a network $G = (V, A)$, a node $n \in V$ having a code fragment π, and a new code fragment (Π, v_Π) for G. Then,*

- Accuracy*: Node n will only change its code to an updated one.*

- Update: *Eventually node n will permanently update its code to* Π.

The liveness part of the specification, i.e., the update property, for strong code dissemination ensures that $\bigcup_{0 \leq i \leq k_\Pi} R_i^\Pi = V$ (see Definition 8). On the other hand, Definition 9, through the accuracy property, puts an additional constraint on code dissemination in that nodes only change their code with an updated one. In other words, $\bigcap_{0 \leq i \leq k_\Pi} R_i^\Pi = \emptyset$, i.e., no node updates more than once. In general, in the absence of faults, it can be expected that the system will satisfy the strong CD specification, e.g., [79, 95]. However, due to external factors, such as transient faults, nodes may wrongly outdate their codes and, if they do so, they will eventually have to correct these mistakes.

These wrong code changes (i.e., code outdates) give rise to various possible weaker specifications, namely (i) consistent CD and (ii) best effort CD.

4.2.2.2 Consistent Code Dissemination

Intuition When transient faults occur, it may be the case that a node n cannot distinguish between updated and outdated code. Specifically, a node n that has already updated may download the old code from a neighbour, i.e., n becomes outdated, if it believes that the neighbour has the code update. This is not an ideal situation. However, a node m that has yet to be updated may believe that another node m', with the same code as m, has the code update and may wrongly download the same (old) code. This specification forbids a node to outdate itself.

Definition 10 (Consistent CD) *Given a network $G = (V, A)$, a node $n \in V$ having a code fragment π, and a new code fragment (Π, v_Π) for G. Then,*

- No outdate: *Node n will never change its code to an outdated one.*
- Update: *Eventually node n will permanently update its code to* Π.

This means that, in consistent CD, if transient faults occur, before downloading the new code nodes in the network may download the same code they have, however, will not download the older code.

4.2.2.3 Best Effort Code Dissemination

Intuition The best effort CD specification allows for an updated node n to outdate itself as n may wrongly believe some node m to have the new code while n has the old code.

Definition 11 (Best Effort CD) *Given a network $G = (V, A)$, a node $n \in V$ having a code fragment π, and new code fragment (Π, v_Π) for G. Then,*

- Eventual accuracy*: Eventually, node n will only change its code to an updated one.*
- Liveness*: Eventually node n will permanently update its code from π to Π.*

An example is used to help better understand and differentiate between the three different specifications proposed. At the start of the dissemination of the new code Π, the old code π resides at every node in the network. Assume that the old code has version 0 and the new code has version 1. Hence, for any node n in the network, there are four possible code transitions in the presence of transient faults:

1. $0 \to 0$: Node n has the old code and changes to the old code again - Redundant
2. $0 \to 1$: Node n has the old code and changes to the new code - **Code update**
3. $1 \to 0$: Node n has the new code and changes to the old code - **Code outdate**
4. $1 \to 1$: Node n has the new code and changes to the new code again - Redundant

The strong CD specification only allows the second type of code transition. The consistent CD specification allows the 1, 2 and 4 types of code transitions while the best effort CD specification allows all of them. It can be observed then that BestEffort specification allows more redundant downloads than either of the other two specifications.

4.2.3 Fault Tolerance Issues: An Overview

It can happen that a code dissemination protocol that has been proved correct (i.e., satisfies its strong specification in the absence of faults), violates its specification in the presence of faults due to it not being able to handle faults [12], i.e., the code dissemination protocol is fault-intolerant. As such, there is a variety of fault tolerance properties that the program can satisfy, viz. fail-safe fault tolerance, non-masking fault tolerance and masking fault tolerance [12]. A fail-safe fault-tolerant program guarantees that safety will always be satisfied, while a non-masking fault-tolerant program guarantees that liveness will eventually be satisfied, even if safety can be temporarily violated. On the other hand, a masking fault-tolerant program (the gold standard) guarantees that the program will satisfy its specification even in the presence of faults. To transform a fault-intolerant program into a fail-safe fault-tolerant (resp. non-masking fault-tolerant) program, addition of program components called detectors (resp. correctors) to the fault-intolerant program are both necessary and sufficient. Thus, to make a program masking fault-tolerant, it is necessary and sufficient to add both detectors and correctors [12]. A detector component is one that asserts the validity of a predicate in a running program, while a corrector component enforces a predicate on a running program.

In networking, a non-masking fault-tolerant program is generally suitable as it guarantees that, eventually (i.e., when faults stop), the program will satisfy its specification again. Though non-masking fault tolerance entails the erroneous downloads of codes, it is the one more suited to code dissemination as masking fault tolerance is very expensive, both spatially and temporally, to guarantee. Given the existence of several code dissemination algorithms, it is not intended, in this work, to develop another (non-masking) fault-tolerant code dissemination protocol. The thrust is to *develop a generic corrector protocol* that, when added to a fault-intolerant code dissemination protocol, will enable the resulting protocol to satisfy the liveness specification (i.e., eventually all nodes will permanently update with the updated code). We also require that the corrector protocol is only executed when an erroneous state is detected.

At this point, we need to define the properties of such a corrector protocol that will capture its correctness. Since we wish this corrector protocol to be generic, and work as a wrapper (i.e., it can plug in with various code dissemination protocols), it cannot be based on any specific code dissemination protocol implementation. Rather, the working of the corrector protocol should only be based on the specification of the code dissemination protocol, more specifically its interface and specification. Such an approach is what has been termed as graybox stabilization [10].

Definition 12 (Corrector Component for Code Dissemination) *Given a strong code dissemination specification σ^s (Definition 9) and a weaker version σ^w (Definitions 10 or 11), some transient fault model F, a protocol Σ that satisfies σ^s in the absence of F but violates σ^s in the presence of F, and a program ϕ. Then, ϕ is a* σ^w-corrector program *for σ^s iff*

- Transparency: *In the absence of F, $(\Sigma \circ \phi)$ satisfies σ^s.*
- Stabilizing: *In the presence of F, $(\Sigma \circ \phi)$ satisfies σ^w.*

If σ^w is consistent CD (resp. best effort CD), then ϕ is a consistent (resp. best effort) corrector for strong CD.

Here, $A \circ B$ represents the addition of program A with program B [12]. The set of computations of the composite system $(A \circ B)$ is the smallest fusion-closed set that contains computations of A and B, with the initial states being the set of common initial states of A and B. Definition 12 stipulates that, when there is no transient fault in the network, the corrector program is transparent, i.e., it does not interfere with the working of the code dissemination protocol and satisfies the strong code dissemination. However, when transient faults are occurring, then the corrector program will help the code dissemination protocol to eventually guarantee that a node will permanently download the updated code, after possibly having downloaded stale code.

4.3 Theoretical Results

In this section, we show that (i) it is impossible to solve the strong code dissemination problem in the presence of transient faults, and (ii) there exists no 1-local algorithm to solve the strong code dissemination problem in the presence of a stronger class of transient faults, which we term as *detectable faults.*

4.3.1 Strong Code Dissemination in the Presence of Transient Faults

In this section, we investigate the possibility of developing an algorithm that solves the strong code dissemination problem in the presence of transient faults. Ideally, even in the presence of transient faults, it would be beneficial if a node only updates with new code to prevent redundant downloads, thereby saving energy.

Intuition From the specification of strong code dissemination (Definition 9), it is stated that nodes only update their codes when they are in the presence of a newer code fragment. However, when transient faults occur, the version number that is advertised by or stored at a node can be corrupted, possibly leading to nodes downloading old code fragments. Thus, the first main contribution of the chapter is captured by Theorem 1, which states that it is impossible to solve the strong code dissemination in the presence of transient faults.

Theorem 1 (Impossibility of strong CD) *Given a network $G = (V, A)$, a fault model F that corrupts the program state, and an updated code fragment (Π, v_Π). Then, there exists no deterministic algorithm that solves the strong code dissemination problem for Π in G in the presence of F.*

Proof. Consider the network G, and assume a deterministic algorithm Ψ that solves the strong code dissemination problem. We will construct an appropriate state and show that, under Ψ, a node may wrongly update, hence a contradiction.

Assumptions: Two nodes n_i and n_j where n_i (resp. n_j) is both an upstream and downstream neighbour of n_j (resp. n_i).

Consider a fault free computation $C = s_0 \cdot s_1 \ldots$ of Ψ. In a given state s_k in C, two nodes $n_i, n_j \in V \setminus \{S\}$ have the following code fragments: n_i has Π and n_j has π.

Now, nodes n_i and n_j interact such that n_i and n_j inform each other of their respective code fragments, i.e., about their respective version numbers. Given that Ψ solves the strong code dissemination problem, node n_j will eventually permanently update its code with Π in a state $s_l, l > k$ as Π is an updated code fragment.

Now, consider a faulty computation $C' = s'_0 \cdot s'_1 \ldots$ of Ψ, and a state s'_k which is exactly the same as s_k (above) except for the following: (i) node n_i has a code fragment $(\Pi_{n_i}, v'_{\Pi_{n_i}})$ and node n_j has code fragment $(\pi_{n_j}, v'_{\pi_{n_j}})$. In a state $s'_l, l > k$ of C', assume that n_i has the same version view as n_j in s_l and n_j has the same version view as n_i in s_l.

Since Ψ is deterministic and solves strong CD, node n_i will permanently update its code with π in s'_l, which is a contradiction as π is an old code. Hence, no such deterministic Ψ exists.

The impossibility is underpinned by some major problems, the most prominent being: (i) Nodes are not able to detect unexpected version numbers as, for example, if nodes with the stale code have their respective version numbers corrupted to very high values, old code may propagate through out the network, and (ii) nodes with the updated code may have their respective version numbers corrupted to that of the old code, while nodes with the old code have their respective version numbers corrupted to the new one, i.e., all updated nodes appear as outdated and all outdated nodes appear as updated.

To attempt to circumvent this impossibility of Theorem 1, there are different possible avenues. For example, one may allow algorithms to make a finite number of mistakes, thereby solving a weaker problem specification (such as the weak code dissemination). Another example might be to solve the strong code dissemination problem in the presence of a *stronger* fault model, i.e., a fault model where the set of possible corruptions is constrained. For the first possibility, if the possibly few updated nodes are overwritten with the old code, then the part of the network may not get the new code. This case is illustrated in Figure 4.1. For example, if a node C, that is three hops away from the sink, gets corrupted and the version number has been changed from 2 to 5, then all nodes may start to download that code, as $5 > 2$. When a node that is a neighbour of the sink, say node A, downloads the code with version number 5, node A may correct itself by downloading from the sink the code with version number 2 (or 3 if it is the new code). However, node B will not get the correct code as its version number 5. So, all nodes that are two or more hops away from the sink may not get the code with version number 2 (or 3, respectively).

Thus, even if a finite number of mistakes are allowed (i.e., download of old code), then there is no guarantee of the entire network getting the correct code. Therefore, in this chapter, we follow the second possibility, i.e., we assume a stronger fault

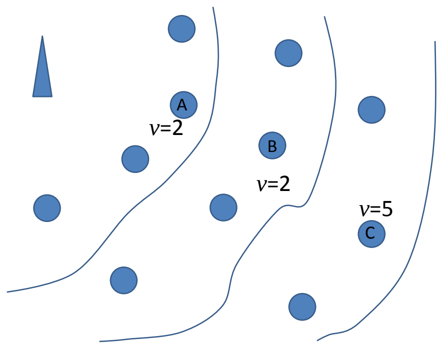


Figure 4.1: Current code version number is 2. The version number of node C is corrupted to 5.

model.

Thus, the assumed fault model needs to be such that the two stated problems are handled: *we require the faults to result in detectable errors.* Thus, we rule out a few fault actions: (i) we require that the fault model does neither make outdated nodes appear as updated and updated as outdated, (ii) nodes in a neighbourhood need to be corrupted differently (so nodes in a neighbourhood do not appear as outdated/updated). We call the resulting fault model as the *detectable* fault model, which we assume in the rest of the chapter. An example of a possible fault ruled out by the first constraint can be illustrated with an example: assume the old code version is 1 and the new version is 2, and the version number increases by 1 for each new update. An updated (resp. outdated) node cannot have its version number to be corrupted to 1 (resp. 2). Also, say a node n has its version number corrupted to 5, then a node m in n's neighbourhood cannot have its version number corrupted to 6, as m will appear as updated to n.

Though these faults can actually occur in practice, the probability of such faults to occur is very low. If these faults do occur, then it is impossible to guarantee that dissemination of the new code will terminate properly. One way to circumvent this problem is to require the sink to query the network after some time to determine if nodes have the proper code, using possibly a hash of the code and the version number.

In the presence of detectable faults, a trivial solution to solve the strong code dissemination is to require the sink to periodically start the dissemination process. Whenever a node encounters a version that is unexpected (allowed under the detectable fault model), it does not need to download the code associated with it. When it sees a version number that is expected (i.e., realistic), and since the code associated with the version number cannot be a stale one (as it is ruled out by the fault model), the node can download the code. Unfortunately, such a scheme is expensive as the network will need to spend lots of energy for dissemination, i.e., the protocol is a global one. Thus, we seek to determine whether nodes can rely only on its 1-hop neighbourhood for code dissemination (just as in a fault-free case) in the presence of detectable faults. This is captured in Theorem 2.

Theorem 2 (Impossibility of 1-local strong CD) *Given a network $G = (V, A)$, a detectable fault model F, and an updated code fragment Π. Then, there exists no* 1*-local algorithm that solves the strong code dissemination problem for* Π *in* G *in the presence of* F*.*

Proof. The proof is trivial. If the 1-hop neighbourhood of a node is corrupted in such a way that the version numbers are either unexpected ones or old ones, then the node will not download any code. Hence, a 1-local protocol is not possible.

Intuitively, if a neighbourhood is corrupted by a detectable fault model, then nodes will need to start downloading from uncorrupted nodes outside of the corrupted neighbourhood. Hence, this points towards a f-local algorithm, where f is the diameter of the affected area, that can solve the strong code dissemination algorithm. However, given the nature of WSNs and of the code dissemination process, strong code dissemination in the presence of faults is not appropriate, due to the overhead it induces on the network. To this end, we focus on the two other specifications, viz. best effort CD and consistent CD. In the next section, we present two corrector programs that, when added to a fault-intolerant code dissemination protocol, solve the BestEffort CD and Consistent CD problems.

4.4 Code Dissemination Correction: Two Generic Corrector Protocols

In this section, we present two generic *corrector* programs, namely (i) *BestEffort-Repair* and (ii) *Consistent-Repair*. Each of the two protocols can be added to a fault-intolerant code dissemination protocol to make the code dissemination protocol satisfy some correctness specification.

As stated before, rather than developing a single (non-masking) fault-tolerant code dissemination protocol, the focus is on transforming existing fault-intolerant code dissemination protocols into non-masking fault-tolerant ones. To enable this, we adopt the technique for graybox stabilisation [10] whereby, rather than developing a corrector for a particular code dissemination protocol, a (generic) corrector protocol is designed based on a specification. This corrector can then be added to any implementation that satisfies the specification, resulting in the eventual program to be non-masking fault-tolerant. In that way, the corrector is reusable.

Further, from the definition of a corrector component (Definition 12), the corrector should be transparent to the code dissemination protocol when there are no faults in the network, i.e., the behaviour of the composite corrector and dissemination protocol should be identical to that of the dissemination protocol alone in the absence of faults. To achieve this, we include a detector component in the code dissemination protocol that, when satisfied (during faulty periods), triggers the corrector component. It would be advantageous to then be able to develop a detector based on a specification. However, in such a case, the efficiency of the detector is not very high, in that it can suffer from high false positives or false negatives, which can then cause the corrector to violate its transparency property [54]. What this means is that, in general, a generic detector would miss some errors, leading to erroneous downloads of code or, in the worst case, the whole network having the old code. For instance, corruption of variables other than the version numbers can still lead to the code dissemination protocols not working properly (e.g., in Varuna, corrupting the neighbourhood table caused problems). To compensate, we design the detectors based on protocol implementations, i.e., a detector is needed for each different code dissemination protocol. Overall, our approach is to develop a protocol-specific detector which, when its corresponding detection predicate becomes true, triggers the execution of the generic corrector program, making the code dissemination protocol non-masking fault-tolerant.

A design methodology suggested for graybox stabilisation is to design a program that contains two different components [10]: (i) a process-specific component and (ii) an interprocess-specific component. The process-specific component is responsible for making the state of a single process consistent, whereas an interprocess-specific component is responsible for correcting any inconsistency between different processes. In a fault-intolerant code dissemination protocol, since the only relevant information nodes keep about the code is the version number, then state inconsistency at the process level is irrelevant, i.e., the state of a single process is trivially consistent. On the other hand, state inconsistency can be detected when comparing the version numbers of two different processes. Thus, interprocess-specific component of a corrector program only needs to correct the states of processes that are inconsistent with each other.

In Sections 4.4.1 and 4.4.2, we present two corrector protocols that correct any state inconsistency between processes, transforming the fault-intolerant code dissemination protocols in non-masking fault-tolerant ones. The BestEffort-Repair protocol, as the name suggests, attempts to correct the state inconsistencies as fast as possible, while the Consistent-Repair protocol attempts to correct the state consistencies as intelligently as possible. In other words, the worst case scenario for BestEffort-Repair may be worse than that of the Consistent-Repair but the best case scenario for BestEffort-Repair is also better than that of Consistent-Repair.

4.4.1 The ***BestEffort-Repair*** Protocol

Before describing the *BestEffort-Repair* protocol and giving its formal description, we present the main idea behind it, and the special packets it uses. Since 1-local fault tolerance is not possible (Theorem 2), the main idea is to correct (i.e., repair) the protocol state as fast as possible. Correcting a state inconsistency (i.e., error) quickly means that the error does not propagate through out the whole network.

BestEffort-Repair uses six special types of data packets (we call them BestEffort-Repair packets), which we describe below.

- **Prob**: It contains the code's version number and it is used to ask a neighbouring node to correct an error.
- **Check**: A node sends a Check packet to request the current version number of neighbouring nodes.
- **Rep**: A node sends a Rep packet in response to a Check packet and it contains the node's (stored) version number.
- **OK**: It is used to release some nodes from the correction process.
- **Cor**: A node sends a Cor packet to inform other neighbouring nodes about the correct version number.

- **Hello**: A node sends a Hello packet to inform other neighbouring nodes about the correct version number and also that it has the updated code.

Informally, BestEffort-Repair works as follows: When a node n_1 detects an error after communicating with a node n_2, it attempts to correct the erroneous state. A Prob packet is sent by n_1 to n_2 to indicate a problem, asking n_2 to correct the problem. If the error cannot be corrected by n_2, i.e., the version numbers of the neighbours of n_2 are not same, then Check packets are broadcast, creating a *correction tree* (see Figure 4.2), rooted at the node (n_1) that detected the error. The leaf nodes of the tree responds to Check packets by sending Rep packets. If n_2 detects an error with any of the leaf nodes, it will spawn a subtree, within the main correction tree. Once a region in the network is reached where no fault has occurred, i.e., outside of the fault-affected area (see Figure 4.2), then no more subtree is spawned. This means that a node's, say n_l, neighbourhood (i.e., all the children of the node within the correction tree) have the same code version, as the version is correct (under the detectable fault model). In other words, n_l has received Rep packets from its children with the same version number. Then, ultimately, the node n_l responds through a Hello or Cor packet, and its subtree "disappears". Any node sending a Hello or Cor packet will cause its subtree to "disappear" since the node has ascertained the correct version number (and in the case of Hello packet, n_l also notified its parent about the availability of the code as well).

4.4.1.1 BestEffort-Repair: An Overview

When a node n_1 detects an error (which is protocol-specific) after receiving a message from a neighbouring node n_2, n_1 sends a Prob packet to n_2, thereby asking n_2 to check whether it is the source of the error (we will shortly explain what happens if n_2 does not receive the Prob packet from n_1). Node n_1 then goes to the *Wait* state, where n_1 will wait for some predefined time. In turn, n_2 asks its neighbouring nodes,

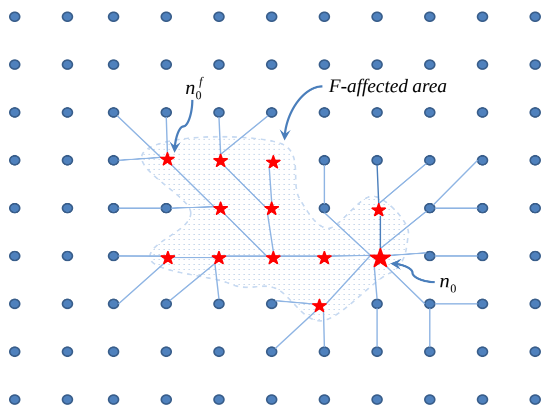


Figure 4.2: An example of a fault-affected area and correction tree. Star nodes are faulty nodes

except n_1, for their version numbers by broadcasting a Check packet. Node n_2 then goes to the *Wait_Rep* state where it will wait for Rep packets from its neighbours over a certain time interval. All nodes that receive the Check packet from n_2 send a Rep packet to n_2. Now, node n_2 will compare all the received version numbers obtained from the Rep packets. If the version numbers are equal and match its own version number, then n_2 sends a Hello packet to n_1. By sending a Hello packet, node n_2 says to node n_1 that it has the correct version and it has the associated code too.

Now, if the received version numbers from the Rep packets are the same but differ from that of n_2, node n_2 will send a Cor packet to n_1 and corrects its code fragment by downloading from one of the Rep senders. By sending a Cor packet to n_1, node n_2 tells n_1 about the currently available version, which n_1 can download from another node with the associated code. If at least one of the received Rep packets contains a different version number i.e., the received version numbers are not identical, then n_2 goes to the *Wait* state and broadcasts a Prob packet, as done by n_1 earlier. This process continues until a node that sent a Prob packet will get a Hello or Cor packet. Nodes that were in the *Wait* or *Wait_Rep* state after updating their code fragments, go to the *Temp* state where they broadcast a Hello packet a few times.

Because of reasons such as transient link failures, packets may not be delivered, for example, a Prob packet sent by n_1 may not be delivered to n_2. To overcome this issue n_1 periodically sends a Prob packet to n_2 some predefined times until n_1 receives an implicit acknowledgement packet like Check, OK or Prob from n_2 or Hello or Cor from any node. To address the same problem, Hello packets are sent more than once. In general, if a node n in a state s cannot send or receive packets due to transient link failures, then n waits in s until the corresponding timer expires and may restart the procedure. For example, if a node n after detecting an error and going to the *Wait* state cannot correct its version number, n waits until the corresponding timer expires and goes to the *Stable* state. After some time, n or other node may detect the error and start the correction procedure again.

4.4.1.2 BestEffort-Repair: Formal Protocol Description

The variables and code for the *BestEffort-Repair* algorithm is shown in Figures 4.4 and 4.6.

Figure 4.3 illustrates the state machine of BestEffort-Repair, which we now detail.

- **Stable (PS) state:**
  - If a node n_1 detects a fault after receiving an H^3 packet from a node n_2, it goes to the *Wait* state.
  - If a node n_1 receives a *Check* packet from a node n_2, after waiting a random time between [0, *SendRep*], it sends a *Rep* packet to n_2.
  - If a node n_1 receives a *Prob* packet from a node n_2 and if n_1 is the destination node, it waits *SendProb*+t time to receive other possible *Prob*

[3] Application layer packet or code update maintenance packet

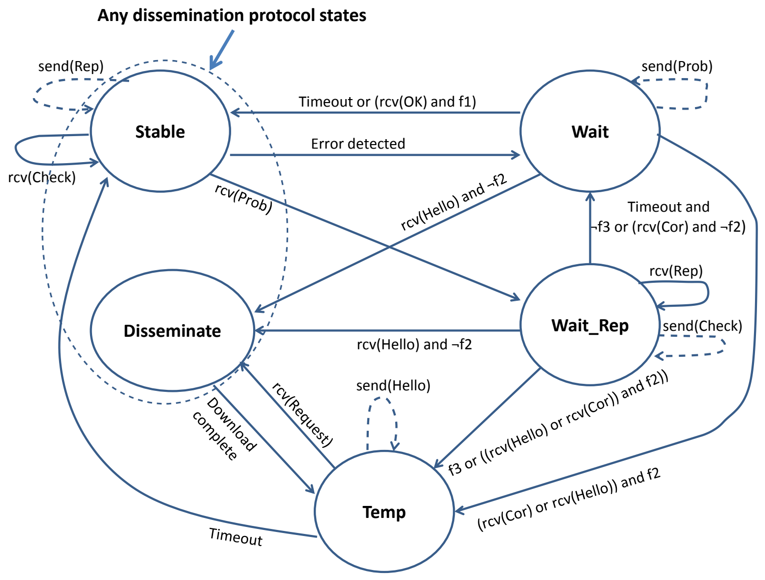


Figure 4.3: The state machine for BestEffort-Repair. Two states in dashed area are the states of any dissemination protocol. f1=TRUE if sender of *OK* packet is the node which sent *H*, f2=TRUE if Sender.Vers=Receiver.Vers, f3=TRUE if all received metadata are the same.

Variables of process i:

PacketType $\in \{H, Prob, Check, Hello, OK, Rep\}$
% H packets are application layer(data)/code update maintenance packets.
% The other packets are Repair packets.

state $\in \{Stable, Wait, Wait\_Rep, Temp, Disseminate\}$ **Init** state =Stable;
h, p, version, countH, countP : $\mathbb{N}$
Init countH:= 0, countP:=0
firstProb $\in \{0, 1\}$ **Init** firstProb = 1

TableProb, TableRep:{(id, version): id $\in \mathbb{N}$, version $\in \mathbb{N}$}
% Keep track of nodes and version number they sent/receive

% Protocol Timers

PeriodProb: Timer
SendProb: Timer
Wait_Time: Timer
SendRep: Timer
WaitRep_Time: Timer
SendCheck: Timer
Temp_Time: Timer
SendHello: Timer
t: Timer
U: Timer % Parameter from application layer for periodic traffic.

Figure 4.4: Variables of BestEffort-Repair algorithm.

```
state==Stable
1  case(upon⟨rcv ($H, n, i$)⟩ and detect())
2    state:=Wait
3    setTimer($Prob$, SendProb, $n$)
4    setTimer($WAIT$, Wait_Time)
5  case(upon⟨rcv ($Check$, $n$, $i$)⟩)
6    setTimer($Rep$, SendRep, $n$)
7  case(upon⟨rcv ($Prob$, $n$, $i$)⟩)
8    if(firstProb==1)
9      firstProb:=0
10     setTimer(Timer, SendProb+t)
11   else
12     TableProb ∪ ($n$, $n$.version)
13   endif
14 case(upon⟨rcv ($Prob$, $n$, $ALL$)⟩)
15     if($i$.version!=$n$.version)
16       state:=Wait_Rep
17       setTimer($WAITREP$, WaitRep_Time)
18       setTimer($Check$, SendCheck, $ALL$)
19     endif
20 endcase

21 if(timeout(Timer))
22   compare all $n_j$.versions ∈ TableProb
23   if(all are equal)
24     bCast($OK$, $i$, $ALL$)
25     state:=Temp
26     setTimer($TEMP$, Temp_Time)
27   else
28     state:=Wait_Rep
29     setTimer($WAITREP$, WaitRep_Time)
30     setTimer($Check$, SendCheck, $ALL$)
31   endif
32 endif
```

```
state==Wait
1  repeat every $PeriodProb$
2    send($Prob$, $i$, $n$)
3    countP:=countP+1
4  until (rcv.type!=$H$ or countP > p)
5  case(upon⟨rcv ($OK$, $n$, $i$)⟩)
6    state:=Stable
7    stopAllTimers()
8    TableProb:=∅
9    TableRep:=∅
10 case(upon⟨rcv ($Cor$, $n$, $i$)⟩)
11   if($i$.version==$n$.version)
12     state:=Temp
13     setTimer($TEMP$,Temp_Time)
14   else
15     bCast($Cor$, $i$, $ALL$)
16   endif
17 case(upon⟨rcv ($Hello$, $n$, $i$)⟩)
18   if($i$.version==$n$.version)
19     state:=Temp
20     setTimer($TEMP$,Temp_Time)
21   else
22     state:=Disseminate
23   endif
24 endcase

25 if(timeout($WAIT$))
26   state:=Stable
27   stopAllTimers()
28   TableProb:=∅; TableRep:=∅
29 endif
```

```
state==Wait_Rep
1  case(upon⟨rcv ($Cor$, $n$, $i$)⟩)
2    if($i$.version==$n$.version)
3      state:=Temp
4      setTimer($TEMP$, Temp_Time)
5    else
6      bCast($Cor$, $i$, $ALL$)
7      state:=Wait
8      setTimer($WAIT$, Wait_Time)
9    endif
10 case(upon⟨rcv ($Hello$, $n$, $i$)⟩)
11   if($i$.version==$n$.version)
12     state:=Temp
13     setTimer($TEMP$, Temp_Time)
14   else
15     state:=Disseminate
16   endif
17 case(upon⟨rcv ($Rep$, $n$, $i$)⟩)
18   TableRep:= TableRep ∪ {($n$, $n$.version)}
19 endcase

20 if(timeout($WAITREP$))
21   compare all $n_j$.versions ∈ TableRep
22   if(all and $i$.version are equal)
23     send($Hello$, $i$, $n$)
24     state:=Wait
25     setTimer($TEMP$, Temp_Time)
26   elsif(all are equal and $i$.version is not equal)
27     state:=Disseminate
28   else
29     bCast($Prob$, $i$, $ALL$)
30     state:=Wait
31     setTimer($WAIT$, Wait_Time)
32   endif
33 endif
```

Figure 4.5: BestEffort-Repair Algorithm.

```
state==Temp
repeat every SendHello
  bCast(Hello, i, ALL)
  countH:=countH+1
until count>h
if(countH>h)
  state:=Stable
  stopAllTimers()
  TableProb:=∅
  TableRep:=∅
endif
case(upon⟨rcv (Code Request, n, i)⟩)
  state:=Disseminate
  countH:=0
endcase
if(timeout(TEMP))
  state:=Stable
  stopAllTimers()
  TableProb:=∅; TableRep:=∅
endif
```

```
state==Disseminate
send/download the code.
state:=Temp
setTimer(TEMP,Temp_Time)

if(timeout(PacketType, j))
  if(j==ALL)
    bCast(PacketType, i, ALL)
  else
    send(PacketType, i, j)
  endif
endif
```

Figure 4.6: BestEffort-Repair Algorithm.

packets destined to n_1, and compares the received metadata. If they are equal n_1 broadcasts *OK* packet, requests the new code fragment from one of the Prob senders and goes to the *Disseminate* state. If at least one of the received metadata is not equal, then n_1 goes to the *Wait_Rep* state.

- If a node n_1 receives a *Prob* packet and it is not destination node, then n_1 compares its metadata with the received one, if they are not equal n_1 goes to the *Wait_Rep* state.

- **Wait state:** Let n_1 be the node which transitions to the *Wait* state after receiving an H packet from a node n_2 and after detecting the fault.
  - After waiting a random time between [0, *SendProb*], n_1 sends a *Prob* packet to n_2.
  - If n_1 receives an *OK* packet from n_2, it goes to the *Stable* state.
  - Every *PeriodProb* time n_1 sends a *Prob* packet to n_2 upto p times unless it receives *OK*, *Check*, *Prob* or *Hello* packet.

  - When n_1 receives a *Cor* packet, it compares the received metadata with its own. If they are the same, then n_1 goes to the *Temp* state. If they are different, then n_1 broadcasts a *Cor* packet.
  - If n_1 receives a *Hello* packet from a node n_3 and its metadata is the same with the metadata of n_3, then n_1 goes to the *Temp* state. If metadata are different, n_1 requests the correct code fragment from n_3 and goes to the *Disseminate* state to update its code fragment.
  - If n_1 does not receive a *Hello* packet during the *Wait_Time*, it goes to the *Stable* state.

- **Wait_Rep state:** Let n_1 be the node which transitions to the *Wait_Rep* state after receiving a *Prob* packet from a node n_2.
  - After waiting a random time between [0, *SendCheck*], n_1 broadcasts a *Check* packet.
  - After waiting for *WaitRep_Time*, the node n_1 compares all metadata received in *Rep* packets from the neighbours. If all of them are the same, and if the metadata of n_1 is equal to the received ones, n_1 sends *Hello* to n_2 and goes to the *Temp* state. If all of them are equal, but metadata of n_1 is not equal to the received ones, n_1 requests the correct code fragment from one of the *Rep* senders and goes to the *Disseminate* state. If at least one of the received metadata is different from the other received metadata, n_1 broadcasts a *Prob* packet and goes to the *Wait* state.
  - If n_1 receives a *Cor* packet and its metadata is the same with the received metadata, n_1 goes to the *Temp* state. If the metadata are different, n_1 broadcasts a *Cor* and goes to the *Wait* state.
  - If n_1 receives a *Hello* packet from a node n_3 and its metadata is the same with metadata of n_3, n_1 goes to the *Temp* state. If they are different,

n_1 requests the correct code fragment from n_3 and goes to *Disseminate* state.

- **Temp state:**
  - After entering this state a node n_1 broadcasts a *Hello* packet every *Send-Hello* seconds upto h times during *Temp_Time*.
  - If during *Temp_Time* n_1 does not receive any request message, n_1 goes to the *Stable* state.
  - If a node n_1 receives request message it goes to the *Disseminate* state to send the requested code fragment.
- **Disseminate state:** After sending or downloading requested code, a node goes to the *Temp* state.

The idea of the algorithm is explained through the example illustrated in Figure 4.7. When node A receives a packet from node B and detects an error, it sends a *Prob* packet to node B and goes to the *Wait* state 4.7(a). Node B broadcasts a *Check* packet to its neighbours to ask their version numbers and goes to the *Wait_Rep* state 4.7(b). After waiting a random time, the neighbours send *Rep* packets to node B 4.7(c). Notice that, as node A is in the *Wait* state, it does not reply to node B. Then, node B compares if all the version numbers in the *Rep* packets are the same. If the version numbers are the same, node B sends a *Hello* packet to node A and goes to the *Temp* state, where it broadcasts *Hello* packet several times 4.7(d). Otherwise, node B broadcasts a *Prob* packet and goes to the *Wait* state 4.7(e). The same procedure will repeat until a node receives *Rep* packets from its neighbors that contain the same version numbers 4.7(f), 4.7(g), 4.7(h). Then, when a node receives a *Hello* packet, it sends a *Hello* packet to the node from which it has received a *Prob* packet 4.7(i).

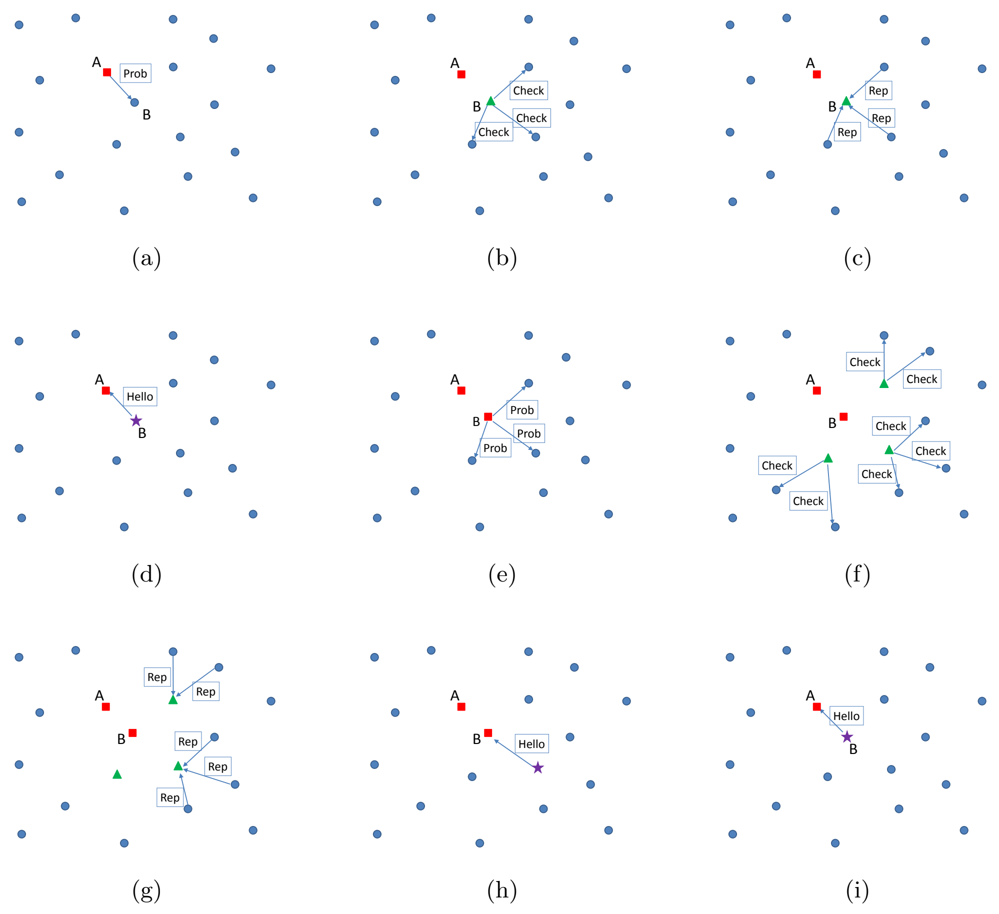


Figure 4.7: The Best-Effort algorithm. Square nodes are in Wait state, triangular nodes are in Wait_Rep state, star nodes are in Temp state, circular nodes are in Stable state.

We now prove that BestEffort-Repair is a corrector component.

Lemma 1 (Containment of BestEffort-Repair) *Given a network $G = (V, A)$, detectable fault model F, an F-affected area $G' = (V', A')$, then, at most $O(|V'|)$ nodes will download the old code.*

Proof From BestEffort-Repair, a node n, after sending Check packets to its neighbours - due to receiving a Prob packet from a node n', waits for Rep packets. If the received Rep packets are all identical in their version numbers, then n will either broadcast a Cor packet (stating the expected correct version number and then download the code) or it will broadcast a Hello packet. If the version number is the old one, then, n will download the old code. All other nodes that receive the Hello packet will also download the old code. Thus, at most, all nodes in $m \in G'$ and all nodes $p \in M_u$ will receive a Hello packet with the version number being the old one.

From Lemma 1, it can be observed that only a finite number of nodes, including updated ones, will change their code to the old one in presence of transient faults. Since there will then be the old code and the new code in the network, eventually, the code dissemination protocol will ensure that all nodes get the updated code. This is captured in Theorem 3

Theorem 3 (Correctness of BestEffort-Repair) *Given a network $G = (V, A)$, detectable fault model F, a strong code dissemination specification σ for G, Best-Effort CD σ^b, a protocol Σ that satisfies σ in the absence of F but violates σ in the presence of F. Then, BestEffort-Repair is a* BestEffort-corrector component *for strong CD.*

Proof. For the transparency property, since BestEffort-Repair is only triggered when there is an error in the network, then safety is satisfied by the correctness of Σ. The stabilizing property follows from Lemma 1 and the nodes will the old code with ultimately download the correct code due to Σ.

4.4.2 The *Consistent-Repair* Protocol

4.4.2.1 Consistent-Repair: An Overview

Consistent-Repair works in a similar way to BestEffort-Repair, with differences when a node changes its code.

When a node n_1 detects an error (which is protocol-specific) after receiving a message from a neighbouring node n_2, n_1 sends a Prob packet to n_2, thereby asking

n_2 to check whether it is the source of the error (we will shortly explain what happens if n_2 does not receive the Prob packet from n_1). Node n_1 then goes to the Wait state, where n_1 will wait for some predefined time. In turn, n_2 asks its neighbouring nodes, except n_1, for their version numbers by broadcasting a Check packet. Node n_2 then goes to the Wait_Rep state, where it will wait for Rep packets from its neighbours over a certain time interval. All nodes that receive the Check packet from n_2 send a Rep packet to n_2. Now, node n_2 will compare all the received version numbers obtained from the Rep packets.

If all the received version numbers are equal and match its own version number, then n_2 sends a Hello packet to n_1, stating that the correct version number is that held by n_2 and that it has the updated code too. If the received version numbers from the Rep packets are the same but differ from that of n_2, n_2 downloads the available code fragment by downloading from one of the Rep senders. After downloading the code, n_2 will eventually send Hello messages. Further, if there are only two different version numbers received, then n_2 chooses the higher one (as there are only two versions in the network during the dissemination process).

On the other hand, if n_2 obtains more than 2 version numbers, then this indicate an error in the network. For any node n_3 that sent Rep packets to n_2 with version numbers that violate the consistency predicate, n_2 send Prob packets to n_3. These nodes, in turn, send check packets to their neighbours and the process is repeated.

Once the recovery process reaches a region outside of the fault-affected area (an example of a fault-affected area is depicted in Figure 4.2), nodes on the border of the fault-affected area will receive Rep packets with at most two different version numbers (this is the case when only part of a neighbourhood has been updated). Once a node receiving these Rep packets decides on the correct version number, it broadcasts a Hello message (after possibly downloading the associated code, if it does not already have it) a few times to ensure that its neighborhood learns about the correct version.

Because of reasons such as transient link failures, a Prob packet sent by n_1 may not be received by n_2. To overcome this issue, n_1 periodically sends a Prob packet to n_2 for some predefined times until n_1 receives an implicit acknowledgement packet, such as Hello or Check.

4.4.2.2 Consistent-Repair: Formal Protocol Description

The Consistent-Repair protocol, when added to a fault-intolerant code dissemination protocols, transforms the protocol into a non-masking fault-tolerant one that satisfies the Consistent CD specification. It leverages the fact that, at any time during the new code dissemination, there will be at most two codes in the network: (i) the old one and (ii) the new one. This means that any node only need to know about two different version numbers. Once a node knows about these, it can choose the higher one, which is associated with the updated code.

Figure 4.8 illustrates the state machine of the Consistent-Repair protocol, which we now detail. The protocol is shown in Figure 4.9. The code for when the process in in state 2 (Wait) and state 3 (Wait_Rep) is shown. The code for when the process is in state 1, 4 and 5 is the same as for BestEffort-Repair (Figure 4.6)

There are three main states in Consistent-Repair:

- **Wait state:** Let n_1 be the node which transitions to the *Wait* state after receiving an H packet from a node n_2 and after detecting an error.
  - If n_1 receives a *Hello* packet first time, say from a node n_3, it stores n_3 and its version number. Whenever n_1 receives a *Hello* packet which contains a different version number than the first one stored (version number of n_3), n_1 compares the two received version numbers and its own metadata. If the version number of n_1 is equal to the bigger of the two received

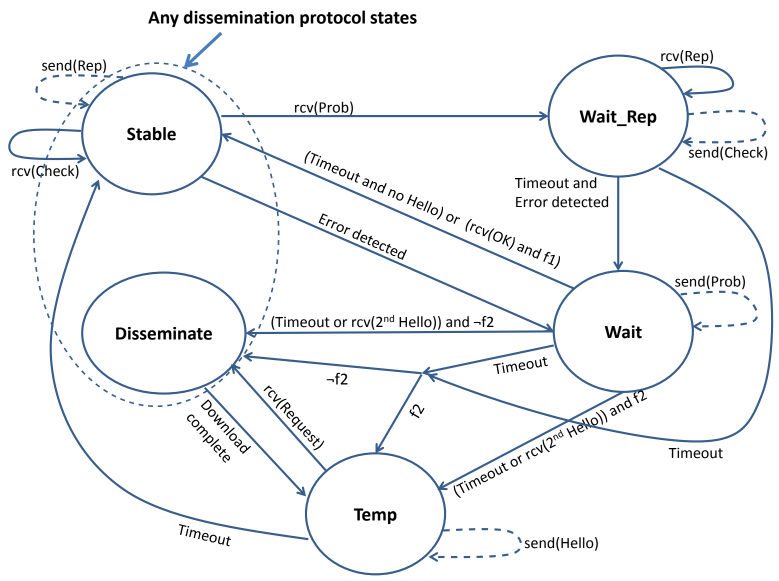

Any dissemination protocol states
send(Rep)
rcv(Check)
Stable
rcv(Prob)
Wait_Rep
rcv(Rep)
send(Check)
(Timeout and no Hello) or (rcv(OK) and f1)
Error detected
Timeout and Error detected
Wait
send(Prob)
Disseminate
(Timeout or rcv(2nd Hello)) and ¬f2
Timeout
¬f2
f2
rcv(Request)
Download complete
(Timeout or rcv(2nd Hello)) and f2
Timeout
Temp
send(Hello)
Timeout

Figure 4.8: The state machine for Consistent-Repair. The two states in dashed circle are the states of any code dissemination protocol. f1=TRUE if the sender of an *OK* packet is the node which sent an *H* packet, f2=TRUE if a node is updated, FALSE otherwise.

versions, then n_1 goes to the *Temp* state (n_1 concludes that it already has the updated code). Otherwise, it goes to the *Disseminate* state to update its code fragment from the node which has the new metadata.

– After waiting for *Wait_time*, if n_1 has not received at least one *Hello* packet, then it goes to the *Stable* state. If it receives one *Hello* packet, then n_1 goes to the *Temp* state if its metadata is not older than the received one. Otherwise, n_1 goes to the *Disseminate* state to update its code fragment. Note that, when n_1 receives two *Hello* packets with different metadata, it goes to the *Temp* or *Disseminate* state as mentioned above.

- **Wait_Rep state:** Let n_1 be the node which transitions to the *Wait_Rep* state.

  – After waiting for *WaitRep_Time*, if n_1 detects a fault after receiving *Rep* packets from its neighbours, then it broadcasts a *Prob* packet. Otherwise, n_1 checks if its metadata is newer than the received ones. If it is newer, then it goes the the *Temp* state, else it goes to the *Disseminate* state to update its code fragment.

We prove an important property of Consistent-Repair, in that Consistent-Repair generates a correction tree of depth at most $f + 2$, where f is the diameter of the F-affected area.

Lemma 2 (Correction Tree) *Given a network $G = (V, A)$, a detectable fault model F, and an F-affected area G', with the diameter of the area being f. Then, Consistent-Repair constructs a tree of depth at most $f + 2$ rooted at the node that first detects an error.*

Proof:

Assumptions: We denote a node that first detects an error by n_0, and we denote a node a distance d from n_0 by n_0^d. We assume that node n_0 detects an error after receiving a packet from some node n_0^1, and that n_0 is on the boundary of the F-affected area (some of its neighbours are F-affected, some are not), and $f > 1$.

According to Consistent-Repair, n_0 will send a Prob packet to n_0^1 and then goes to the Wait state. This starts a graph with n_0 as the root at depth=0 (see Figure 4.10). Node n_0^1, the child of n_0, has depth = 1. Node n_0^1, in turn, broadcasts Check packets to its neighbours. Node n_0^1 then goes to the Wait_Rep state to wait for Rep packets from the informed neighbours, which are at depth=2.

Now, for node n_0^1, since the diameter of the F-affected area is f, this means that n_0^1 will eventually send Prob packets to all senders of faulty Rep packets. We focus on one such node, which we denote by n_0^2. This process spawns a (sub)tree, with n_0^1

Variables of process *i*:
firstHello:{(id,version):id$\in \mathbb{N}$,version$\in \mathbb{N}$}

```
state==Wait

 case(upon⟨rcv (Hello, n, i)⟩)
   if(firstHello.version==∅)
     firstHello.version:=n.version
     firstHello.source:=n
   elseif(firstHello.version > n.version)
     if(i.version==firstHello.version)
       state:=Temp
       setTimer(TEMP,Temp_Time)
     else
       state:=Disseminate
     endif
   elseif(firstHello.version < n.version)
     if(i.version==n.version)
       state:=Temp
       setTimer(TEMP,Temp_Time)
     else
       state:=Disseminate
     endif
   endif
 endcase

 if(timeout(WAIT))
  if(firstHello.version==∅)
    state:=Stable
    stopAllTimers()
    TableProb:=∅; TableRep:=∅
  else
    if(i.version > firstHello.version)
      state:=Temp
      setTimer(TEMP, Temp_Time)
    else
      state:=Disseminate
    endif
  endif
 endif
```

```
state==Wait_Rep

case(upon⟨rcv (Rep, n, i)⟩)
  TableRep:= TableRep ∪ {(n, n.version)}
endcase

if(timeout(WAITREP))
  if(detect())
    bCast(Prob, i, ALL)
    state:=Wait
    setTimer(WAIT, Wait_Time)
  elseif(i.version < Table.version))
    state:=Disseminate
  else
    state:=Temp
    setTimer(TEMP, Temp_Time)
  endif
endif
```

Figure 4.9: Consistent-Repair Algorithm.

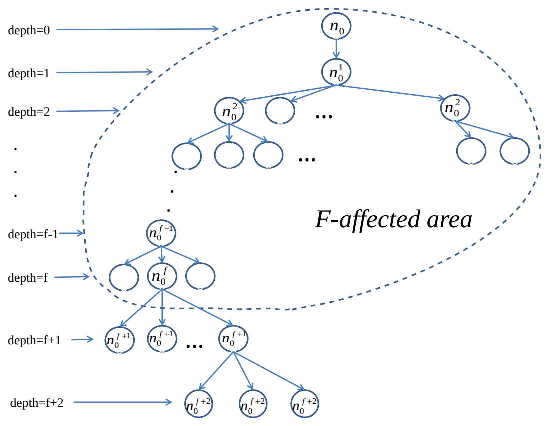


Figure 4.10: Correction Tree Constructed by Consistent-Repair

as the root of the subtree, and node n_0^2 will send Check packets to its neighbours (see Figure 4.10). When the node n_0^f (at the other end of the F-affected area is reached), in the worst case, it will detect faulty Rep packet from at least one node, which we denote by n_0^{f+1} (at a distance of $f+1$ from n_0). It will then send a Prob packet to n_0^{f+1}, which, in turn, sends Check packets to its neighbours at a distance of $f+2$. Since the F-affected area is of diameter f, all of the Rep packets to node n_0^{f+1} will hold at most two version numbers (and the tree does not grow anymore). At this point, node n_0^{f+1} can decide on an appropriate version number. Hence, the tree is of a depth of at most $f+2$.

In effect, when a node sends a Prob or Check packet, new subtrees are created, and the depth of the tree increases by 1. When a node receives identical information from its children, it sends a Hello packet to its parent, indicating that it has the associated code. At this point, the dependency of its children ends, reducing the depth of the tree by 1. It should be further noted that a tree is constructed for every node that detects an error. So, at any point in time, there may be several correction trees in the network.

We now prove the correctness of Consistent-Repair.

Lemma 3 (f-local correction) *Given a network $G = (V, A)$, a detectable fault model F, and an F-affected area G' of diameter f. Then, Consistent-Repair guarantees that, eventually, all nodes in G' will have a state consistent with their neighbourhood.*

Proof.

We will prove by induction on the correction tree (see Lemma 2) that node n_0, and all nodes in G', will eventually download the correct code.

Assumptions: (i) We assume a node n_0 has downloaded the stale code, (ii) node n_0 has detected an error (state inconsistency) (i.e., node n_0 is the root of the correction tree). We will denote a node at a distance d from another node n_0 by n_0^d.

Base case:

We prove for the case of a node, which we denote by n_0^{f+1} at depth $= f + 1$ (i.e., the last rooted subtree). Node n_0^{f+1} will eventually receive a set of Rep packets with identical version numbers. Node n_0^{f+1} will then eventually download the correct code from one of the Rep packets senders.

Inductive hypothesis:

Assume that a node n_0^i, where $0 < i \leq f$, eventually receives a *Hello* packet from a node n_0^{i+1} and then updates its code.

Inductive Step:

We need to prove that a node n_0^{i-1}, a neighbour node of n_0^i, eventually receives a *Hello* packet and updates its code.

In Consistent-Repair, node n_0^i will broadcast *Hello* packets periodically up to h times after receiving a *Hello* packet or having updated its code. If node n_0^{i-1} receives a *Hello* packet from n_0^i, n_0^{i-1} will update its code from one of the n_0^i nodes, which proves the inductive step. Else, if due to message losses, n_0^{i-1} does not receive a *Hello* packet from n_0^i, then, node n_0^{i-1} waits *Wait_Time* and goes to *PS* state and operates normally. Eventually, node n_0^{i-1} or a neighbour node of n_0^{i-1} will detect the error, and executes Consistent-Repair again. Assuming that the number of message losses is finite, eventually, node n_0^{i-1} will eventually get a *Hello* packet, when n_0^{i-1} can download the code from the node it receives the Hello packet from.

4.5 Experimental Setup and Results

In this section, we first present the simulation setup and results to evaluate the working and performance of both BestEffort-Repair and Consistent-Repair. Then, to confirm the consistency of the simulation results, we first perform a deployment of the protocols on a small-scale testbed available locally, and then, to obtain further confirmation, we perform a large-scale deployment.

4.5.1 Simulation Experiments

4.5.1.1 Simulation Setup

To evaluate the overhead of both protocols in large-scale networks, we conducted simulation experiments using TOSSIM [78] as simulator. The topology used in the simulations is 20x20 grid, with the distance between two nodes set at around 10 ft, with nodes having a communication radius of 30 ft. In WSNs, the grid topology is usually used to monitor (cover) a given area with the minimum number of nodes [24, 118]. For example, the grid topology has been used in intrusion detection and target tracking applications [11, 38]. However, note that no assumptions have been made regarding the specific network topology and size (see Lemma 3 for algorithm correctness and Section 4.5.2 for testbed experiments on a network with 3D topology). The only implicit assumption made is that the network should be connected, i.e., the node should have at least one bidirectional path to the sink, so that the node could correct its state, in the worst case, using the information from the sink. Further, as we use asymmetric links, the number of neighbours of the nodes varies. The network topology with asymmetric links is constructed by a tool given on tinyos.net. Each node is given a noise model from the heavy-meyer noise trace file located in Tossim/noise folder. TOSSIM takes a noise trace as input to generate a model that can capture bursts of interference and other correlated phenomena to improve the quality of the RF simulation [2].

Wait_Time	50 (30,300) sec	*SendRep*	2 sec
SendCheck	2 sec	*WaitRep_Time*	4 sec
SendHello	1.5 sec	*Temp_Time*	30 (20,150) sec
SendProb	1 sec	t	0.2 sec
PeriodProb	7 sec	p	5
U	60 (1) sec	h	2

Table 4.2: Parameter values used in simulation and testbed experiments (testbed values (small testbed, Indriya testbed) are within brackets, respectively).

The parameter values for the various timers of both protocols used in our testbed and simulation experiments are given in Table 4.2. Check, Hello, Prob and Rep are sent randomly between 0 and *SendCheck*, *SendHello*, *SendProb* and *SendRep* seconds, respectively. These timers are used to reduce the number of packet collisions and can be set to any other values. However, some of the parameter values depend on other parameters. For example, the timer used in the *Wait_Rep* state, *WaitRep_Time*, is the time for waiting for *Rep* packets after broadcasting a *Check* packet. So, *WaitRep_Time* $\geq$ *SendCheck*+*SendRep*. *Wait_Time* should be set according to the code size and the size of the network. If the network and code size is large, this time should be large enough to allow neighbouring nodes to correct their code and forward it. Otherwise, nodes in the *Wait* state may go to the *Stable* state without correcting themselves, and re-run the Repair algorithm again. Therefore, *Wait_Time* may notably affect the performance of the algorithm. Usually nodes enter the *Temp* state from the *Wait* state where it waits for a shorter time. The only case when a node waits for *Wait_Time* is when there is a packet loss. *Temp_Time* time is independent of other parameters. The value of t should be small because a node waits a maximum of *SendProb* time units to receive all possible *Prob* packets. The values of h and p can be set to any value depending on the link quality of the network. For example, if the network is lossy, then h could be set to higher values.

In our experiments, each node periodically broadcasts an application packet (or any other traffic that drives the dissemination) H, with the period randomly selected between $[0, U]$ at the start.

Recall that both BestEffort-Repair and Consistent-Repair is executed only when an error (i.e., erroneous state) is detected, so the faults injected were such that variables were modified in such a way to trigger an error that will be detected, leading to the execution of *Repair*. In our case, faults were artificially injected in Varuna by changing the version number and/or neighbourhood table entries of the faulty nodes, as in these cases faults can be detected. In the experiments, the faulty node(s) were booted after all correct nodes were successfully booted so as to assess the impact of faulty nodes on a dissemination process that is already in progress.

Although we assume only transient data faults that may alter the values in the memory and message (see Section 3.3.4), the algorithms by itself tolerate node crashes and message losses as long as the network is connected. However, it may take slightly longer latency if a node involved in the correction process crashes or message losses occur during the correction process. For example, if a node n crashes after receiving a request message, e.g., *Prob* or *Check*, or does not receive the same packet due to message loss, from its neighbouring node m, m may not correct its state as it will not get any information from n. However, as every state in the protocol has a timer, after waiting for a definite time, m will go to the *Stable* state, where it detects the error again and executes the algorithm to correct its state.

4.5.1.2 Simulation Scenarios

In our simulations, we simulated two scenarios: (i) **Scenario 1**: we varied the number of corrupted nodes per circular area, which has diameter of 60 feet (varying the fault density), and (ii) **Scenario 2**: we kept the number of corrupted nodes to 5 and increased the size of a given (square) area, i.e., decrease the fault density. In both scenarios, the nodes to be corrupted were selected randomly in the given area. We then counted (i) the number of Repair packets (BestEffort-Repair or Consistent-Repair) sent, (ii) the number of involved nodes, i.e., nodes that sent at least one Repair packet, and (iii) the number of nodes which changed their states to *Wait* and/or *Wait_Rep* states. For each given number of corrupted nodes in the first scenario and for each length of square area in the second scenario, we ran the simulations 5 times and computed the min, average and max values over the 5 runs.

4.5.1.3 Simulation Results

We first present the results of simulation experiments of BestEffort-Repair and then the results of Consistent-Repair.

BestEffort-Repair

Number of nodes: From Figure 4.25(a), we observe that, on average, the number of nodes executing the protocol varies linearly with the number of corrupted nodes. Given that the number of nodes involved is much less than the size of the network, it indicates that the number of nodes involved in the stabilisation process is proportional to the size of the corrupted area. Further, in Figure 4.25(b), we observe that, as the size of the area is increased (i.e., fault density decreases), the number of nodes executing the protocol becomes almost constant, on the average. This is because, with decreasing fault density, most faults tend to appear as a single independent fault, with each of them involving a similar number of nodes, and may only involve at most their 2-hop neighbourhood. This implies that, in general, BestEffort-Repair tends to access only a bounded neighbourhood (similar to Consistent-Repair). These two observations support the fact that *Repair* is f-local, with f being the diameter of the fault-affected region.

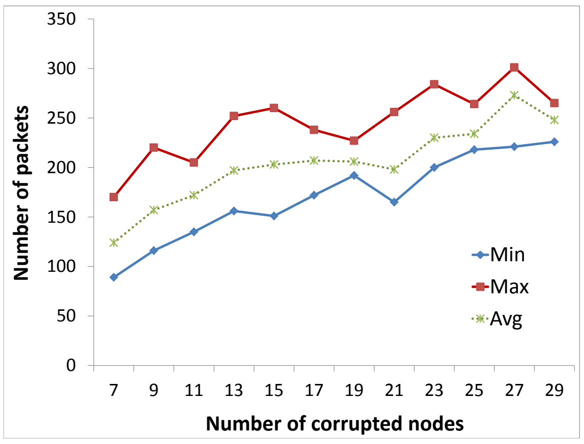
Number of packets: We notice a similar trend in Figure 4.11 that supports the observation that, in general, BestEffort-Repair tends to access a bounded neighbourhood. In Figure 4.11(a), we observe that the number of *Repair* packets sent varies linearly with the number of corrupted nodes. Since the number of *Repair* messages sent is much less than the size of the network, it implies that only part of the network was involved in the stabilisation process.

The discrepancy between the maximum number and minimum number of nodes or *Repair* packets is often due to the link quality, making retransmissions necessary.

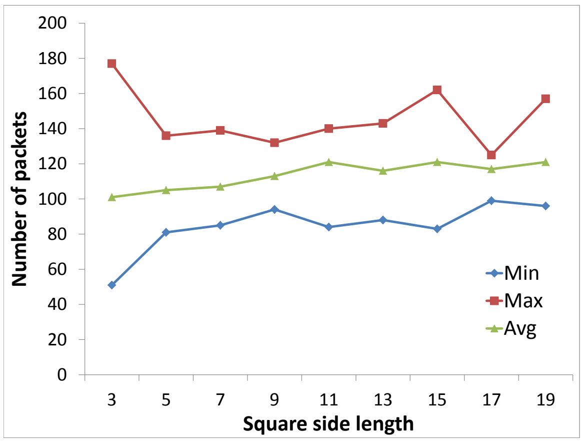
Consistent-Repair

We now present the result of simulation experiments of Consistent-Repair.

Number of nodes: From Figure 4.12(a), we observe that, on average, the number of nodes executing the Consistent-Repair varies linearly with the number of corrupted nodes. Given that the number of nodes involved is much less than the size of the network, it implies that the number of nodes involved in the stabilisation process is proportional to the size of the corrupted area. We also observe, in Figure 4.12(b), that, as the size of the area within which faults occur is increased (i.e., fault density decreases), the number of nodes executing the protocol becomes almost constant, on the average. This is because, with the decreasing fault density, most faults tend to appear as single independent faults, i.e., the fault-affected area is of size 1. Each corrupted node may only involve at most their 2-hop neighbourhood during recovery. These two observations support the fact that Consistent-Repair in f-local, with f being the diameter of the fault-affected region.

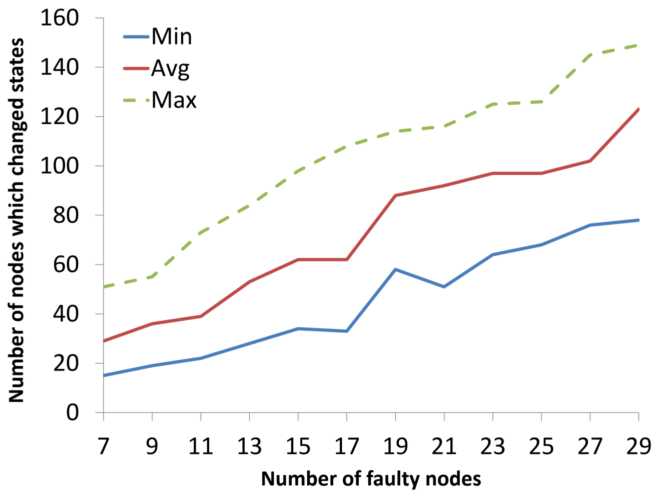


(a) Scenario 1: Number of packets vs Number of corrupted nodes, Area diameter = 60ft

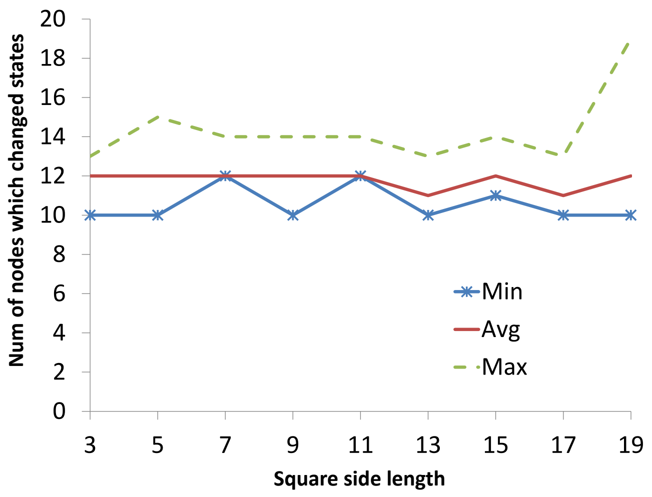


(b) Scenario 2: Number of packets for 5 corrupted nodes per square area.

Figure 4.11: Maximum, minimum and average number of transmitted BestEffort-Repair packets



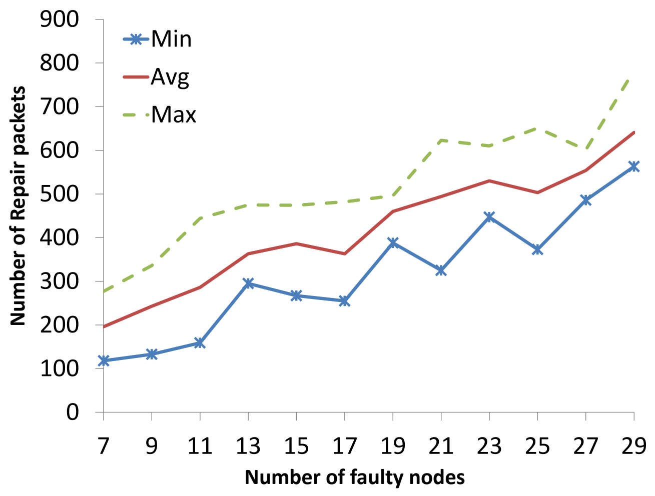
(a) Scenario 1: Num. of nodes executing Consistent-Repair vs Num. of Corrupted Nodes, Area diameter = 60ft



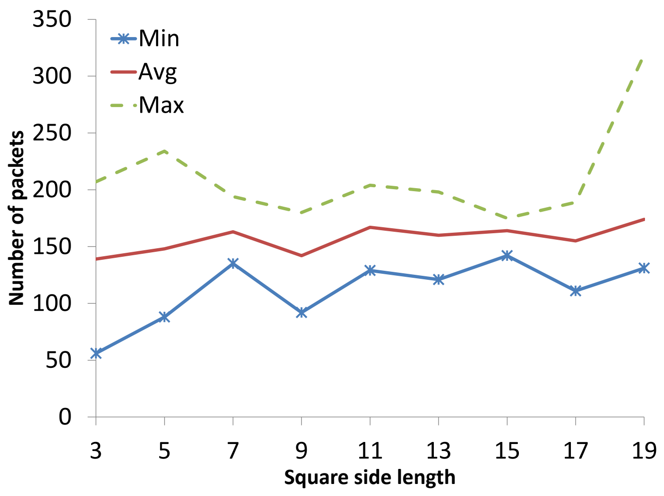
(b) Scenario 2: Num. of nodes executing Consistent-Repair for 5 corrupted nodes per sq. area.

Figure 4.12: Maximum, minimum and average number of nodes executing Consistent-Repair

Number of packets: We observe a similar trend in Figure 4.13 that supports the f-locality property of Consistent-Repair. In Figure 4.13(a), we observe that the number of *Repair* packets sent varies linearly with the number of corrupted nodes. Since the number of *Repair* messages sent is much less than the size of the network,



(a) Scenario 1: Num. of packets vs Num. of corrupted nodes, Area diameter = 60ft



(b) Scenario 2: Num. of packets for 5 corrupted nodes per sq. area.

Figure 4.13: Maximum, minimum and average number of transmitted Consistent-Repair packets

it implies that only part of the network was involved in the stabilisation process.

As in the case with BestEffort-Repair, the discrepancy between the maximum number and minimum number of nodes for Consistent-Repair packets is often due to the link quality, making retransmissions necessary.

Differences Between BestEffort-Repair and Consistent-Repair

From Figures 4.25 to 4.11 (for BestEffort-Repair) and Figures 4.12 to 4.13 (for Consistent-Repair), it can be observed that, in general, Consistent-Repair involves more messages and nodes. This is due to the fact that, given that Consistent-Repair makes more informed decisions to prevent any erroneous downloads, more nodes are involved and, thus, they send more messages. On the other hand, given that BestEffort-Repair is biased towards fast recovery, it attempts to make the network state consistent again, even if erroneous downloads are involved.

Tables 4.3 to 4.6 (for BestEffort-Repair) and Tables 4.7 to 4.10 (for Consistent-Repair) confirm that (i) the best case for BestEffort-Repair (i.e., minimum values) is, in general, better than that of Consistent-Repair and (ii) the worst case for BestEffort-Repair (i.e., maximum values) is, in general, worse than that of Consistent-Repair. As mentioned earlier, BestEffort-Repair can, in the worst case, involve the whole network during recovery, as opposed to Consistent-Repair which will only involve its $f + 2$ hop neighbourhood. In the best case, BestEffort-Repair may receive the proper Hello message first and helps the affected area to receiver quickly, whereas Consistent-Repair will wait for several messages to arrive before reaching the decision.

4.5.1.4 Case Study: Adding BestEffort-Repair and Consistent-Repair To Varuna

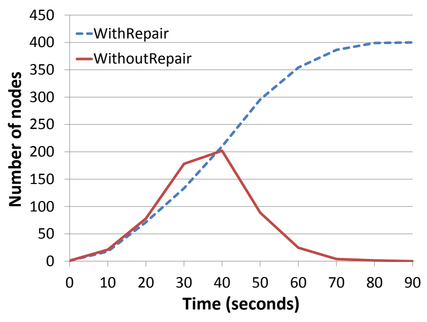
In this section, we discuss the addition of BestEffort-Repair and Consistent-Repair to Varuna [95]. The reason for choosing Varuna is that it is one of the latest code dissemination protocols that have been proposed. The code that was used in the deployment was reused for the simulation experiments described in this section.

As mentioned before, both protocols are triggered by the detection of an error in the state of the code dissemination protocol, in this case Varuna. In Varuna, such a detection is enabled by one of the following conditions: (i) two nodes' version numbers are corrupted in such a way that the difference in versions is strictly greater than 1, and (ii) the receiver of an advertisement message finds that its version is bigger than the advertised one and, at the same time, the sender of the message exists in its neighbourhood table. Also, we disallow faults, under the detectable fault model, that cause old code to appear as new and new as old (i.e., all updated nodes have old version numbers and non-updated nodes have the new version number). Also, this disallows nodes to be corrupted in identical ways.

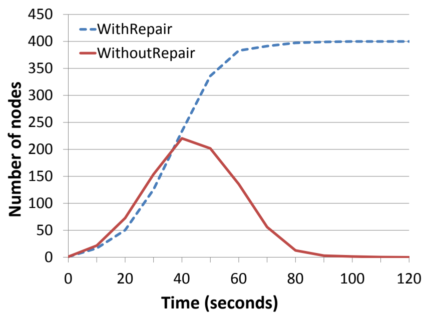
We simulated the composite protocol of Varuna and BestEffort-Repair and Consistent-Repair in TOSSIM. The experimental setup is the same as in 4.5.1.1. All nodes, except faulty nodes, are booted in the first minute. Faulty nodes are located at the center of the network. A packet with new version number is injected after 2 minutes. We simulated three faulty scenarios: (i) with 1 fault, (ii) with 4 faults and (iii) with 7 faults. For each faulty scenario, we booted the faulty nodes (i) 30 seconds, (ii) 45 seconds, and (iii) 60 seconds. This is so that only a proportion of nodes has the updated code version. The reason for booting faulty nodes some time after the updated code is injected is to ensure that nodes that have the stale code are chosen to have faults injected into them. We are specifically interested in (i) the overhead induced by Repair on the performance of Varuna and (ii) the number of nodes with correct code at a given time. We simulated Varuna in conditions similar to those detailed in Section 4.5. Further, the values for Varuna-specific parameters are: DISS-RAND=2 sec, ADV-RAND=2 sec, τ=8 sec, T_{MOODY}=1 min.

Performance of Best-Effort Repair

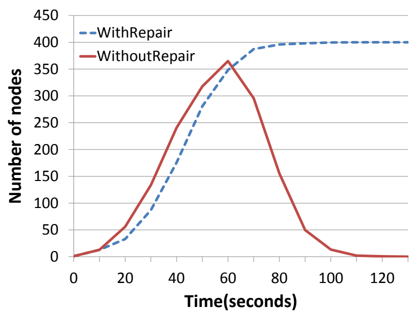
From Figures 4.14, 4.15 and 4.16, we make two important observations: (i) In all cases, injecting transient faults in the network during Varuna execution causes the whole network to disseminate stale code. This shows that Varuna cannot handle



(a)



(b)



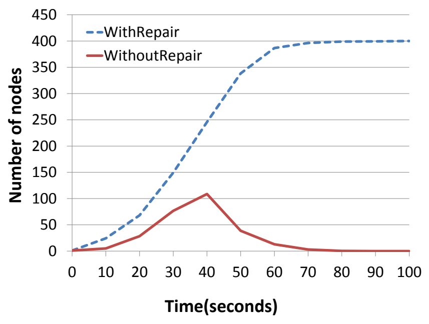
(c)

Figure 4.14: Varuna and Varuna ∘ Best-Effort-Repair: 1 faulty node booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

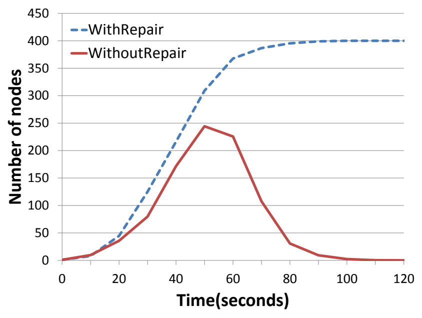
transient faults. On the other hand, when BestEffort-Repair is added to Varuna, every node eventually downloads the correct code.

Special Case

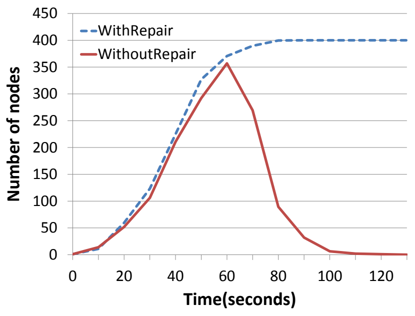
In a special case, we simulated the case where, due to situations such as duty cycling, some nodes may have been sleeping, missing the code update. In Figure 4.17, 4 such nodes are booted 180 seconds after the code update has been injected into the network. Further, these 4 nodes are faulty as well. We observe that, in Varuna, all nodes in the network eventually end up downloading the stale code, while the composite protocol of Varuna and Repair ensures that the whole network has the updated code, without any updated node erroneously downloading stale code.



(a)



(b)

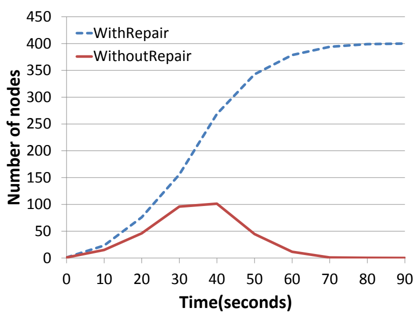


(c)

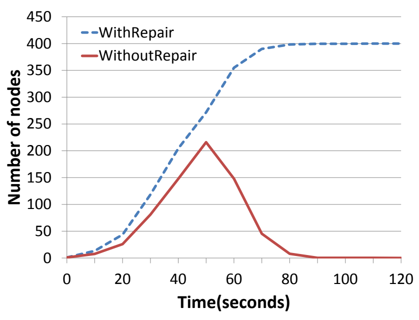
Figure 4.15: Varuna and Varuna ∘ Best-Effort-Repair: 4 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

Packet Overhead In Figure 4.18(a), it can be seen that the packets overhead induced by Repair on Varuna is low. Specifically, with 4 faulty nodes, the packet overhead is 0.4% while, with 7 faulty nodes, the packet overhead is less than 3%. From Figure 4.11, it can be observed that the number of Repair packets will increase linearly with increasing number of corrupted nodes. The reason for the linear increase (as opposed to a constant value) is that the fault density increases when more corrupted nodes appear at the centre of the network (condition under which we simulated the composite protocol).

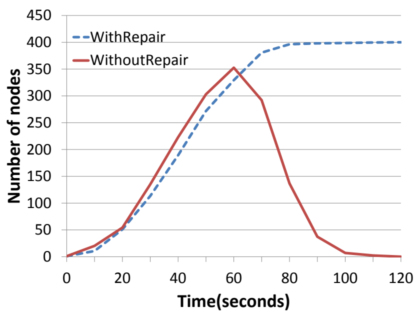
Temporal Overhead In Figure 4.18(b), it can be observed that the whole network receives the new code in approximately 80 seconds, after the new code has been injected into the network. Further, it can be observed that, when there are faulty



(a)



(b)



(c)

Figure 4.16: Varuna and Varuna ∘ Best-Effort-Repair: 7 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

nodes in the network, the time for the whole network to receive the correct code is approximately 80 seconds. Thus, *there is almost no temporal overhead induced by BestEffort-Repair on Varuna*, highlighting the fact that BestEffort-Repair is biased towards fast recovery.

Performance of Consistent-Repair

From Figures 4.19, 4.20 and 4.21, we make one important observation: When Consistent-Repair is added to Varuna, every node eventually downloads the correct code.

Packet Overhead In Figure 4.22, it can be seen that the packets overhead induced by Consistent-Repair on Varuna is very low. Specifically, with 4 faulty nodes, the

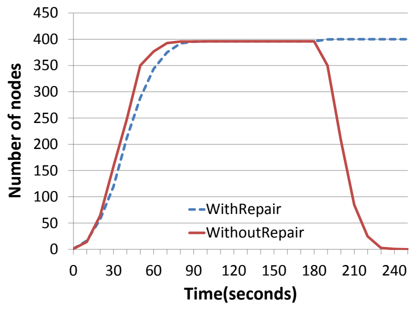
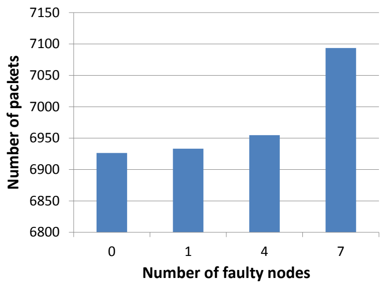
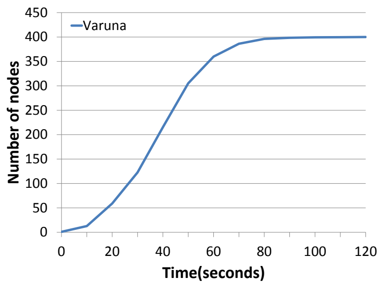


Figure 4.17: 4 faulty nodes booted 180 seconds after updated code injection.



(a) Number of (Repair + ADV) packets sent vs number of corrupted nodes

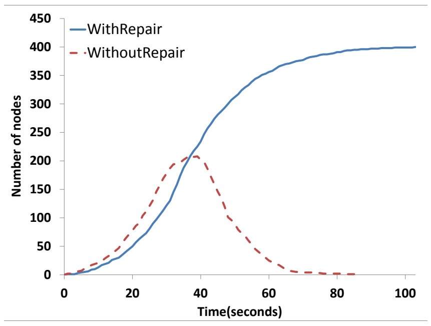


(b) Completion of Varuna (no transient faults)

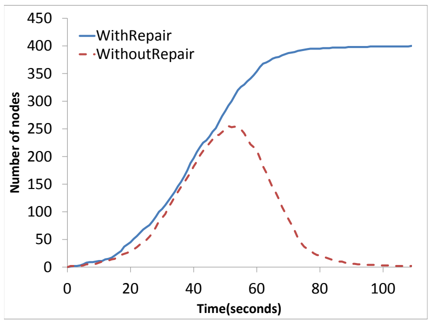
Figure 4.18: Varuna ∘ BestEffort-Repair : Network Size: 20 * 20, Nodes corrupted at random

packet overhead is less than 0.8% (see Figure 4.22). From Figure 4.13, it can be observed that the number of Repair packets will increase linearly with increasing number of corrupted nodes. The reason for the linear increase (as opposed to a constant value) is that the fault density increases when more corrupted nodes appear at the centre of the network (condition under which we simulated the composite protocol).

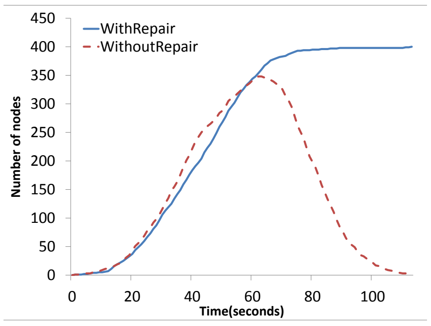
Temporal Overhead In Figure 4.18(b), the whole network receives the new code in approximately 80 seconds after the new code has been injected into the network. Further, it can be observed that, when there are faulty nodes in the network, the time for the whole network to receive the correct code is approximately 90-100



(a)



(b)



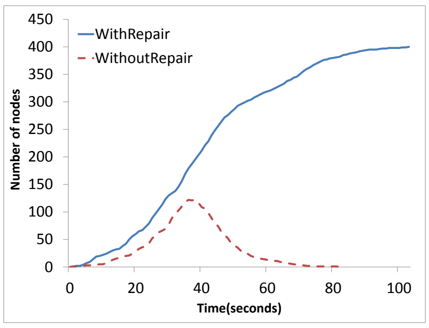
(c)

Figure 4.19: Varuna and Varuna ∘ Consistent-Repair: 1 faulty node booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

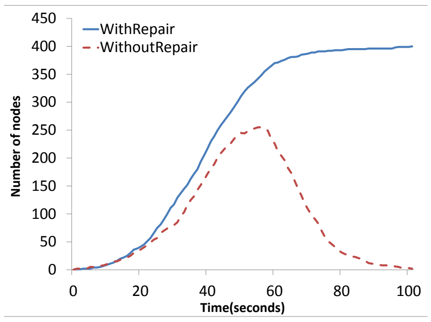
seconds. Thus, *there temporal overhead induced by Consistent-Repair on Varuna is approximately 10%–20%*. This supports the fact that Consistent-Repair needs more time for informed decisions (as opposed to BestEffort-Repair).

Difference Between BestEffort-Repair and Consistent-Repair

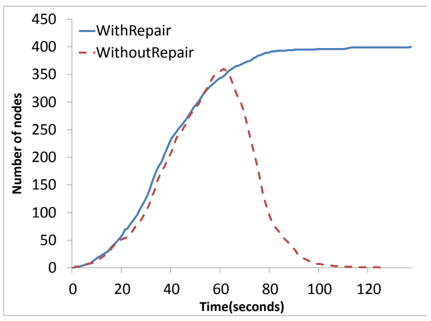
Up to now, we have observed that when adding BestEffort-Repair and Consistent-Repair to Varuna, all nodes eventually download the updated code, meaning that both of them are correctors for strong CD. It has been shown that the temporal overhead induced by BestEffort-Repair on Varuna is lower than that of Consistent-Repair as well as the packet overhead of BestEffort-Repair (0.4%) on Varuna is roughly 2 times as low as Consistent-Repair (0.75%). This is due to the fact that BestEffort-



(a)



(b)



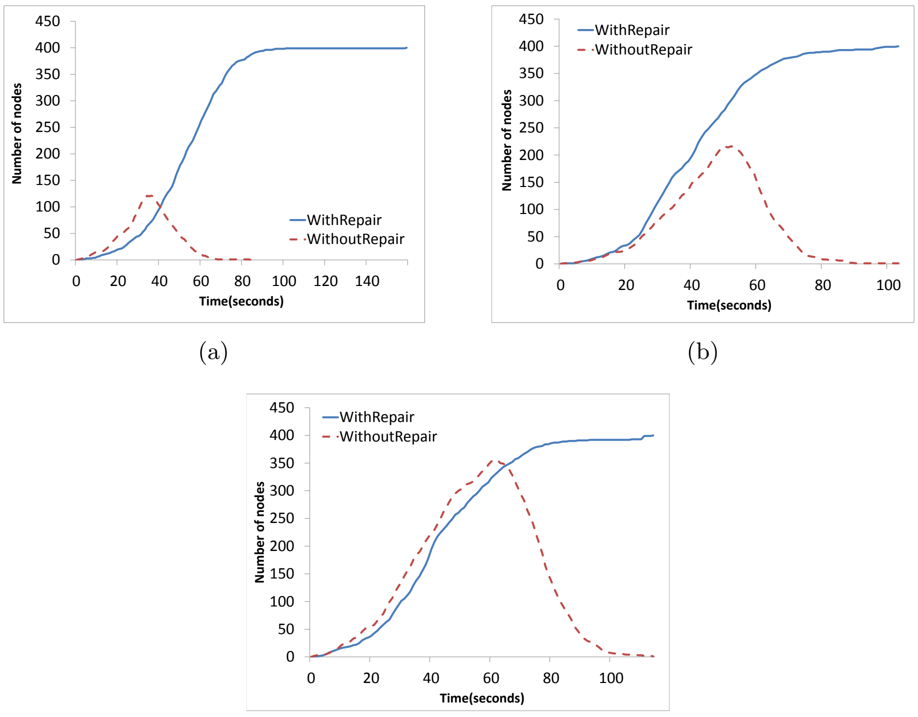
(c)

Figure 4.20: Varuna and Varuna ∘ Consistent-Repair: 4 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

Repair is biased towards fast recovery, requiring less packets. On the other hand, we motivated Consistent-Repair to allow for more informed recovery in that it reduces the number of erroneous downloads (where an updated node ends up downloading the old code to eventually update again). In this respect, in our experiments, we observed that, on average, BestEffort-Repair causes 5 erroneous downloads - which is allowed under the BestEffort CD specification (to eventually download the correct code), whereas, with Consistent-Repair, there were no erroneous downloads.

4.5.2 Testbed Experiments

As mentioned earlier, to confirm the consistency of the simulation results, we perform experiments on real-world testbeds.



(a) (b) (c)

Figure 4.21: Varuna and Varuna ∘ Consistent-Repair: 7 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

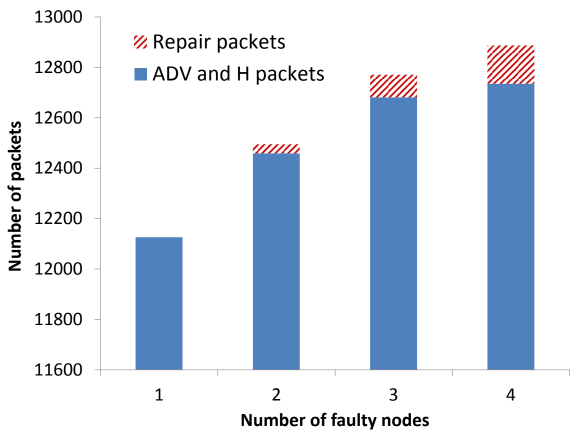


Figure 4.22: Number of (Repair + ADV) packets sent vs number of corrupted nodes

4.5.2.1 Testbed Setup

We have run the experiments on Indriya testbed [36] which has 3D topology and has been deployed over three floors (Figure 4.23) of School of Computing building of the National University of Singapore. When we were performing experiments the number of nodes in Indriya was about 100. The nodes, which are powered over USB, are TelosB motes with C2420 radio, 8 MHz CPU, 10 KB RAM and 48 KB of program memory. The transmission power was set to the default value 31. We used the default channel 26, which has less interference with other wireless technologies [36]. Each experiment, depending on the number of faulty nodes, took from 15 to 30 minutes. On average, an experiment took 20 minutes. In total, we have run about 600 experiments. So, all experiments took about 200 hours. The experiments were conducted at different times of the day and on different days of the week. For each experiment the transmitted packets and timings were logged into a different file through USB for analysis purposes.

We have also run experiments on a small testbed consisting of 10 TelosB motes in a lab setting. In the small testbed experiments we set the transmission power to a very low level 2 to make a small multi-hop network of different topologies. We used the default channel 26. The topologies used in small deployment experiments are described in Section 4.5.2.2. The experiments were run during morning and day times over a period of one week. The LEDs of motes were used to identify whether the algorithm has completed correcting the states. For example, the red led of mote was used to show that an error has occurred, the green led was used to show that the mote is consistent with its neighbours. Whenever the algorithm completed the correction, we manually reset all motes to repeat the experiment. During the experiments the transmitted packets and latency were logged into a file through a USB port.

Our implementation of BestEffort-Repair takes 222 bytes of memory, which in-

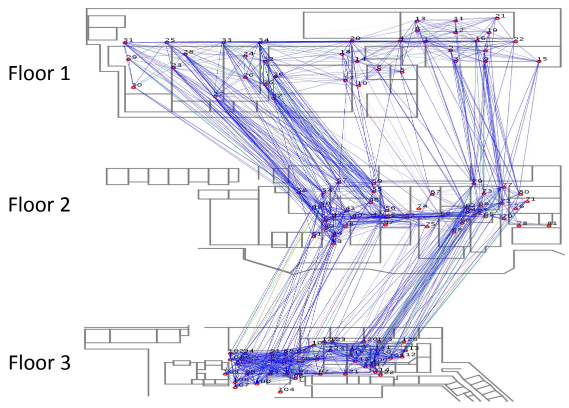


Figure 4.23: Indriya testbed

cludes two tables (TableProb and TableRep - see Figure 4.4), each of size 50 entries of 2 bytes each, and other algorithm related variables. Depending on the size of the network, the size of the tables can be varied. On the other hand, our implementation of Consistent-Repair takes 144 bytes, which includes 3 arrays of size 3*2, with each entry of size 2 bytes.

4.5.2.2 Testbed Scenarios

Small Testbed Scenarios

Our claim is that both BestEffort-Repair and Consistent-Repair can help any code dissemination protocol that has enough state to enable the detection of an erroneous state to eventually guarantee that every node has the updated code. As a result, we tested both BestEffort-Repair and Consistent-Repair by adding them to Varuna [95], one of the latest code dissemination protocols, on three different network topologies under four different scenarios. In the first scenario, the network was complete where all nodes could communicate with each other. In the second scenario, the network topology was formed by placing a faulty node at one end of the network in such a way that it has only one neighbour, with the network remaining connected and multi-hop (see Figure 4.24).

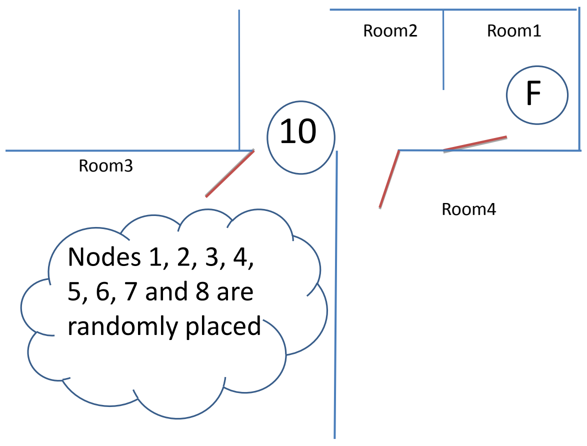


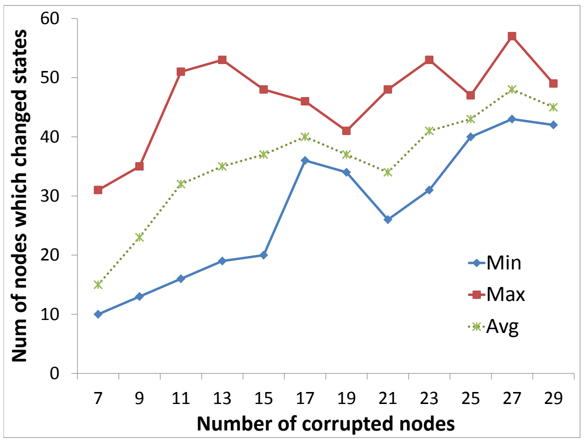
Figure 4.24: Second scenario: Topology where a faulty node has only one neighbour.

In the third scenario, the topology was formed by randomly deploying the nodes such that the network is connected and multi-hop. In these three scenarios, only one node is used as a faulty node with a corrupted version number or corrupted neighbourhood table, since Varuna's state consists of a neighbourhood table and a variable holding the version number. Finally, in the fourth scenario, the network topology that is used is the same as that used in scenario three, but with two faulty nodes: one with the version number corrupted and the other with the neighbourhood table corrupted.

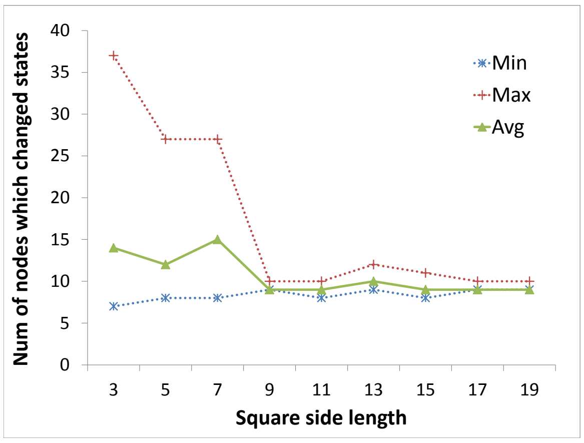
Indriya Testbed Scenarios

As mentioned above, in Indriya testbed more than 100 nodes have been deployed across three floors (Figure 4.23). However, during our experiments only 100 of them were available.

In the first testbed scenario, we varied the number of faulty nodes to 1, 4 and 7. All faulty nodes were selected from the third floor. Also, we varied the number of faulty nodes depending on the probability. Thus, in the second scenario, of 24 nodes



(a) Scenario 1: Number of nodes executing BestEffort-Repair vs Number of corrupted nodes, Area diameter = 60ft



(b) Scenario 2: Number of nodes executing BestEffort-Repair for 5 corrupted nodes per square area

Figure 4.25: Maximum, minimum and average number of nodes executing BestEffort-Repair

from the third floor, nodes become faulty with probability 0.5. In the third scenario, nodes from the entire network (about 100 nodes) become faulty with probability 0.25. In all scenarios, the base station was selected from the first floor.

4.5.2.3 Testbed Results

In our experiments, we measured (i) the number of transmitted *Repair* packets and (ii) the latency required to correct the error. The first metric is to show the relation between the number of faulty nodes and the message overhead induced to correct the faulty nodes.

Small Testbed Results

For each scenario, we ran BestEffort-Repair and Consistent-Repair 20 times and computed the average of the *Repair* packets and latency.

We first present the results of Best-Effort Repair and then the results of Consistent-Repair.

Best-Effort Repair

As expected, in all of our experiments, adding BestEffort-Repair to Varuna ultimately corrected the errors. The results obtained for scenarios 1 . . . 4 are shown in Tables 4.3, 4.4, 4.5, and 4.6 respectively. In all cases, the average number of *Repair* packets and the latency are reasonably low. For example, focusing on Table 4.3, the minimum number of *Repair* packets is proportional to the size of the neighbourhood of the faulty node. This is as expected since most nodes are expected to send *Rep* packets due to the network being complete. The difference between minimum and maximum values is due to the loss characteristics of the wireless medium. This property of wireless medium can be seen from the Table 4.6, where the highest latency among all experiments was 218218 milliseconds (218.218 seconds) and the largest number of packet transmissions was 82, whereas the smallest latency and number of packets were 14134 milliseconds and 15, respectively. In that particular case, Prob packets sent by a node that detected the error was not received by a receiver. Therefore, the node had to retransmit Prob packets and the number of Prob packet retransmissions was 29 and the time taken for that was about 155000 (155 seconds) milliseconds.

Scenario 1	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	8	23	12.6	7	14	12.4
Time(millisec)	6962	17544	8070	6011	14449	8791

Table 4.3: Scenario 1, BestEffort-Repair: Complete network

Scenario 2	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	10	39	19	9	39	18
Time(millisec)	5497	111142	31518	4719	83760	15013

Table 4.4: Scenario 2, BestEffort-Repair: Faulty node at one end of the network

Scenario 3	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	16	45	24	10	26	24
Time(millisec)	15745	124786	50472	10705	144214	37097

Table 4.5: Scenario 3, BestEffort-Repair: Connected Random graph, 1 fault

Scenario 4. Version and table corrupted			
	Min	Max	Avg
Number of packets	15	82	27
Time(millisec)	14134	218218	59231

Table 4.6: Scenario 4, BestEffort-Repair: Connected Random graph, two faults

Consistent-Repair

As expected, in all of our experiments, adding Consistent-Repair to Varuna ultimately corrected the errors. The results obtained for scenarios 1 . . . 4 are shown in Tables 4.7, 4.8, 4.9, and 4.10 respectively. In all cases, the average number of

Repair packets and the latency are reasonably low. For example, focusing on Table 4.7, the minimum number of *Repair* packets is proportional to the size of the neighbourhood of the faulty node (as in the case for BestEffort-Repair). This is as expected since most nodes are expected to send *Rep* packets due to the network being complete. The difference between minimum and maximum values is due to the loss characteristics of the wireless.

Scenario 1	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	12	29	16.2	14	30	16.3
Time(millisec)	30137	30385	30351	30372	30383	30375

Table 4.7: Scenario 1, Consistent-Repair: Complete network

Scenario 2	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	13	24	15.6	11	36	16.5
Time(millisec)	30179	60937	31467	30183	112728	37304

Table 4.8: Scenario 2, Consistent-Repair: Faulty node at one end of the network

Scenario 3	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	9	32	16.2	9	28	15.5
Time(millisec)	30336	30429	30374	30100	30391	30288

Table 4.9: Scenario 3, Consistent-Repair: Connected Random graph, 1 fault

Scenario 4. Version and table corrupted			
	Min	Max	Avg
Number of packets	12	42	28
Time(millisec)	30545	61003	43498

Table 4.10: Scenario 4, Consistent-Repair: Connected Random graph, two faults

Indriya Testbed Results

On Indriya we have run the Consistent-Repair algorithm as this algorithm minimizes the number of erroneous downloads compared to BestEffort-Repair by making better update decisions. As expected, in all experiments Consistent-Repair corrected faulty nodes.

Latency Figures 4.26(a) and 4.26(b) show the time taken to correct all faulty nodes in the network. As can be observed from Figure 4.26(a), the times to correct networks with 1, 4 and 7 faulty nodes are the same and equal to about 300 seconds. They have the same latency because, as described in 4.4.2.2, the Consistent-Repair algorithm waits until constant time to make decisions, in this case 300 seconds. That means that 300 seconds was enough to correct faulty nodes. While in Figure 4.26(b), as the number of faulty nodes is larger, 300 seconds was not enough to correct all faulty nodes and thus might go to second round (another 300 seconds). The experiments show that the time to wait to make decisions depends on the number of faulty nodes, which confirms the simulation results.

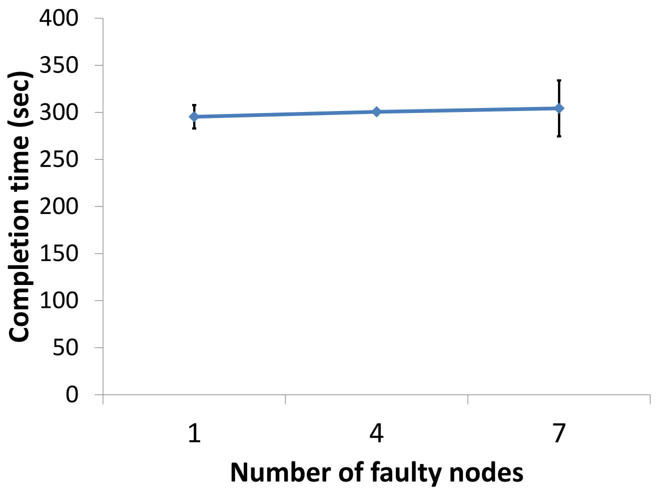
Number of packets Figures 4.27(a) and 4.27(b) show the number of Repair packets transmitted to correct faulty nodes. These figures support the conclusion obtained from the simulation results that the number of transmitted packets to correct faulty nodes varies linearly with the number of faulty nodes.

Number of nodes Figures 4.28(a) and 4.28(b) show the number of executing the Consistent-Repair algorithm. As can be seen from both figures, like in the results obtained from simulations, the number of nodes involved in correcting faulty nodes increases linearly with the number of faulty nodes.

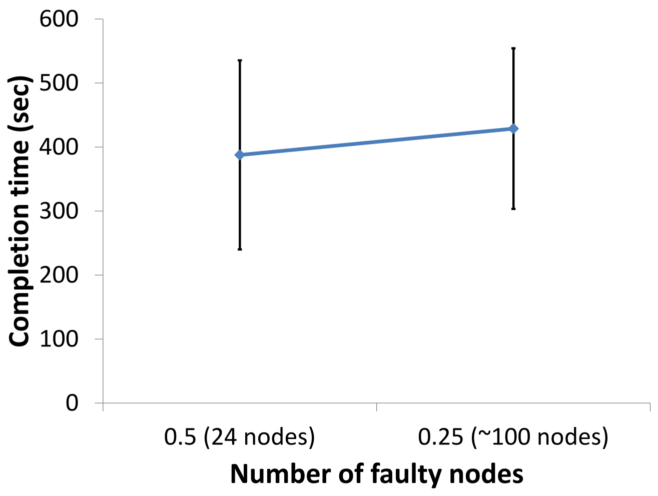
The last two figures confirm the results shown in Figures 4.26(a) and 4.26(b) that the algorithm is local and can work in networks of bigger size.

4.6 Conclusion

In this chapter, we have addressed the problem of data dissemination in the presence of transient faults that corrupt the state of the data dissemination program.



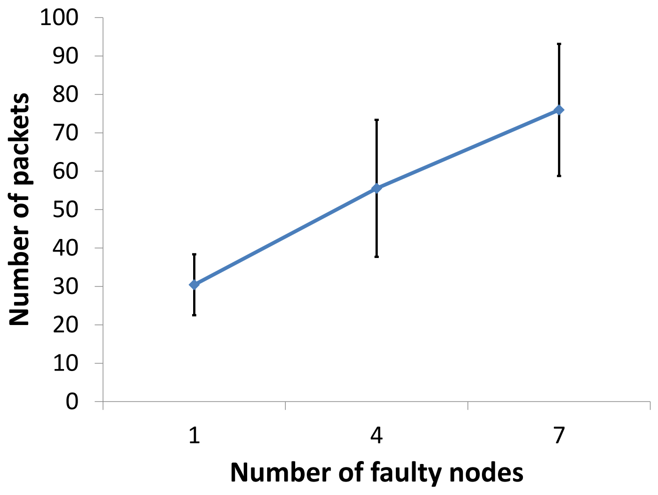
(a) Scenario 1: Fixed number of faults



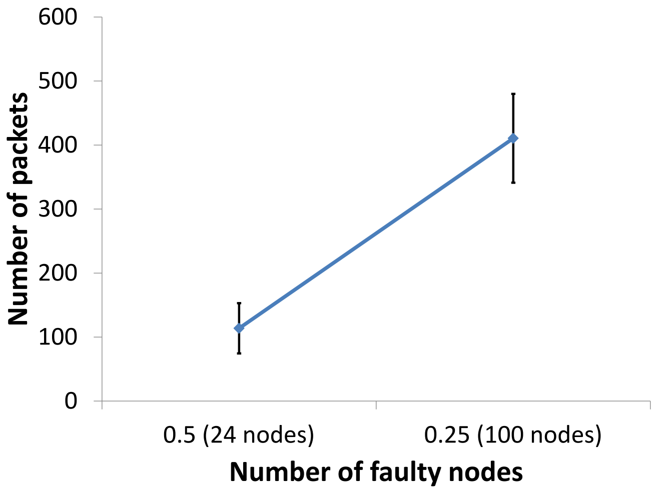
(b) Scenario 2 and 3: 24 nodes and 100 nodes are faulty with probability 0.5 and 0.25, respectively

Figure 4.26: Completion time

We have proposed two protocols, namely BestEffort-Repair and Consistent-Repair. The proposed protocols when added to a fault-intolerant dissemination protocol, make it a non-masking fault-tolerant protocol. The protocols are generic corrector protocols, i.e., the protocols are not based on a specific protocol implementation, rather they are based on the protocol specification. In general, BestEffort-Repair



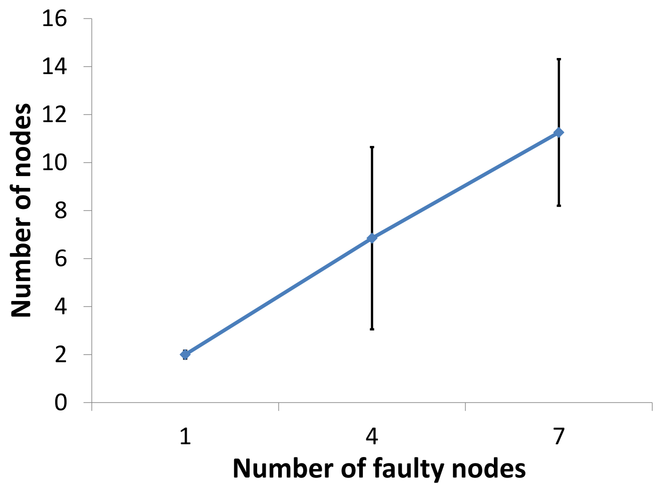
(a) Scenario 1: Fixed number of faults



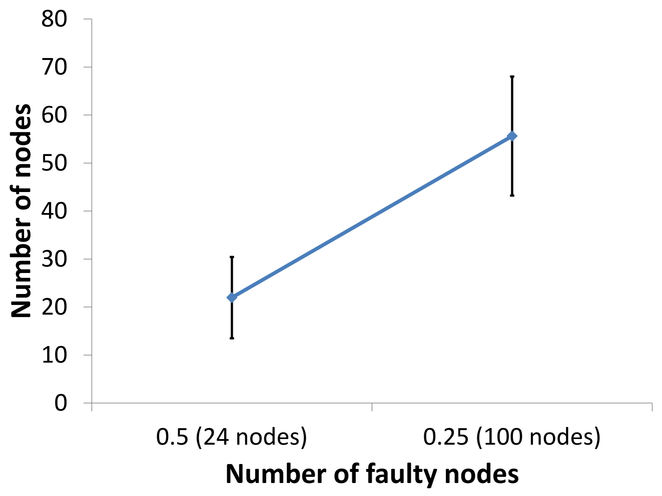
(b) Scenario 2 and 3: 24 nodes and 100 nodes are faulty with probability 0.5 and 0.25, respectively

Figure 4.27: Number of transmitted Repair packets to correct faulty nodes

and Consistent-Repair could be integrated to protocols where all nodes in the network at the end should have a common value. So, they could be integrated not only to data dissemination protocols where the sink disseminates data to all other nodes, but also to protocols like [46, 83, 113] where the node broadcasts its information or sensor observations to all other sensor nodes for system state synchronization and



(a) Scenario 1: Fixed number of faults



(b) Scenario 2 and 3: 24 nodes and 100 nodes are faulty with probability 0.5 and 0.25, respectively

Figure 4.28: Number of nodes involved in correcting faulty nodes

fault tolerant and security issues.

In the area of WSN, the same technique can be applied, for example, to the class of TDMA MAC protocols to correct conflicting slots. Specifically, the MAC protocol can be specified in terms of safety and liveness (like we have specified the dissemination protocol in this chapter). And then, a generic corrector component can be developed that will cause the protocol to correct itself irrespective of the implementation. That is, whenever a generic corrector component detects that there are two nodes that have conflicting slots, it will force them to repair themselves irrespective of which TDMA MAC protocol they use.

CHAPTER 5

Triva: An Adaptive Code Dissemination Protocol for WSNs

5.1 Introduction

Any code dissemination protocol needs to satisfy some important properties: (i) *energy efficiency*: wireless communication has a high energy cost, and primarily defines the system lifetime. Where laptops or mobile phones can be recharged, sensor networks die due to energy exhaustion. Thus, an effective code dissemination protocol must send as few packets as possible, while ensuring that all target nodes receive the code update. (ii) *dissemination latency*: while the new code is being propagated, the network may be in an erroneous, useless state, since interacting nodes may have different code versions running, possibly running different missions. In this case, transition time is wasted time, leading to a waste in energy. Therefore, an effective code dissemination protocol must also propagate new code quickly [79].

The code dissemination protocol typically consists of two components, namely (i) a code maintenance part and (ii) a code download part. The code maintenance enables nodes to determine if they need to download new code or not, whereas the code download part enables relevant nodes to download the code.

To achieve the first goal, viz. energy efficiency, protocols control energy expenditure in various ways, especially during code maintenance. For example, energy efficiency is achieved by either reducing the number of messages being sent [79] (using some form of "polite gossip") or using some well-defined duty-cycling [70] or by integrating the version inconsistency detection during payload communication, thereby avoiding special packet transmission [95]. On the other hand, one way of reducing the code dissemination latency is to periodically perform a "polite gossip", i.e., periodically perform a code maintenance (to determine whether any node needs code updating). Therefore, there seems to be a tradeoff between dissemination latency and energy efficiency: specifically, to reduce latency, version inconsistency needs to be detected fast, requiring periodic transmission of code information. On the other hand, to reduce energy consumption, inconsistencies need only be detected "when needed", i.e., when nodes communicate.

The Trickle [79] (see Section 2.1.1.1) is an algorithm that achieves low dissemination latency through periodic advertisement of the new code. However, there is a steady expenditure of energy, even when the network is in a steady state (i.e., when no node needs updating). This is due to the proactive step to detect inconsistencies. On the other hand, in Varuna [95] (see Section 2.1.1.3), inconsistencies are detected during application message communication. In this case, no additional energy is spent in the steady state.

After all, these two algorithms have drawbacks which will be detailed in the next two sections.

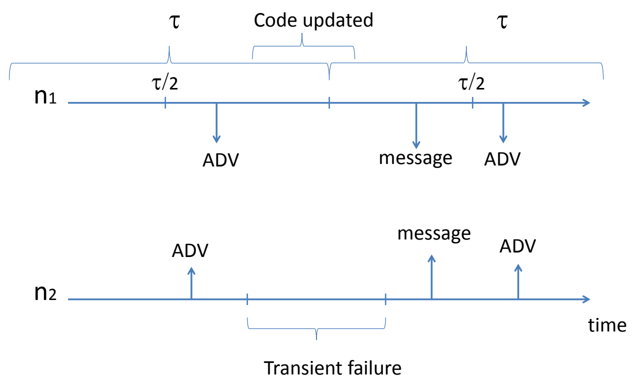


Figure 5.1: Illustration for the Trickle problem.

5.1.1 Drawbacks of Trickle

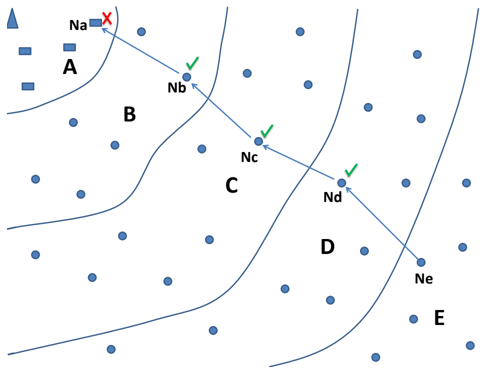
Other than constant energy consumption in the steady phase, as noted in [95], Trickle algorithm has another problem that nodes could communicate with each other even though they have different codes. As a result, the sink could receive an incorrect report such as a false alarm. This problem could happen if the advertisement interval τ is larger than the code download period. For example (see Figure 5.1), let two nodes n_1 and n_2 be neighbours of each other. After first exchange of ADV packets both nodes will agree on the consistency. Assume that n_1 updates its code after its first ADV transmission, and in that period n_2 goes disconnected for reasons such as transient failures and therefore has not updated. Now, all message exchanges between n_1 and n_2 will be inconsistent until their next ADV transmission.

5.1.2 Drawbacks of Varuna

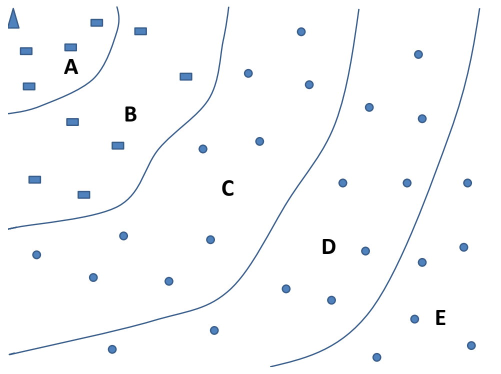
WSN applications can be classified according to their data delivery model as either continuous (periodic), event-driven, observer-initiated or hybrid [109]. For event-based WSNs, messages are sent only when a given event is detected, over a time period [11, 116]. In applications like forest fire detection and flood detection, when a node detects a fire or a flood, it must immediately send an alarm to the sink for a certain time.

However, if events of interest are far between, i.e., the occurrences of the events are rare, then any code dissemination protocol that are based on regular data communication, e.g., Varuna, can suffer from high dissemination latency. If data packets are sent rarely, like in such event-based sensor networks, nodes further away from the sink will update their code very late, depending on how frequently data packet is sent.

Moreover, in Varuna, due to code incompatibilities most of the application data will be discarded by intermediate nodes during convergecast. This reduces the yield of the network. Figure 5.2(a) illustrates these drawbacks of Varuna. For the sake of simplicity, consider a multi-hop network in which nodes located in the same region labeled with X have the same number of hops away from the *sink*. The nodes in neighbouring regions have one hop distance. Assume that the triangular node located in the corner is sink, the rectangular nodes in the region A are updated nodes and the circular nodes in other regions are not updated. According to Varuna protocol, nodes in regions B, C, D and E communicate and accept each others data because they have the same version code. However, since the nodes in region A have a bigger version number than the nodes in other regions, the nodes in region A should not communicate with other nodes. They will not forward messages received from the nodes in B and will discard them. They detect inconsistency and let the nodes in region B update their code. For example, assume that a node N_e in region E has detected a fire and wants to send an alarm message to the *sink* immediately. It sends to a node N_d in D, which accepts and forwards it further to a node N_c in C. N_c forwards to a node N_b. When data is forwarded from N_b to a node N_a in region A, N_a will cancel it because the node N_b has lower version than N_a. In other words, the data originated at N_e will be waste after flowing through the entire



(a) Multihop network divided into regions according to hop distance from the sink.



(b) Network after communications of nodes in region B with the nodes in region A.

Figure 5.2: Varuna in action.

network. Similarly, many packets that have been sent since the time when a new code was injected from the nodes in B, C, D and E will be discarded. Therefore, when events are rare, leading to bursts of data packets, Varuna not only wastes resources such as energy and bandwidth, but also increases dissemination latency and delivery latency. Figure 5.2(b) shows the network after communication of nodes in region B with the nodes in region A.

Further, in WSNs, link quality can fluctuate [13], causing asymmetric links to exist in the network, and this can be due to several reasons. For example, the network can transmit low-power signals, thus creating links that are often asymmetric. The link quality depends strongly on hardware inaccuracy and environmental factors [13]. In [82], the authors observed that transceiver frequency mismatch can also be a reason for asymmetric links. Further, the duration of these asymmetric links may be very small (i.e., transient) or very long (i.e., permanent) [91]. Asymmetric links create several major problems in wireless sensor networks [110]. For example, protocols that assume bidirectional links may not work or may not be efficient in networks with asymmetric links. Specifically, Varuna will not work efficiently if asymmetric links exist in the network.

Overall, Trickle spends constant energy, even in steady state due to communication for code maintenance, while Varuna, which addresses the shortcomings of Trickle, does not perform well in presence of asymmetric links or if the network application is event-based.

Thus, there is a need for a code dissemination protocol that performs well in event-based WSNs and that tolerates asymmetric links. Such a protocol needs to have the following properties: (i) low dissemination latency as Trickle, and (ii) does not incur steady energy expenditure during steady state, as Varuna. To achieve this, in this chapter we propose a new protocol, called *Triva*, which is an adaptive code update maintenance protocol that leverages the properties of Trickle and Varuna. Specifically, when dissemination is needed, Triva behaves like Trickle, but when dissemination is not needed (maintenance is needed), it behaves like Varuna. Our results show that Triva outperforms both protocols in general. Further, Triva outperforms both Trickle and Varuna in presence of asymmetric links. We also show that Triva handles bursty traffic much better than Varuna.

5.1.3 Chapter Structure

This chapter is structured as follows: We present Triva, an adaptive algorithm for event-based WSNs in Section 5.2. We explain our experimental setup in Section 5.3, and discuss our results in Section 5.4. Finally, we conclude the chapter in Section 5.5.

5.2 Triva Algorithm

In this section, we explain our proposed algorithm, *Triva*, and subsequently give a formal description of its working.

Triva is a code maintenance algorithm, as part of a code dissemination protocol intended specifically for event-based wireless sensor networks. It works in such a way so as to enable nodes to update their code quickly, very much like Trickle. However, unlike Trickle, it does not consume much energy in steady state due to communication. Specifically, it consumes little energy, like Varuna, when there is no new code in the network. Further, when there unidirectional links exist in the network, Varuna's energy efficiency, due to message overhead, *drops drastically* as redundant message transmissions are necessary. *Triva* tries to address the unidirectional link problem by making use of other neighbours, circumventing the problem when a relevant neighbour cannot be reached.

Informally, our protocol leverages the working of both Trickle and Varuna to achieve efficient dissemination and works in the following way: When a node n_1 updates its code, it tries to quickly disseminate the code to its neighbours. n_1 broadcasts advertisement messages at a random time in given period, as in Trickle. If, during these transmissions, a neighbouring node n_2 requests the new code, n_1 sends the new code to n_2. However, unlike in Trickle, if n_1 receives an advertisement message with the same version number from n_2, it saves n_2's *ID* in its neighbourhood table. After broadcasting advertisement messages for some time, the node stops broadcasting and acts like in Varuna in the steady phase to save energy. In *Triva*, there is no concern with selecting an upper bound value τ_h (see Section 2.1.1.1) as in Trickle, because in *Triva* a node sends advertisement messages only for a short period after the node has updated its code.

Specifically, we extend Trickle with some Varuna-like variables, such as neighbourhood table to reduce the broadcasting of advertisement messages. We call the resulting algorithm Trickle$'$. Then, in steady state, Triva behaves as Varuna, while in the dissemination phase, it behaves like Trickle. Further, we extend the working of Varuna to handle asymmetric links, through HELP packets (see Figure 5.3). We call the extended version Varuna$'$.

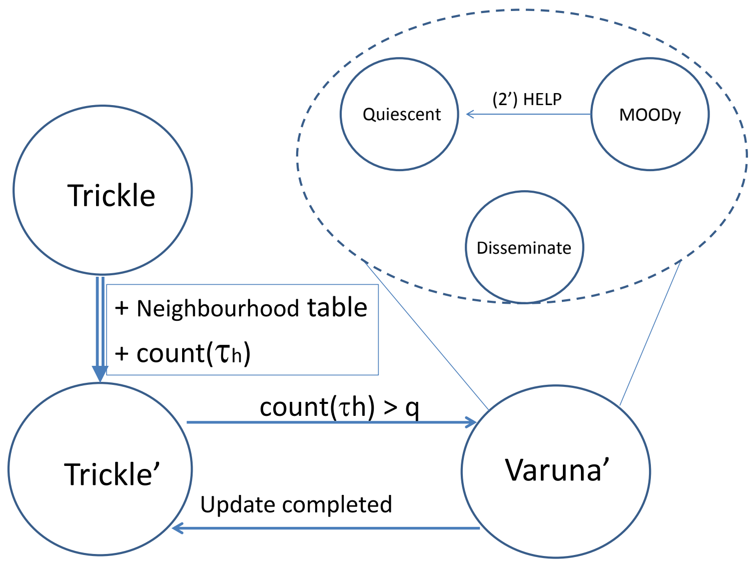


Figure 5.3: The state machine for Triva, which is a combination of Trickle$'$ and Varuna, where Trickle$'$ is obtained by adding a *Neighbourhood* table and variable *count*(τ_h) to Trickle.

5.2.1 Formal protocol description

Figure 5.3 illustrates the state machine of our protocol, which we now detail.

- **Trickle$'$ state:**

  When a node n_1 enters this state, it sets *counter*, which is a Trickle variable, to 0, and sets τ to τ_l. In this state, n_1 sends an advertisement message periodically at a random time between $[\tau/2,\ \tau]$, if it has not heard k advertisement messages about the same version number, i.e., if *counter*$< k$, otherwise, it doubles τ up to τ_h.

  - If node n_1's τ becomes τ_h, *count*(τ_h) is incremented by one.
  - If a node n_1 receives an advertisement message from a node n_2, n_1 compares the received version number with its own. If the version number of n_2 is bigger, then n_1 requests the code after a randomly chosen time between $[0, R]$ seconds, then downloads the code, and updates from n_2.

If the version number of n_2 is smaller, then n_1 broadcasts advertisement messages. If the version numbers are equal and if n_2 does not exist in its neighbourhood table, n_1 adds n_2 in its table, then doubles τ up to τ_h' and increments *counter* by one.

- If a node n_1 receives a request message from n_2, n_1 sends the new code to n_2.
- If a node n_1 updates its code from n_2, it first clears its neighbourhood table and then adds n_2 in its table. It sets *counter* to 0, sets $count(\tau_h)$ to 0 and sets τ to τ_l.
- If, during τ time, n_1 does not send any advertisement message, it doubles τ up to τ_h. If τ is already equal to τ_h, it increments $count(\tau_h)$ by one and sets *counter* to 0.
- If $count(\tau_h) > q$, n_1 sets *counter* to 0, $count(\tau_h)$ to 0 and goes to *Varuna*$'$ state.

• **Varuna$'$:**

- If a node n_1 updates its code from a node n_2, it clears its neighbourhood table and adds n_2 in its table, and then it goes to Trickle$'$ state.
- If a node n_3 receives an advertisement message destined to n_2 from a node n_1, and if n_2 is in the neighbourhood table of n_3 and if n_1's version number is equal to n_3's version number, then n_3 sends a *HELP* packet (see Figure 5.3), which includes the ID of n_2, to n_1 with a probability P after random time between [0, 1] second.
- If a node n_1 receives a *HELP* packet that is destined to itself, then n_1 extracts the node ID n_2 from the packet and adds it into its neighbourhood table if n_2 does not exist in the table (see Figure 5.4).

  - Otherwise, Varuna′ behaves as original Varuna.

- **Download state:**
  - After sending a new code, a node returns to the previous state.
  - After downloading a new code, a node goes to the Trickle′ state.

5.2.2 Fast dissemination

In Triva, every node tries to quickly disseminate its code whenever it receives a new code. It does so by broadcasting advertisement packet periodically between $[\tau/2,\tau]$. Since a node broadcasts only a limited number of advertisement packets, τ can be a small number. Therefore, the amount of energy spent sending advertisement messages is bounded, unlike in Trickle. However, it is important to note that as in Triva a node advertises periodically only limited amount of time after it updates, some neighbours may not receive the advertisement messages because of reasons such as transient link/node failures. In this case, the node may not get the update quickly, but updates only after it sends an application message to an updated node. Nevertheless, Triva's performance will still be better than Varuna, as only some nodes will have transient link/node failures.

5.2.3 Constant energy consumption

In Triva, a node eventually fills its neighbourhood table with the IDs of all of its neighbours, which means that all of its neighbouring nodes have received the new code. Therefore, like in Varuna, in this steady phase, there is no advertisement message transmission, thereby limiting the energy expenditure due to advertisement broadcasts.

5.2.4 The impact of asymmetric links on Varuna and Trickle

Trickle will not be affected by the presence of asymmetric links as, in Trickle, nodes independently send advertisement packets periodically.

In the case of Varuna, using the example in Figure 5.4, n_1 will send advertisement packets i times periodically until it gets a response from n_2. n_1 does this every time it receives any data packet from n_2. Therefore, in Varuna, a node transmits i ADV packets periodically, whenever it gets a data packet and there is no uplink. Thus, if the nodes in the network send packets periodically every T time, then each node transmits $O(A * i)$ advertisement packets even when there is no new version in the network. Here, A is the number of unidirectional links (downlinks) between a node and its neighbours. Therefore, Varuna keeps sending ADV packets even though the version numbers are consistent.

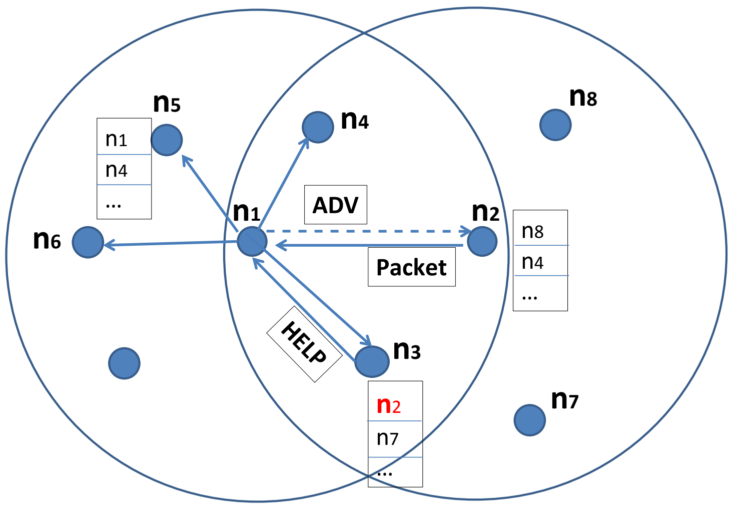


Figure 5.4: Triva: n_3 is addressing the asymmetric link between n_1 and n_2. There exists only one link between n_1 and n_2: the link from n_2 to n_1

5.2.5 Addressing asymmetric/unidirectional links

Unlike Varuna, in Triva, a node tries to help two of its neighbouring nodes that may have an asymmetric link between them. It does so by informing them about their code consistency if the node's code is consistent with those two nodes. Figure 5.4 depicts the Triva process of addressing the unidirectional links between n_1 and n_2: Node n_1, after receiving any packet from n_2, checks if n_2 exists in its neighbourhood table. If not, then n_1 sends an advertisement message to n_2. However, n_2 will not respond as it cannot hear packets from n_1 (due to asymmetric link). If n_3 and n_4 can overhear messages between the nodes n_1 and n_2, then they can help n_1 receive information about n_2's version number, as long as their codes are consistent with both n_1's and n_2's code. They do it by sending a *HELP* packet, which contains the ID of n_2, to n_1. However, as there may be several nodes that can hear both n_1 and n_2, to minimize the number of redundant *HELP* packets, they send the *HELP* packet with a probability P.

In addressing the asymmetric link problem, we assume that the two nodes which have asymmetric links between them share a common neighbour. If there is no such a node, then Triva will not send *HELP* packets, thereby making no message overhead.

5.3 Experimental Setup

To evaluate Triva we perform simulation experiments. Further, to confirm the simulation results we perform real-world testbed experiments on two different testbeds.

We have run Triva on Indriya [36] and TWIST [43] testbeds which have been deployed at the National University of Singapore and Technische Universität Berlin, respectively. In Indriya there are about 100 nodes and in TWIST there are about 90 nodes. In both, the nodes are placed over three floors. Both testbeds have

TelosB motes with C2420 radio, 8 MHz CPU, 10 KB RAM and 48 KB of program memory. On Indriya, each experiment took 20-25 minutes. In total, we have run about 520 experiments. So, all experiments run on Indriya took about 170 hours. The experiments were conducted at different times of the day and on different days of the week. Whereas, the experiments in TWIST were run at random days of a month depending on the availability of testbed. We have run 5 experiments. Each experiment of them took slightly more than 24 hours. So, all experiments run on TWIST took about 120 hours. In both, Indriya and TWIST, in all experiments, the transmission power was set to default value 31 and the used channel was 26. In our experiments, we assume that nodes are not duty cycled. For each experiment the transmitted packets and latency were logged into a different file through USB for analysis purposes.

Our implementation of Triva takes 989 bytes in memory, which includes two tables, each of size 20 entries of 2 bytes, and other algorithm related variables.

For the simulations we used TOSSIM [78] simulator on a 20x20 grid network. In WSNs, the grid topology is usually used to monitor (cover) a given area with the minimum number of nodes [24, 118]. For example, the grid topology has been used in intrusion detection and target tracking applications [11, 38]. However, we do not assume any specific network topology. Because, as Triva is based on Trickle and Varuna, and as these algorithms do not depend on the network topology, Triva is also independent of network topology. (See Section 5.4.3 where the experiments are run on two different testbeds with different 3D network topologies). The only implicit assumption, as in Trickle and Varuna, is that the node in the network should have at least one bidirectional path between itself and the sink to be able to detect inconsistency and download the new code. Without this assumption it is impossible for a node to download the new code. Further, as Triva algorithm is local, i.e., nodes exchange messages with neighbours only, the algorithm is scalable and can run on different sized networks.

Triva, as Varuna and Trickle, by itself tolerates node crashes and message losses as long as the network is connected. However, crashes and message losses may add extra performance burden. For example, the latency will increase in a case when an outdated node requests the new code from an updated node and then the updated node crashes. In this case, the outdated node will not download the new code, however, after some time it will detect inconsistency when exchanging messages with another node. So, this will increase the latency.

We used a network topology generator tool given on tinyos.net to construct the network. We set the distance between neighbouring nodes to 10 feet. By appropriately choosing the power decay value for the reference distance, we constructed the network such that a node has a communication radius of around 30 feet. Each node is given a noise model from the "casino-lab" noise trace file, which is real noise trace taken in the Casino Lab of Colorado School of Mines. The file itself can be found in Tossim/noise folder. TOSSIM takes a noise trace as input to generate a model that can capture bursts of interference and other correlated phenomena to improve the quality of the RF simulation [2].

q	15	τ_h'	20 sec	T_{MOODY}	60 sec	DIS_RAND	2 sec
τ_l	1 sec	k	2	τ_v	8 sec	ADV_RAND	2 sec
τ_h	60 sec	b	1, 30, 60	R	1 sec	P	0.1-0.5

Table 5.1: Parameters for the experiments

The parameter values used in our experiments are given in Table 5.1, while all other parameter values used in Triva are the same as in Trickle and Varuna.

The value of τ_h' can be set to the value of τ_h. However, since Triva runs Trickle for a short period, we keep the value of τ_h' small. τ_h is the parameter used in Tickle. As discussed in Section 2.1.1.1, if we increase this value, the latency will decrease, however, the number of transmitted ADV packets will be smaller. The value of q

can be set according to how fast a node's neighbourhood table can be filled with its neighbouring nodes' *IDs*. If the value of q is set to a large value then, Triva could transmit unnecessary packets. The experiments show that the table can be filled with almost all ($\geq 90\%$) neighbouring nodes' *IDs* when q is between 15 to 20. The parameter τ_v is τ used in Varuna. Setting the probability P to a proper value is important as unnecessary packets could be transmitted. P can be set according to the density of the network. If the network is dense, P can be set to a small value. If the network is sparse, then P can be set to larger values. In our experiments, we varied the value of P between 0.1 and 0.5.

According to real world deployments such as [98, 97, 112, 73], the traffic in WSNs varies. For example, in [98], nodes sense sample their sensor and send their readings to the sink every 70 seconds, while in [73] every 10 minutes. In [97], nodes send every 4 hours, while in [112], nodes send every 5 minutes. Therefore, in our experiments we varied traffic and evaluated three scenarios:

1. *Periodic traffic* : Each node periodically sends data packet, with the period randomly selected between $[0 \ldots 1]$ minute.

2. *Event-based traffic* : Each node sends only one packet at a randomly selected time between $[\lambda, \lambda + 1]$ minute, where λ is the time the event occurred. Events occur every λ minutes.

3. *Event-based bursty traffic* [122] : Each node sends b packets, one packet per second, after a randomly selected time between $[\lambda, \lambda + 1]$ minute, where λ is the time the event occurred. Events occur every λ minutes.

To evaluate the performance of the three protocols on networks with different link symmetry, we simulated them on a network (i) with symmetric links and (ii) with asymmetric links.

In our experiments, all nodes boot randomly in the first minute and a packet with a new version is injected into the top-left node (the sink) after 2 minutes. We also assume that Triva is in the Varuna′ state (see Figure 5.3) at the beginning of execution of the experiments.

5.4 Simulation and Testbed Results

In this section, we show the simulation and testbed results of Triva, Varuna and Trickle in terms of different metrics.

5.4.1 Metrics

In our simulations, we used the following metrics: (i) the number of advertisement (and *HELP* relevant for Triva) packets transmitted, as the number of transmissions is directly related to energy consumption in the steady state [95, 79], (ii) the number of discarded application packets (relevant for Varuna) from the time when the new version is injected, as this metric captures the amount of resources wasted as bandwidth and energy, and (iii) completion time (dissemination latency), which is another important criteria in code dissemination, to enable the network to work efficiently. We counted the number of ADV packets by logging transmitted ADV packets into a file in simulations and by sending the same packets to USB port in testbed experiments.

Note that, in general, radios can operate in four modes of operation: Transmit, Receive, Idle and Sleep. Among these, in most cases, the Transmit operation consumes more energy than other operations. The Receive and Idle mode operations may also consume considerable amount of energy, and they consume almost equal energy [99]. Therefore, although the first metric, the number of advertisement packets, does not define the total energy cost and may not have a significant fraction of total energy cost when the radio is always turned on (duty cycle is 100%), considerable amount of energy can still be conserved due to less packet transmissions. As mentioned earlier, in the experiments, we assume that nodes are always on. As most of the time Triva behaves like Varuna (in the steady state), if the nodes are duty cycled, then the same amount of energy could be saved, however, in this case, the latency will be higher [95].

5.4.2 Simulation Results

Number of Advertisement Packets Figures 5.5 and 5.6 show the number of advertisement packets transmitted during 15 hours. Figure 5.5 shows the number of advertisement packets where a node periodically sends one data packet, with the period randomly chosen between $[0 \ldots 1]$ minute in the network with symmetric links. As can be observed, the number of advertisement packets Trickle sends increases linearly with time, even when there is no new code in the network, while the amount of transmitted advertisement messages in Varuna decreases eventually. Varuna decreases its number of transmissions eventually, because according to the Varuna protocol, in Varuna, a node keeps sending advertisement packets until all its neighbouring nodes' IDs are stored in its neighbouring table. And, as the links in the network were not perfectly symmetric, Varuna kept sending advertisement packets (see Section 5.2.4 for the impact of asymmetric links on Varuna). As it can be observed from Figure 5.5, in Triva, the transmission of advertisement and *HELP* packets stops after some time and never transmitted again (unlike Varuna), as long as there is no new code in the network. The reason for why Triva stopped sending advertisement and *HELP* packets is that all nodes in the network have stored most of their neighbours' IDs when the nodes were in the Trickle state and have stored the remaining nodes' IDs, i.e., the IDs of nodes that had asymmetric links in between, with the help of *HELP* packets (see Section 5.2.5 for how Triva addresses the problem of asymmetric links). In the network with symmetric links, Triva sends advertisement and *HELP* packets 10 times less than Trickle and 3 times less than Varuna.

Figure 5.6 shows the number of advertisement packets transmitted in the network with high asymmetric links. Here, we can also see that Triva transmits advertisement packets 10 times less than Trickle and 7 times less than Varuna.

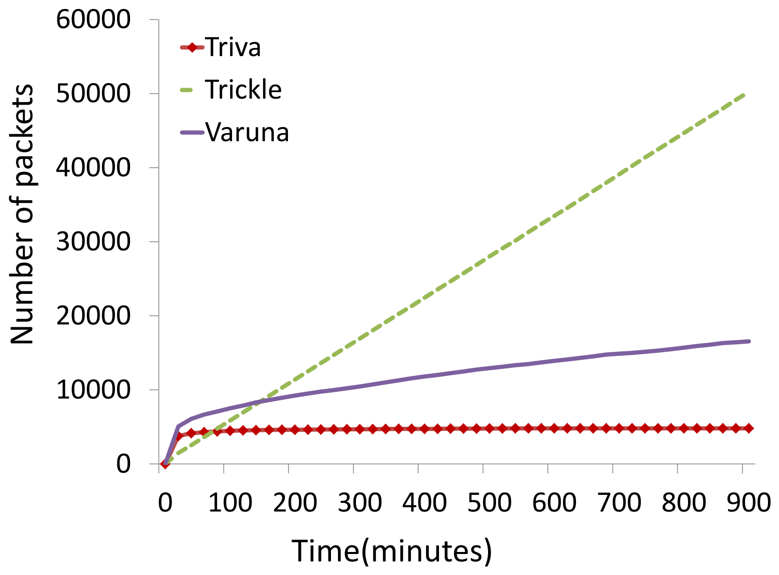


Figure 5.5: Scenario 1 - **Periodic Traffic, Symmetric Links**: Data packets generated randomly every $0 \dots 1$ minute for periodic traffic

In Figure 5.7, Triva and Varuna are compared in terms of the number of advertisement packets sent, when each node in the network sends 30 packets back-to-back at 1 packet/second, with $\lambda = 5$ minutes. In this figure, because of the reasons as mentioned above, Triva again stops transmitting advertisement packets after some time and performs better than Varuna.

Overall, the figures 5.5, 5.6 and 5.7 show that Triva outperforms Trickle and Varuna in terms of number of transmitted packets, thereby energy consumption.

Number of Discarded Application Packets Figures 5.8 and 5.9 show the

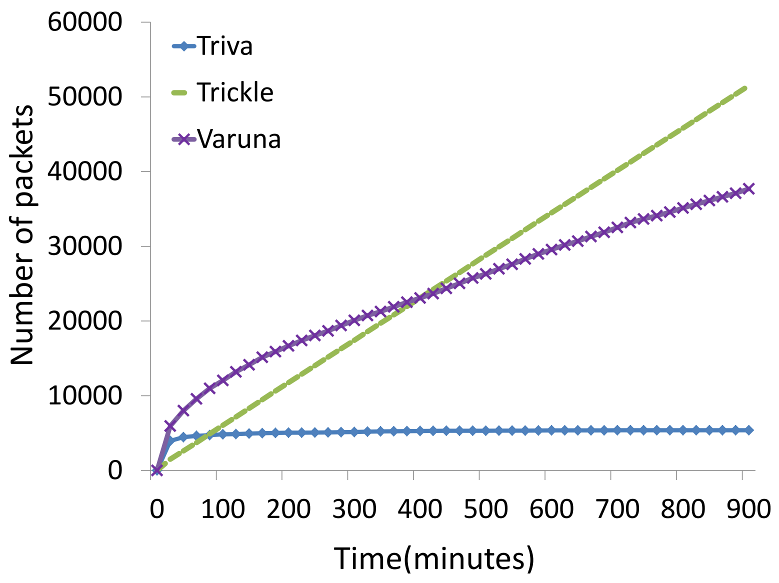


Figure 5.6: Scenario 1 - **Periodic Traffic, Asymmetric Links**: Data packets generated randomly every $0 \ldots 1$ minute for periodic traffic.

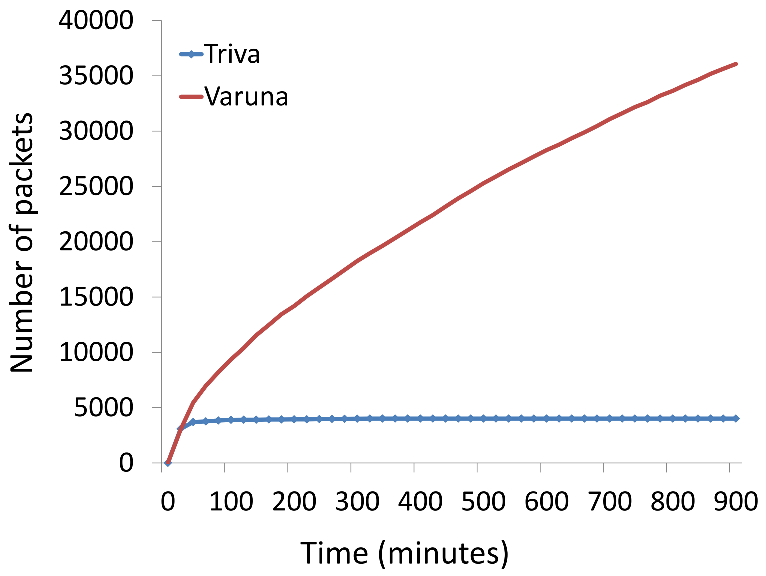


Figure 5.7: Scenario 2 - **Bursty Traffic**: 30 packets are sent at 1 packet/second, $\lambda = 5$ minutes for bursty traffic.

number of discarded application data packets after the time when a packet with a new version is injected, i.e., after 2 minutes. Figure 5.8 shows the values obtained

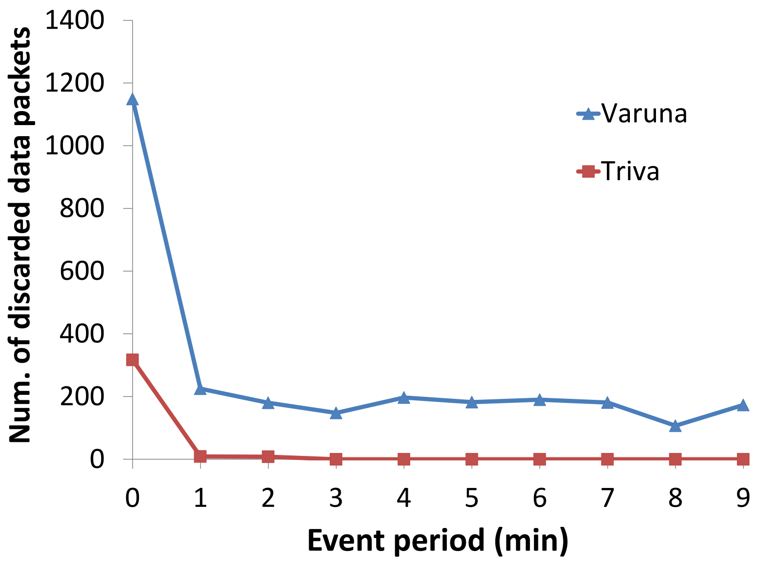


Figure 5.8: Scenario 3 - **Event-Based Traffic**: (λ, λ+1) minutes, λ represents x-axis, Data packets: One data packet is sent for every event.

from the network in which a node periodically sends only one, i.e, b=1, data packet at randomly selected time between $[\lambda, \lambda + 1]$ minutes. In both Triva and Varuna, values are relatively high when λ=0, as the randomly selected time by a node could be very small such as 150 ms, which forces nodes to send more data packets in a small amount of time.

In Figure 5.9, a node sends b=60 packets back-to-back every 1 second. The number of data packets discarded by Varuna remains constant, with increasing idle time. On the other hand, the number of dropped packets in Triva is very low, due to the fact that the nodes quickly obtain the updated code.

Overall, when the number of discarded data packets is low, it says that the energy and bandwidth are not wasted and also it captures the fact that all nodes have the same code version, i.e., code dissemination has completed quickly.

Code Dissemination Latency Figure 5.10 shows the time taken from the point when a packet with new version is injected to the point when the last node in a

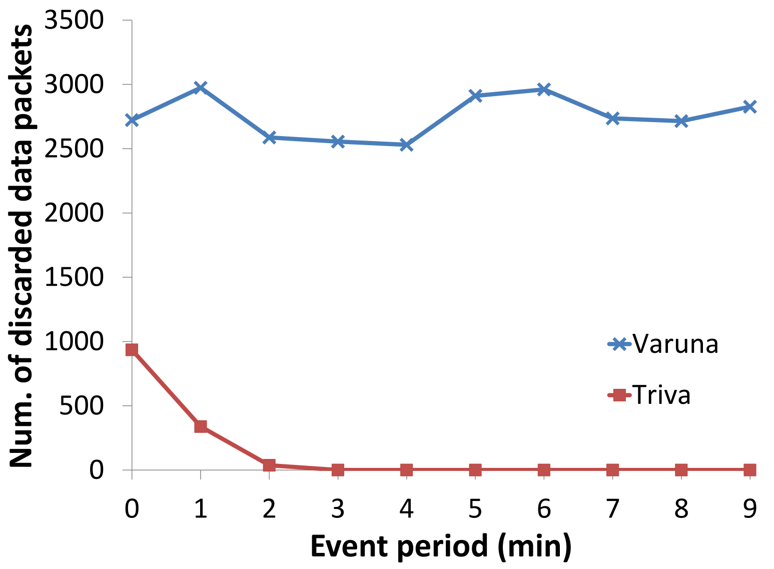


Figure 5.9: Scenario 4: **Event-Based Bursty Traffic**: (λ, λ+1) minutes, λ represents x-axis, Data packets: 60 packets are sent at 1 packet/second.

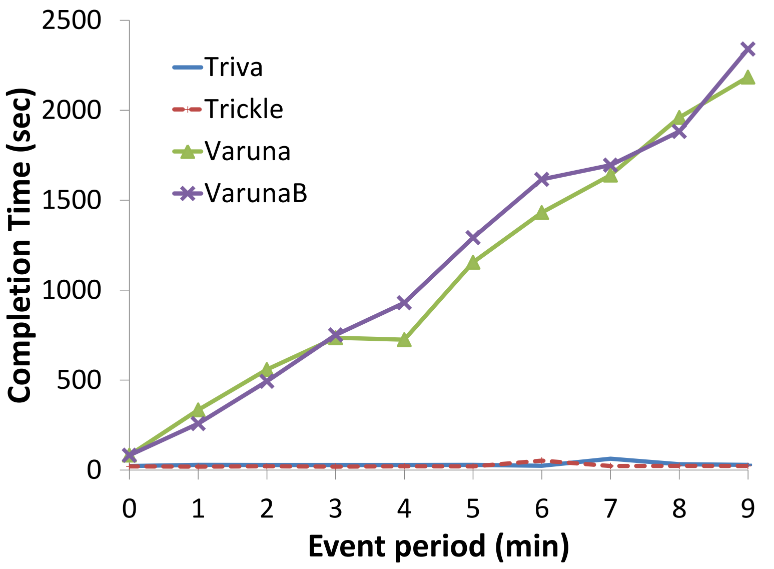


Figure 5.10: Scenarios 6 and 7: Completion time as a function of λ, the event period. Scenario 1: Data packets generated every $(0 \ldots 1)$ minute, Scenario 3: 30 packets per event at 1 packet/second.

network receives that packet, i.e., the code dissemination latency. We investigated the latency under two scenarios: (i) periodic traffic and ii) bursty traffic. Triva and Trickle have very low dissemination latency (for both scenarios), whereas Varuna has increasing latency with increasing idle time.

5.4.3 Testbed Results

Now to confirm the results obtained from the simulations, we present the results obtained from the Indriya and TWIST testbeds. We have run the protocols on Indriya to check the *dissemination latency*, *number of discarded packets*, and on TWIST to check the *number of advertisement packets.* The used parameters are same as in simulation experiments except that in testbed experiments a new code was injected after 4 minutes.

Code Dissemination Latency Figures 5.11 and 5.12 show the code dissemination latency under two scenarios: (i) periodic traffic and ii) bursty traffic. As the results obtained from simulations, the latency of Varuna increases linearly as a function of event period, while Triva shows constant latency independent of event period.

Number of Discarded Application Packets Likewise, Figures 5.13 and 5.14 show the number of discarded packets of Triva and Varuna under two scenarios: (i) periodic traffic and ii) bursty traffic. As can be observed, Varuna transmits redundant packets proportional to traffic. On the other hand, the number of discarded packets in Triva is negligible.

The above results confirm that Triva outperforms Varuna and works best in event-based WSN applications or in applications where data collection is done with large periodicity.

Number of Advertisement Packets To compare the number advertisement packets (and *HELP* packets relevant for Triva), we have run each of the algorithms on TWIST for 24 hours. As we have mentioned above, the number of *HELP* packets

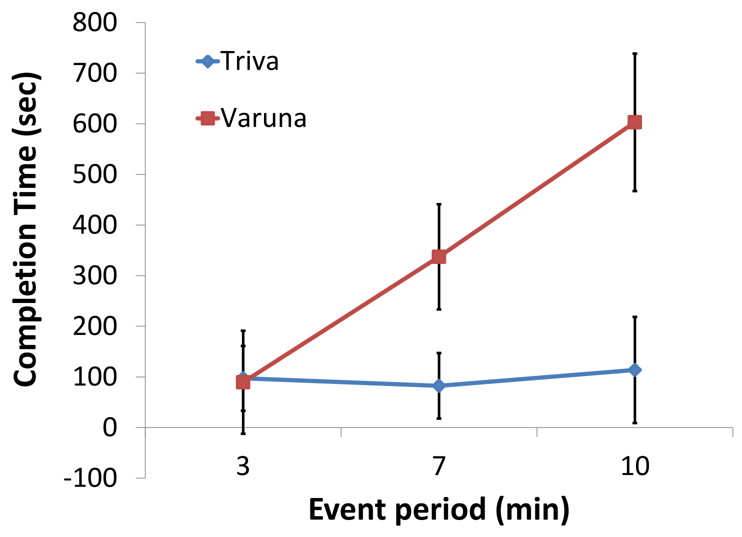


Figure 5.11: Scenario 3 - **Event-Based Traffic**: (λ, λ+1) minutes, λ represents x-axis, Data packets: One data packet is sent for every event.

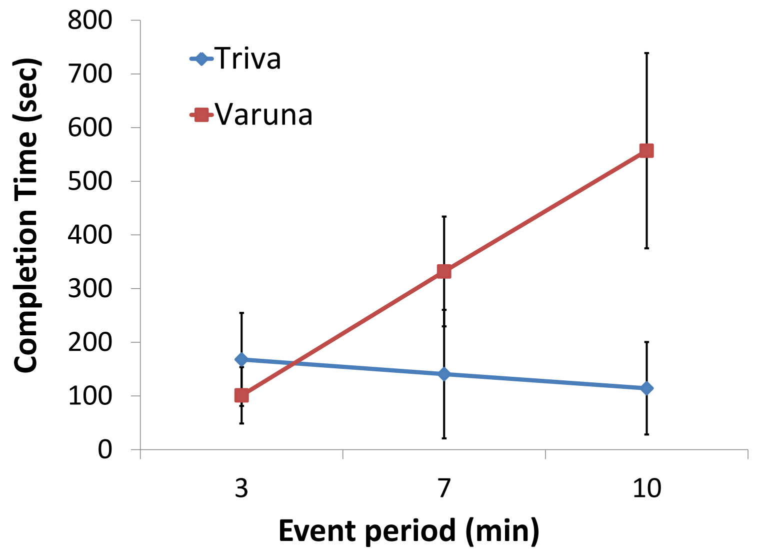


Figure 5.12: Scenario 4: **Event-Based Bursty Traffic**: (λ, λ+1) minutes, λ represents x-axis, Data packets: 20 packets are sent at 1 packet/second for every event.

transmitted depends on P. So, we varied the value of P to different values 0.1, 0.2 and 0.3. Section 5.4.4 discusses about the relation between P and the number of

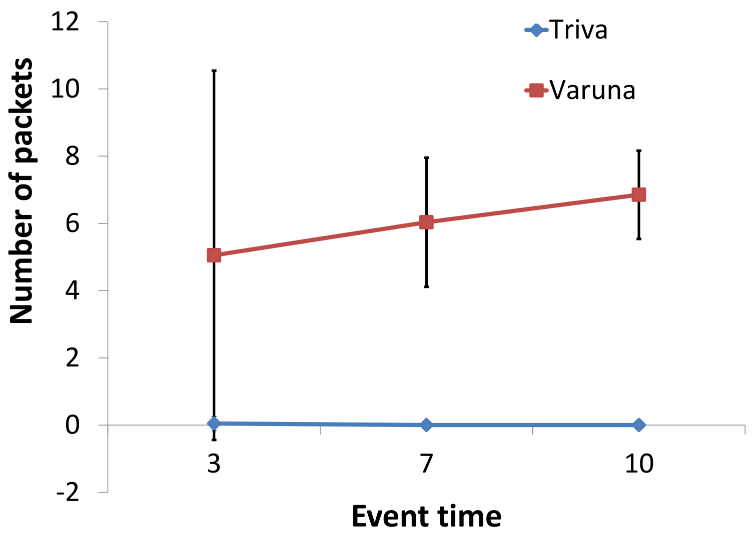


Figure 5.13: **Number of discarded data packets, Event-Based Bursty Traffic**: (λ, λ+1) minutes, λ represents x-axis, One data packet is sent for every event.

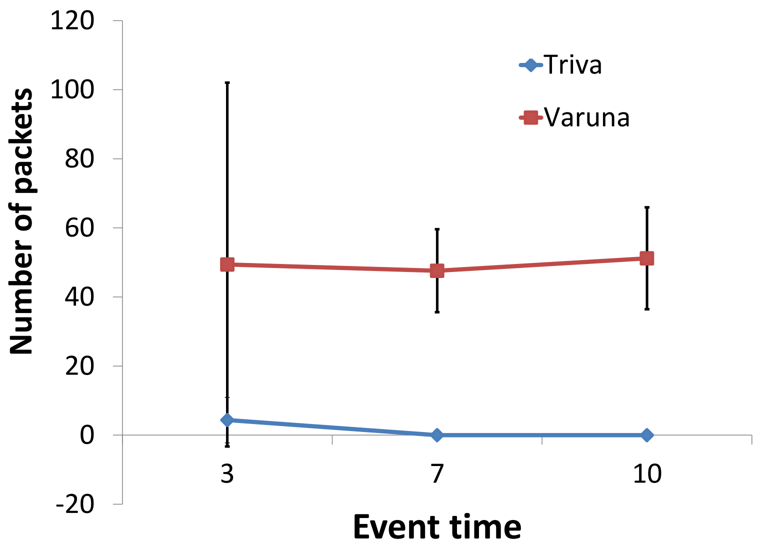


Figure 5.14: Scenario 4: **Number of discarded data packets, Event-Based Bursty Traffic**: (λ, λ+1) minutes, λ represents x-axis, 20 packets are sent at 1 packet/second.

neighbours, i.e., the network density.

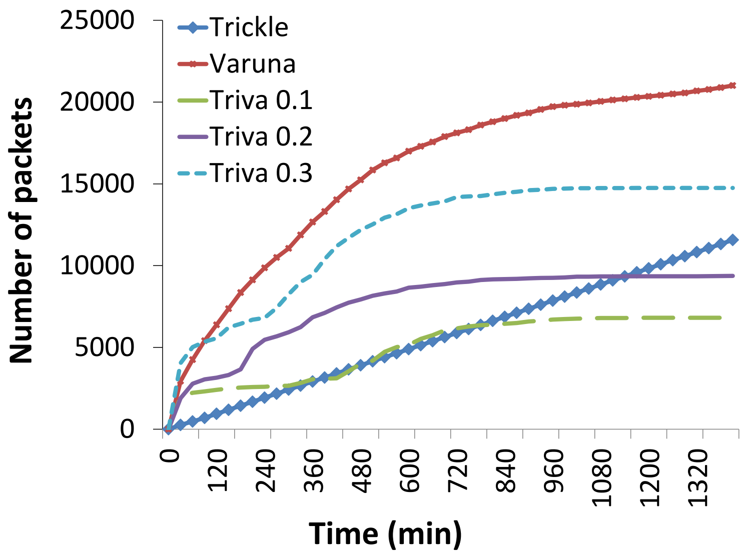


Figure 5.15: Scenario 1 - **Periodic Traffic**: Data packets generated randomly every $0 \dots 1$ minute for periodic traffic. Different values for P (0.1, 0.2 and 0.3) are used in Triva

Figure 5.15 shows the number of advertisement packets transmitted by Triva, Trickle and Varuna. As expected, the number advertisement packets transmitted by Trickle increases linearly with time. Varuna has the largest number of packets during 24 hours, however, as can be observed, the function decreases eventually and after about 34 hours it could be less than that of Trickle. While Triva with $P = 0.1$ and $P = 0.2$ has the smallest number of packets after 24 hours. The impact of P on the number of transmitted *HELP* packets will be discussed in the next section.

Overall, these results, as in the simulation results, show that Triva outperforms Trickle and Varuna in terms of number of transmitted packets, thereby energy conservation.

5.4.4 What should the value of P be?

In Triva, as mentioned in Section 5.2.5, sending *HELP* packets depends on the probability P. And this value affects the message overhead of Triva. If P is set to 1, all nodes in the neighbourhood that are ready to send the *HELP* packet will send it. However, this may cause not only a lot of message overhead, but also collisions which wastes resources such as energy [70]. Conversely, if P is set to a very small value, then nodes might not send *HELP* packets and the asymmetric link problem may not be solved for a time until one of the nodes sends a *HELP* packet. Thus, there is a tradeoff between the message overhead and latency.

For simplistic assumptions, like a circular communication range and uniformly distributed networks, the value of P could be approximated as following. Let k be the number of neighbours of a node. Let n_1 and n_2 be neighbours of each other. Then, if the coverage area of a node is Πr^2, then the intersection of coverage areas of n_1 and n_2 is $2\Pi r^2/3 - r^2\sqrt{3}/2$ (see Figure 5.16 for illustration). As there are k nodes in Πr^2, there are around $k/3$ nodes in $2\Pi r^2/3 - r^2\sqrt{3}/2$. Therefore, if there exists an asymmetric link between n_1 and n_2, the neighbours of both should set their P to a number between 1 and $3/q$, i.e., $1 \geq P \geq 3/k$. For example, if the number of neighbours of a node is $k \leq 3$, then P should be set to 1; if $k = 30$, then $P \geq 0.1$.

One way to relax the above assumptions is to have nodes to broadcast their neighbours' IDs so that each node could identify the number of common nodes (the intersection area) for each pair of its neighbours. And according to that number, nodes could define their P. However, it needs extra number of packet transmissions and memory to store the number of common nodes for each pair of neighbours. Moreover, this way will not work well for time-varying topologies where nodes may crash, join or disjoin the network.

To find P for time-varying topologies, nodes should exchange messages periodically or when there is a change in their neighbourhood. In WSNs, such a technique

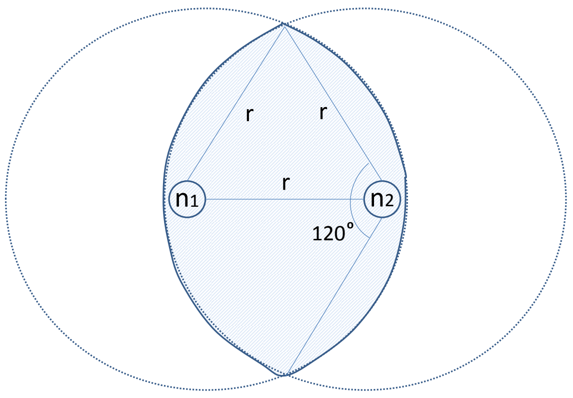


Figure 5.16: Computing an approximate value of P. The area of the shaded region is equal to $2\Pi r^2/3 - r^2\sqrt{3}/2$

is usually used for different objectives. For example, in [70], this technique is used to select a single sender in a neighbourhood at a time. So, to find the value of P for time-varying topologies, like the above method, first, nodes should broadcast their neighbours' IDs to find the initial value of P, and then whenever a node detects any change in its neighbourhood, the node informs its neighbours about the change so that its neighbours could update their P accordingly. However, as mentioned above, this is costly and may not be practical, especially for resource constrained wireless sensor nodes. Also, while trying to reduce message overhead, this method, conversely, may increase it.

A simpler and more practical solution could be, instead of computing the value of P according to topology changes, to set P to a constant value and apply a basic suppression method, as in Trickle [79], to reduce the number of redundant *HELP* packet transmissions. That is, if node n_1 hears node n_2 transmitting a *HELP* packet to node n_3, then there is no need for n_1 to transmit a *HELP* packet to n_3. Otherwise, if n_1 does not hear any *HELP* packet, then n_1 sends a *HELP* packet to n_3 with probability P. This solution will greatly reduce the message overhead, even when the network is dense and $P = 1$, as most of the neighbours will suppress their transmissions. But then the question is how many redundant *HELP* packets will be sent if $P = 1$? The answer is only few. According to the simulation results reported in [79], if we apply the above suppression method, in a 1-hop network of sizes 8, 64 and 256, then the number of transmissions is 2, 3 and 4 with 0% packet loss rate, and 4, 7 and 10 with 60% packet loss rate, respectively. This is because, based on the simple geometric observation, there could be at most 5 nodes that may not be connected with each other if the communication radius of nodes is same [121]. Therefore, the network topology will not affect much. To reduce the message overhead further, we can set P to a smaller constant value such as 0.5, which might slightly increase the time to solve the asymmetric problem.

In general, for any network topology, setting P to a small value bigger than 0, $P > 0$, will still lead to a better performance as it overcomes the problem of asymmetric links. However, it may take more time to solve the problem. Setting $P = 1$ may generate lots of additional traffic as long as there exist asymmetric links; however, when all nodes solve the asymmetric link problem, Triva does not send HELP/ADV packets anymore. We think that this is reasonable if the number of asymmetric links is large and they are permanent, though more transmission energy may be used (see Figure 5.17).

We use the casino lab noise trace to make the network more realistic. However, the casino lab file has a high SiNR (Signal-to-Noise Ratio). The impact of a low SiNR, such as heavy-meyer trace file has to be determined. Figure 5.18 shows the impact of a signal with high SiNR 5.18(a) and low SiNR 5.18(b). As can be observed, the number of transmitted packets with low SiNR is more than that of with high SiNR.

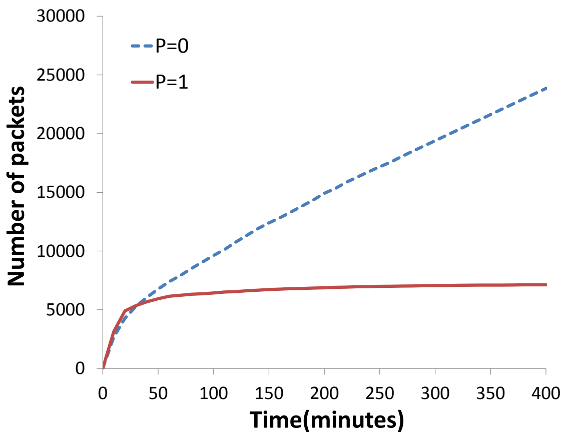
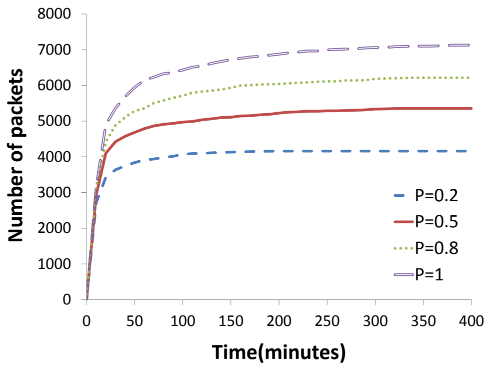
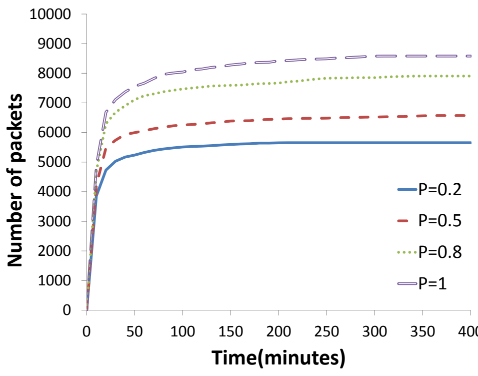


Figure 5.17: The number of transmitted ADV and HELP packets when P=0 and P=1



(a) The number of packets with varying P and high a SiNR



(b) The number of packets with varying P and low a SiNR

Figure 5.18: The impact of P and SiNR

5.5 Conclusion

In this chapter, we have proposed an algorithm that can be used to maintain code updates in a wireless sensor networks. Code update maintenance algorithms are necessary as not all nodes may update to a new code during code dissemination phase due to reasons such as transient link failures. The existing algorithms of this kind are efficient in terms of different metrics such as latency and energy. However, they have drawbacks as well.

The algorithms, namely Trickle and Varuna, have drawbacks in terms of energy and latency, respectively. In the presence of asymmetric links the latter does not consume constant energy as expected. The proposed code maintenance algorithm in this chapter, called Triva, tries to address the drawbacks of the algorithms by leveraging the properties of both. It adapts both Trickle and Varuna to achieve energy efficiency and low dissemination latency in event-based sensor networks and in the presence of asymmetric links. The results of performed simulation and real-world testbed experiments show that Triva outperforms both Trickle and Varuna (i) in periodic traffic, (ii) event-based traffic, (iii) bursty traffic, and (iv) in networks with asymmetric links.

To address the asymmetric link problem, which Varuna does not solve, Triva uses additional packets called *HELP*. However, *HELP* packets are sent only when there exist asymmetric links in the network, thereby reducing the message overhead. The experiment results showed that by solving the asymmetric link problem Triva stops sending ADV packets when there is no new code in the network, which is desirable especially when updates occur rarely.

All in all, although Triva, like Trickle and Varuna, may not work with dissemination protocols that have protocol specific mechanisms such as sender selector in MNP, the mentioned algorithms are orthogonal to many code dissemination algorithms and thus can be integrated to any of them to maintain code updates. Therefore, Triva is an algorithm that could be integrated to any compatible dissemination protocols to save more energy.

CHAPTER 6

Data Aggregation Scheduling with Two Sinks

6.1 Introduction

Traditionally, WSNs have been deployed with a single sink [6]. However, there are several reasons that limit the usefulness of a single sink, for example (i) the emergence of more sophisticated applications [90] and (ii) fault tolerance issues [76, 113, 103]. Two scenarios are hereby provided:

Application scenarios: WSNs are increasingly being used to control several actuators embedded in the environment [5]. In these situations, the application requires that data sensed from multiple sources is delivered to multiple sinks (i.e., the actuators). As a matter of example, a decentralized building automation system [34] can provide functionality such as heating, ventilation, and air conditioning along with fire alert. The actuator nodes embedded in the environment may include, inter-alia, air conditioning units, water sprinklers and fire alarms. Sensor nodes (e.g., for temperature and humidity) are deployed to support the functionality of the actuators. Often, to meet the application requirements, the sensor nodes need to report to multiple sinks. For example, the same temperature sensor may report to multiple air conditioners. Several such applications have appeared recently [90].

Fault tolerance: Typically, WSNs, once deployed, are left unattended for extended periods of time. During this time, the network can experience a range of faulty scenarios: (i) any sensor node may fail due to energy exhaustion, (ii) links may fail due to interference or (iii) the sink may fail to communicate due to some reasons such as link failures, node and sink failures. For example, in [97], the authors observe that 4 of the correctly working 7 nodes had communication failure with the sink over time, furthermore, they observe sink outage due to power failure. In a deployment [98], a crash of the database running on the sink node resulted in the complete loss of data for two weeks. Likewise, in a deployment [112], two weeks of data were lost due to a sink outage. In [107], authors observed a sink outage due to harsh weather. In such situations, as mentioned, the loss of the sink results in the loss of the network. One way to increase the reliability of such WSNs is to deploy with more than one sink.

In wireless sensor networks, TDMA-based protocols are often used to (i) avoid message collisions and (ii) guarantee timeliness properties. TDMA MAC protocols work by breaking the timeline into slots and assigning those slots to nodes. Each node then can only transmit in a slot it has been assigned. However, most TDMA-based MAC protocols have been developed with a single sink assumption. In a WSN with multiple sinks, such slot assignment will result in a very high latency for one of the sink.

Thus, this implies that there is a need for TDMA-based MAC protocols specifically for WSNs with multiple sinks. There is a dearth of work in this area. As discussed in Chapter 2, some works on routing in multi-sink scenario have been developed [108, 90]. Data aggregation scheduling algorithms have been presented in [59, 21]. However, in the works, the data aggregation scheduling is done from *many* nodes to *one* sink, whereas this chapter considers the data aggregation scheduling from *many* nodes to *many* sinks.

One way to solve the data aggregation scheduling problem is to first develop a backbone that connects sinks and then allocate slots to nodes that connect to the backbone. The problem of developing the backbone, i.e., connecting the sinks is directly related to the problem of developing a Steiner tree [42]. In this case, we are interested in developing a minimum Steiner tree, which is known to be NP-complete [58]. To address the computational complexity of this problem, there are different ways of going about it. One of them is to look at specific instances of the problem that can lead to a polynomial-time solution to the problem. In this case, we focus on the problem with 2 sinks, since the minimum Steiner tree can be computed in polynomial-time, as the minimum Steiner tree is the shortest path between the sinks. However, our proposed algorithm for 2 sinks also works for WSNs with more than 2 sinks, but in sub-optimal ways (see Section 6.4.4). And as a future work (see Chapter 7), we will address the more general problem of n sinks.

In this chapter, we first formalise the problem of Data Aggregation Scheduling (DAS) in a WSN. Then we prove a number of impossibility results, as well as show a lower bound for solving a variant of DAS called weak DAS. Further, we propose two algorithms, called Balancing Tree Formation (BTF) and Energy-Efficient Collision Free (EECF), which taken together, solves weak DAS and results in a schedule that matches the predicted lower bound. Through simulations, we show that our methodology enables a modular design of DAS algorithms, whereby different properties can be enforced. Finally, we perform real-world testbed experiments on Indriya [36] to support the results obtained from the simulation results.

This chapter is structured as follows: We present the problem formulation in Section 6.2. In Section 6.3, we present our theoretical contributions of this chapter. In Section 6.4, we describe a balancing tree and data aggregation convergecast scheduling algorithm. Section 6.5 explains our experimental setup and Section 6.6 discusses the results of the experiments. In Section 6.7 we survey existing balancing tree algorithms and, finally, in Section 6.8 we conclude the chapter.

6.2 Problem Formulation

We present the following definitions that we will use in this chapter.

Definition 13 (Schedule) *A schedule $\mathbb{S} : V \to 2^{\mathbb{N}}$ is a function that maps a node to a set of time slots.*

When scheduling, for a node n to be able to send its data to a sink, n should have at least one neighbour m that sends after node n.

Definition 14 (DAS-label) *Given a network $G = (V, E)$, a sink Δ, a schedule $\mathbb{S}$ and a path $\gamma = n \cdot m \dots \Delta$, we say that* n is DAS-labeled under $\mathbb{S}$ on γ for Δ *if $\exists t \in \mathbb{S}\ (n) \cdot \exists t' \in \mathbb{S}\ (m) : t' > t$.*

We call the node m on γ the Δ-parent of n and γ the DAS-path for n. The number of DAS-paths shows the resilience of the schedule for node failures.

Definition 15 (Strong and Weak schedule) *Given a network $G = (V, E)$, a sink $\Delta \in V$ and a schedule $\mathbb{S}$, $\mathbb{S}$ is said to be a* strong DAS schedule for Δ *for a node $n \in V$ iff $\forall$ path $\gamma_i = n \cdot m_i \dots \Delta$, n is DAS-labeled under $\mathbb{S}$ on γ_i for Δ. $\mathbb{S}$ is a* weak DAS schedule for Δ *for n if $\exists$ path $\gamma = n \cdot m_i \dots \Delta$ such that n is DAS-labeled under $\mathbb{S}$ on γ for Δ.*

A schedule $\mathbb{S}$ is strong DAS (resp. weak DAS) for G iff $\forall n \in V$, $\mathbb{S}$ is strong DAS schedule (resp. weak DAS schedule) for Δ for n.

We will only say a strong or weak schedule whenever Δ is obvious from the context. A strong schedule, in essence, is resilient to problems that occur in the network such as radio links not working or node crashes during deployment. On the other hand, a weak schedule is not resilient and, any problem happening, will entail that a message from node n to m will be lost.

It has been shown in [53] that it is impossible to develop strong schedules.

Given a network with 2 sinks Δ_1, Δ_2, we wish to develop a weak schedule for Δ_1 and Δ_2. There are several possibilities to achieve this. In general, to develop a weak schedule, several works have adopted the approach whereby a tree is first constructed, rooted at the sink, and then slots assigned along the branches to satisfy the data aggregation constraints. A trivial solution is to construct two trees, each rooted at a sink, and then to assign slots to nodes along the trees. This means that nodes can have two slots, i.e., meaning that nodes may have to do two transmissions for the same message. Thus, we seek to reduce the number of slots for nodes to transmit in.

6.2.1 DAS Scheduling

We model our problem as follows:

We capture slots assignment with a set of decision variables.

$$t_n^{\mathbb{S}} = \begin{cases} 1 & t \in \mathbb{S}(n) \\ 0 & \text{otherwise} \end{cases}$$

A set value assignment to these variables represent a possible schedule. The number of slots used, which equates to the number of transmission by nodes, has to be reduced for extending the lifetime of the network. The number of slots used is given by:

$$numSlots_{\mathbb{S}} = \sum_{t \in T, n \in V} t_n^{\mathbb{S}} \tag{6.1}$$

We also capture the number of nodes with multiple slots as follows:

$$f_n^{\mathbb{S}} = \begin{cases} 1 & |\mathbb{S}(n)| > 1 \\ 0 & \text{otherwise} \end{cases}$$

However, such a schedule may not assign a slot to a given node, so we need to rule out some schedules with a constraint:

$$\forall n \in V \cdot \exists t : t_n^{\mathbb{S}} = 1$$

The above constraint means that all nodes in the network will be assigned at least one slot. We also rule out schedules $\mathbb{S}$ that assign the same slot to two nodes that are in the two-hop neighbourhood, i.e,

$$\forall m, n \in V : t_m^{\mathbb{S}} = 1 \wedge t_n^{\mathbb{S}} = 1 \Rightarrow \neg 2HopN(m, n)$$

This can be done by using information about two-hop neighbourhood, and it can be obtained by exchanging messages with neighbours.

Finally, we require to generate weak DAS schedules $\mathbb{S}$, i.e.,

$$\forall m \in V \cdot \exists n \in V, (m \cdot n \ldots \Delta_1) : t_m^{\mathbb{S}} = 1 \Rightarrow \exists \tau > t : \tau_n^{\mathbb{S}} = 1$$

$$\forall m \in V \cdot \exists n \in V, (m \cdot n \ldots \Delta_2) : t_m^{\mathbb{S}} = 1 \Rightarrow \exists \tau > t : \tau_n^{\mathbb{S}} = 1$$

Thus, to generate an energy-efficient collision-free weak DAS schedule for both Δ_1 and Δ_2, there are different possibilities. For example, one may seek to minimise *numSlots* to reduce the number of slots during which nodes transmit. Another possibility is to reduce the number of times any node can transmit, in some sort of load balancing. Thus, we solve the following problem, which we call the *EECF-2-DAS* problem (for energy-efficient collision-free 2-sinks DAS).:

EECF-2-DAS problem: Obtain an $\mathbb{S}$ such that

minimise $\sum_{\forall t} \sum_{\forall n \in V} f_n^{\mathbb{S}}$ subject to

1. $\forall n \in V \cdot \exists t : t_n^{\mathbb{S}} \neq 0$
2. $\forall m \in V \cdot \exists n \in V, (m \cdot n \ldots \Delta_1) : t_m^{\mathbb{S}} = 1 \Rightarrow \exists \tau > t : \tau_n^{\mathbb{S}} = 1$
3. $\forall m \in V \cdot \exists n \in V, (m \cdot n \ldots \Delta_2) : t_m^{\mathbb{S}} = 1 \Rightarrow \exists \tau > t : \tau_n^{\mathbb{S}} = 1$
4. $\forall m, n \in V : t_m^{\mathbb{S}} = 1 \wedge t_n^{\mathbb{S}} = 1 \Rightarrow \neg 2HopN(m, n)$

The EECF-2-DAS problem consists of two subproblems: (i) The first three conditions amount to what we call the *weak DAS problem* and (ii) the fourth condition ensures that any weak DAS schedule is collision-free. Collision freedom is guaranteed by ensuring that no two nodes in a 2-hop neighbourhood share the same slot.

6.3 Theoretical Contributions

In this section, we investigate how small can the number of nodes with multiple slots be, to generate an energy efficient collision-free weak schedule in a network with 2 sinks.

6.3.1 All Nodes Have Multiple Slots ($\sum_{\forall n \in V} f_n^{\mathbb{S}} = |V|$)

A trivial solution to this is as follows: generate two trees, each rooted at a different sink. For sink Δ_1, starting with slot $|V|$, assign, in decreasing order, slots to nodes in using BFS. The process is repeated with the other tree rooted at Δ_2. This sets an upper bound for collision-free weak schedules for wireless sensor networks.

6.3.2 Every Node Has a Single Slot ($\sum_{\forall n \in V} f_n^{\mathbb{S}} = 0$)

In this section, we seek a lower bound on the number of nodes that can have multiple slots assigned to them. As a starting point, we endeavour to determine whether all nodes can have only 1 slot.

Intuition For all nodes to have only one slot, either they have to be directly connected to both sinks or they have to send their values to a node n that is directly connected to both sinks. If there is such n, then it means that there exists a path between two sinks of length 2 hops. If there is no path of length 2, then there should be another node m that should forward n's value. As m has also a value to send via node n, node n should have at least 2 slots. This means that if the shortest path between two sinks is bigger than 2, then there must be at least one node with more than 1 slot. This is captured in Theorem 4.

Theorem 4 (Impossibility of 1 slot) *Given a network $G = (V, E)$ with 2 sinks Δ_1, Δ_2, where the shortest distance between Δ_1, Δ_2 is bigger than 2, then there exists no weak DAS schedule $\mathbb{S}$ for Δ_1, Δ_2 such that $\sum_{\forall n \in V} f_n^{\mathbb{S}} = 0$*

Proof. We assume there is such a weak DAS $\mathbb{S}$ and then show a contradiction.

Assumptions: We assume a network G with 2 sinks Δ_1, Δ_2 such that the distance between Δ_1 and Δ_2 is bigger than 2. We also assume part of the network is as follows: Focusing on the sink Δ_1, there is a set H_1 of nodes in its first hop. There is also a set H_2 of nodes in its second hop. We also denote by n_h^1, the node in

H_1 with the largest slot number. We also assume, for some set of nodes $H_2' \subseteq H_2$, that all nodes in H_2' have n_h^1 as a Δ_1-parent.

Since the schedule is weak DAS, then $\forall n \in H_2 \cdot \exists m \in H_1 : \mathbb{S}(m) > \mathbb{S}(n)$[1]. Also, because the schedule is weak DAS, no node in H_2' can be a Δ_2-parent for n_h^1. Thus, there $\exists \eta \in H_2, \eta \notin H_2'$ such that η is a Δ_2-parent for n_h^1 and, given that $\mathbb{S}$ is a weak DAS schedule, then $\mathbb{S}(\eta) > \mathbb{S}(n_h^1)$.

Now, since $\eta \in H_2$, $\exists m' \in H_1, m' \neq n_h^1$ such that m' is a Δ_1-parent for η and, given that $\mathbb{S}$ is a weak DAS schedule, then $\mathbb{S}(m') > \mathbb{S}(\eta)$. Since we assumed that n_h^1 has the largest slot in H_1, it implies that $\forall m \in H_2 : \mathbb{S}(n_h^1) > \mathbb{S}(m)$. This also means that $\mathbb{S}(\eta) < \mathbb{S}(n_h^1)$, which contradicts the previous conclusion that $\mathbb{S}(\eta) > \mathbb{S}(n_h^1)$.

Hence, no such $\mathbb{S}$ exists.

$\square$

Here, we prove that there exists no algorithm that can generate a weak DAS schedule for both Δ_1 and Δ_2 with all nodes being assigned a single slot. Theorem 4 captures a lower bound for developing a weak DAS schedule for two sinks, in that it means that it is mandatory for some nodes to have at least two slots to solve weak DAS for two sinks.

6.3.3 Towards Minimizing $\sum_{\forall n \in V} f_n^{\mathbb{S}}$

Having established a condition that there should be a certain number of nodes that require at least two slots, an important question is: how are these nodes with 2 slots chosen from the network?

One way of building a network that solves the weak DAS problem is to assign 2 slots to the nodes on the path that connects Δ_1 and Δ_2 and assign 1 slot to all other nodes like shown in Figure 6.1. The values $s + i$ in the figure are the time slots of the nodes. The arrows in the figure shows the direction of packets send at time slot

[1]Since $\mathbb{S}(n)$ returns a set, we abuse the notation here for mathematical comparison.

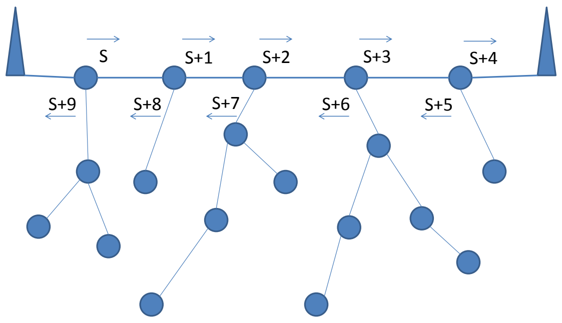


Figure 6.1: An example of network that solves weak DAS. Arrows show to which sink is used the slot. s is an integer that shows the slot of the node.

$s + i$. In the figure, all nodes send their values to one of the nodes on the path. In turn, the nodes on the path use 2 slots to send their aggregated values to two sinks, one slot for each sink. Thus, the minimum number of nodes with at least two slots that can connect two sinks is captured in the following result (Corollary 1):

Corollary 1 *Given a network $G = (V, A)$ with two sinks Δ_1 and Δ_2, then there exists a weak DAS $\mathbb{S}$ for G, $\sum_{\forall n \in V} f_n^{\mathbb{S}} = l - 1$, where l is the length of the shortest path between Δ_1 and Δ_2.*

Since we know that it is possible to obtain a weak DAS schedule $\mathbb{S}$ that assigns two or more slots to at most $l-1$ nodes, the objective is to determine the minimum number of such nodes with at least 2 slots. How many nodes on the shortest path can have only 1 slot? This is captured in the following result (Theorem 5):

Theorem 5 *Given a network $G = (V, A)$ with two sinks Δ_1 and Δ_2, a path $P = \Delta_1 \cdot n_1 \cdot n_2 \ldots n_{l-1} \cdot \Delta_2$ that is a shortest path between Δ_1 and Δ_2 of length l with $l-1$ nodes and a schedule that DAS-labels all nodes $n_1 \ldots n_{l-1}$ on P and P^r for Δ_2 and Δ_1 respectively, where P^r is the reverse of path P . Then, there exists no weak DAS schedule $\mathbb{S}$ such that $\sum_{\forall n \in V} f_n^{\mathbb{S}} \leq l - 3$.*

Proof.

Assumptions: We assume that there exists a weak DAS $\mathbb{S}$ such that $\sum_{\forall n \in V} f_n^{\mathbb{S}} \leq l-3$ under $\mathbb{S}$, and show a contradiction. We denote the set of slots assigned to any node $n_i \in \{n_1, \ldots, n_{l-1}\}$ by $(x_i^1, x_i^2, \ldots x_i^{n_i})$. Assume that there are two nodes $n_i, n_j \in \{n_1, \ldots, n_{l-1}\}$ that has one slot only and, since, under $\mathbb{S}$, they are assigned more than one slot, it means that $x_i^1 = x_i^2 = \ldots = x_i^{n_i}$ and $x_j^1 = x_j^2 = \ldots = x_j^{n_j}$.

Since $\mathbb{S}$ DAS-labels n_i on P for Δ_2, it means that $x_j > x_i$. However, since $\mathbb{S}$ DAS-labels n_j on P^r for Δ_1, it means that $x_i > x_j$, which is a contradiction. Hence, there exists no weak DAS $\mathbb{S}$ such that $\sum_{\forall n \in V} f_n^{\mathbb{S}} \leq l-3$

□

From the proof of Theorem 5, it can observed that the impossibility occurs when comparing two nodes with a single slot (i.e., nodes with 2 slots of the same value). However, if this asymmetry is broken, then it is possible to have a weak DAS schedule $\mathbb{S}$ such that $\sum_{\forall n \in V} f_n^{\mathbb{S}} = l-2$, which is the smallest number of nodes that need to have at least two slots to solve weak DAS for 2 sinks. There is one node, usually a neighbouring node of a sink along the shortest path that may be assigned one slot under such a weak DAS.

6.4 Algorithms

Based on the results developed in Section 6.3, we develop a 3-stage weak DAS algorithm. The first phase computes a shortest path between the two designated sinks. Every node on the shortest path is considered a *virtual sink*. The second phase consists of each virtual sink constructing a tree that satisfies some property, e.g., balanced tree. This phase is explained in section 6.4.2.1. The final phase consists in assigning slots to nodes in the network in such a way to satisfy a given property, e.g., minimum latency. This phase will be explained in 6.4.3.

In this work, we will focus on the following properties: (i) we develop a balanced

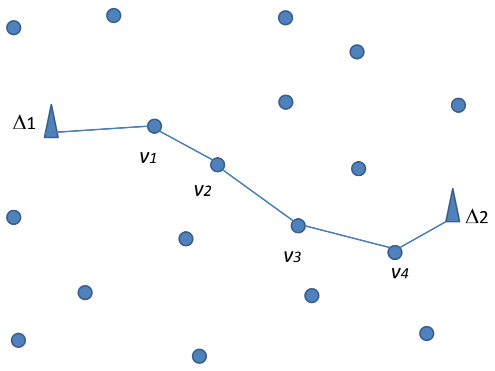


Figure 6.2: Shortest path between two sinks

tree algorithm such that nodes at a given level spend similar amount of energy and (ii) sibling nodes are allocated contiguous slots so that a parent node does not require switching its radio on and off to capture the data of its children, thus saving energy [7, 102, 56].

In the algorithms, we assume that there is no packet loss and no node failures. As mentioned in Chapter 7, we plan to work on addressing these problems in the future. We also assume that data packets are assumed to have the same size, and aggregation of two or more incoming packets at a node results in a single outgoing packet. Further, we do not make any assumptions about the network topology.

6.4.1 Phase 1: Computing the Shortest Path Between the two Sinks

As our results show that a shortest path between the two sinks Δ_1 and Δ_2 is required to minimise the number of nodes with more than a single slot, we first form a shortest path $P = \Delta_1 \cdot v_1 \ldots v_l \cdot \Delta_2$ between Δ_1 and Δ_2. We can obtain such a shortest path with a simple distributed shortest path algorithm using *Request* and *Reply* packets, and using hop number as a cost.

After forming P, all nodes on P will take the role of *virtual sink* and set their variables, called *vsink* to 1, and *hop* to 0. An example of P of length 5 with virtual sinks v_1, v_2, v_3 and v_4 is illustrated in Figure 6.2.

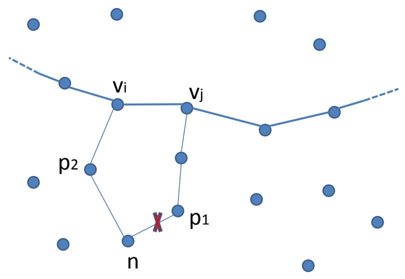
6.4.2 Phase 2: Developing a Tree Structure

Once a shortest path has been obtained from the first phase, there now exists a set of "virtual sinks" in the network, which we denote by VS. The virtual sinks are nodes that lie along the shortest path. In this phase each of these virtual sinks builds their own tree structure, which can be geared or optimised for a given property. In this chapter, we propose a balanced tree algorithm (Section 6.4.2.1), as such a structure enables a balancing of load for nodes at a given level. We also explain another DAS algorithm [120], which is cluster-based, against which we compare our results.

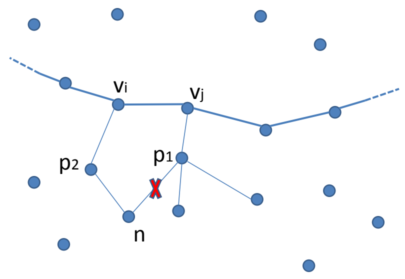
The first two phases of our algorithm is somewhat similar to the algorithm proposed in [52], where authors propose an algorithm that forms Greedy Incremental Tree(GIT)-like tree to perform energy efficient in-network data aggregation. Their algorithm consists of first forming a shortest path between the first source node to a sink and then connecting all other source nodes to the path in a greedy fashion using hop number as a cost. The idea behind building GIT-like tree is that it improves path sharing, i.e., it allows data aggregation earlier to reduce data transmissions. However, their algorithm differs from ours in that their algorithm does not balance the number of children while our algorithm does. The next section explains our balancing algorithm.

6.4.2.1 Balanced tree formation

In this section, we detail the balanced tree formation algorithm that we adopt (see Figures 6.4, 6.5). When developing the balanced tree, we focus on two main parameters, in the order described: (i) a node chooses a parent based on its (hop) distance



(a) Parent with smaller hop distance



(b) Parent with smaller number of children

Figure 6.3: Parent selection

from its virtual sink ancestor, in the sense that the node will choose to join a tree where it is closer to a virtual sink, and (ii) if there are competing trees, then a node will join the tree that will make the overall tree structure balanced among the nodes with the same hop distance.

A node can be in one of four states: ALONE, TEMP, JOINED and BALANCE. Initially, all virtual sinks are in the JOINED state, and all other nodes are in the ALONE state. A node goes to the TEMP state when it finds a potential parent with a smaller hop distance. A node goes to the BALANCE state when it finds a potential parent with a smaller number of children. In the TEMP or BALANCE state, a node waits for some time to get a response from the potential parent.

A node n will change its parent from p_1 to p_2 only if

- p_2 has a smaller hop distance (from a virtual sink) than that of p_1. Figure 6.3(a) illustrates this case.
- If both p_1 and p_2 are equidistant to a virtual sink, then n switches parent if p_2 has a smaller number of children. Figure 6.3(b) illustrates this case.

The first case makes the node have the shortest distance to a virtual sink, and the second case tries to balance the number of children of nodes with the same hop distances.

Informally, the algorithm starts with the virtual sinks broadcasting (i.e., advertising) *JOIN* packets periodically. When a node n_1 receives a *JOIN* packet from a node n_2, it compares its parent hop with the hop of n_2. If the hop of n_2 is smaller, then n_1 requests n_2 to be its parent by sending *REQ* packet, and sets n_2 as its parent if n_1 receives an *ACCEPT* packet from n_2 (states ALONE and TEMP of Figure 6.4, state JOINED lines 12-14 of Figure 6.5). If the hop numbers are equal and the number of children of n_2 is at least two smaller than the number of children of its current parent, then n_1 requests n_2 to be its parent by sending a *REQ_BAL* packet (state JOINED lines 15-17 of Figure 6.5). If n_1 receives a *BAL_ACCEPT* packet from n_2, it notifies its current parent, by sending a *DISCON* packet, stating that it will connect to another parent. It then sets n_2 as its parent. Whenever n_2 sends *ACCEPT* or *BAL_ACCEPT* to n_1, it adds n_1 to its *children* set. Whenever a node receives a *DISCON* packet from a node n, it removes n from its *children* set. When a node stops receiving any packet except *JOIN*, it goes to the *SCHEDULE* state.

Correctness of Balanced Tree Algorithm of Figures 6.4, 6.5

In BTF, as mentioned in the previous section, when selecting a parent, the highest priority is given to a node that has the shortest distance to a virtual sink. If there are more than one potential parent, a priority is given to a node with the smallest number of children in order to balance them. In this section, we show that BTF algorithm correctly achieves this goal.

Lemma 4 (Invariant of BTF) *Given a network $G = (V, A)$ with two sinks Δ_1, Δ_2, then the following is an invariant for the BTF algorithm in Figures 6.4, 6.5:*

$\forall m \in V \setminus VS :$

1.$(m.parent \neq \perp \Rightarrow m.hop \neq \infty)$

Process i

Variables of i
$state$ ∈{ALONE, TEMP, JOINED, BALANCE, SCHEDULE}; **Init:** ALONE
$hop, j \in \mathbb{N}$; **Init:** $j := 0$; $hop := \infty$
$parent$: 2-tuple $\langle id, numchild \rangle$; **Init:** $\perp$
$children : \{id : id \in \mathbb{N}\}$; **Init:** $\emptyset$
$JOIN, ACCEPT, REQ, REQ\_BAL, BAL\_ACCEPT, BAL\_DENY, DISCON$: Packet types

Constants of i
$threshJ$
% After forming a shortest path between Δ_1 and Δ_2, every node on the shortest path enters the JOINED state and starts to broadcast a *JOIN* packet.

```
state=ALONE
1 upon rcv⟨$JOIN, n, n\_hop, n\_numchild$⟩
2   if ($n\_hop+1 < hop$)
3     $state$:=TEMP
4     send($REQ, n$)

state=TEMP
1 upon rcv⟨$ACCEPT, n, i, n\_hop, n\_numchild$⟩
2   $parent.id$:=$n$
3   $parent.numchild$:=$n\_numchild$
4   $hop$:=$n\_hop + 1$
5   $state$:=JOINED

state=BALANCE
1 upon rcv⟨$BAL\_ACCEPT, n, i, n\_hop, n\_numchild$⟩
2    send(DISCON, $i, parent.id$)
3    $parent.id$:=$n$
4    $parent.numchild$:=$n\_numchild$
5    $hop$:=$n\_hop+1$
6    $state$:=JOINED

7  upon rcv⟨$BAL\_DENY, n, i$⟩
8    $state$:=JOINED
```

Figure 6.4: Balanced tree formation algorithm-partA

2. $\wedge\ (m.hop \leq m.hop')$

3.$\wedge[(m.parent' \neq m.parent) \Rightarrow ((m.hop < m.hop') \vee ((m.hop = m.hop') \wedge (m.parent.numchild <$

```
**state=JOINED**
1  **while**($j < threshJ$)
2    **bCast**($JOIN, i, hop, |children|$)
3    $j := j + 1$
4    **if**($j = threshJ$)
5      $state$:=SCHEDULE   % See Fig. 6.8, 6.9
6    **endif**

7  **upon rcv**$\langle JOIN, n, n\_hop, n\_numchild \rangle$
8    **if**($parent.id = n$)
9      $parent.numchild$:=$n\_numchild$
10     $hop := n.hop + 1$
11   **endif**
12   **if** ($n\_hop$+1 $< hop$)
13     $state$:=TEMP
14     **send**($REQ, i, n$)
15   **elseif**($n\_hop$+1=$hop \wedge parent.numchild$-$n\_numchild \geq$2)
16     $state$:=BALANCE
17     **send**($REQ\_BAL, i, n, parent$)
18   **endif**

19 **upon rcv**$\langle REQ, n, i \rangle$
20   **send**($ACCEPT, i, n, hop, |children|$)
21   $children$:=$children$ $\cup\{n\}$
22   $j := 0$

23 **upon rcv**$\langle REQ\_BAL, n, i, n\_parent \rangle$
24   **if**($n\_parent.numchild - |children| \geq$2)
25     $children$:=$children$ $\cup\{n\}$
26     **send**($BAL\_ACCEPT, i, n, hop, |children|$)
27   **else**
28     **send**($BAL\_DENY, i, n$)
29   **endif**
30   $j := 0$

31 **upon rcv**$\langle DISCON, n, i \rangle$
32   $children$:=$children \backslash \{n\}$
33   $j := 0$
```

Figure 6.5: Balanced tree formation algorithm-partB

$m.parent.numchild'$)))]

Proof. To prove that the above is an invariant, we show that it is satisfied in the initial state of the program and subsequent action preserves the invariant.

The invariant is trivially satisfied in the initial state where $parent = \perp$ and $hop = \infty$. In state ALONE, the invariant is not violated as no state change occurs. In state TEMP, the value of hop decreases since a node m receives the $\langle ACCEPT \rangle$ from a node n only if n has a smaller hop (line 2 state ALONE, and line 12 state JOINED), which preserves the invariant. In state TEMP, the value of the parent changes too as either m gets a new parent with a smaller hop (line 12 state JOINED) or gets a parent for the first time (line 2 state ALONE), which preserves the invariant. In state BALANCE, when node m receives a $\langle BAL\_ACCEPT \rangle$ message, m changes parent due to m having a possible new parent with fewer children (state JOINED line 15). In state JOINED, when node m receives a $\langle JOIN \rangle$ packet, it updates its state based on that of its parent's, which preserves the invariant. □

The BTF algorithm, depending on the network topology, may not always balance the number of children of nodes at the same hop distance. However, the BTF algorithm guarantees that if a node n, with a parent p, has several potential parents p_i, $i > 1$, then the number of children of p cannot be more than the number of children of $p_i + 1$. In [84], it has been shown that, in a network where nodes are evenly distributed, the number of children per node is slightly bigger than one and tends to 1 as the hop number increases. And the average number of children per node in a $2D$ network is $\overline{N_{child}} = \frac{2h+1}{2h-1}$. The output of our balancing algorithm shows the same results: all nodes except the virtual sinks have 1 child in average.

6.4.2.2 Cluster-Based DAS Scheduling [120]

To maximise the benefit from the spatial advantage when allocating slots, the authors of [120] build an aggregation tree based on the concept of Connected Dominating Set (CDS). The DAS algorithm adopts the CDS construction algorithm proposed in [114] which, in turn, is based on a Maximal Independent Set, with a little modification. Instead of using the original root of the dominating set, they use the sink as the root of the dominating set. For proof of correctness, or otherwise, of the algorithm, we refer the reader to [120].

6.4.3 Phase 3: Slot Allocation for DAS Scheduling

Once the virtual sinks are obtained (phase 1) and each one has its own tree structure (phase 2), every node will identify its *children*, *parent* and *hop*. Also, a node will determine whether it is a virtual sink through the variable $vsink$. If $vsink = 1$, then the node is a virtual sink.

6.4.3.1 Enery-Efficient Collision Free (EECF) DAS Algorithm for Balanced Tree

In this section, we propose a DAS algorithm that leverages the balanced tree obtained (see Section 6.4.2.1). To make the DAS energy-efficient, we seek to assign contiguous slots to children so that a parent does not need to continuously sleep and wake-up to collect data as this leads to unnecessary energy usage [7, 102, 56]. Further, since the tree is balanced (at a given level), then nodes at that level spend comparable amount of energy.

We propose a weak DAS algorithm that works in a greedy fashion (see Figures 6.8, 6.9). In the algorithm, every node maintains variables $maxslot$ and $minslot$. These variables are used to tell neighboring nodes that its children are assigned to the slots starting from $maxslot$ down to $minslot + 1$. This allows every node's children to have contiguous time slots. The algorithm uses a special packet called $SLOT$ which includes 9 variables necessary for scheduling.

Informally, the scheduling algorithm starts by assigning a time slot to the virtual sink v_1 that is a neighbor of, without loss of generality, Δ_1. As we have proved, there should exist at least $l-2$ nodes with at least two slots. Thus, all nodes except v_1 will be assigned two different slots. The time slot that will be assigned to v_1 is $|V|$. If another virtual sink $v_i, i \neq 1$ receives a $SLOT$ packet from v_{i-1}, it sets its first slot to 1 less than the *first* slot of v_i and second slot to 1 more than the *second* slot of v_i, and then broadcasts a $SLOT$ packet. The *maxslots* of virtual sinks are assigned to 2 less than their *first* slots (it is 2 less because the next smaller slot is reserved for the next virtual sink), and the *minslots* to the differences of *maxslots* and their number of children (lines 2-7 of Figure 6.8). If a node n_1 receives a $SLOT$ packet from its *parent*, n_1 sets its slot to the difference of sender's *maxslot* and *rank* of n_1, and then broadcasts a $SLOT$ packet (lines 8-14 of Figure 6.8). We assume that nodes know their *ranks* before we run the scheduling algorithm. A node can learn its rank in two ways: 1) the parent node may compute and send the rank to its children or 2) the parent node broadcasts IDs of its children and then children computes their ranks themselves.

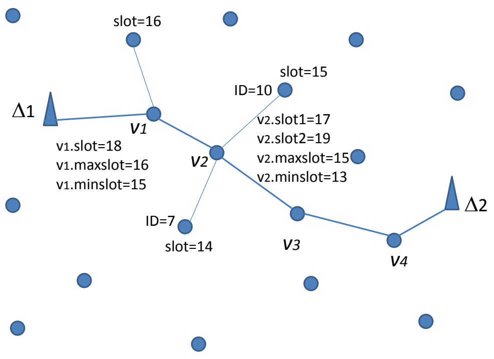


Figure 6.6: Scheduling example

Example. For the sake of clarity, consider the example in Figure 6.6, where the number of nodes $|V|$=18 and the number virtual sinks $|VS|$=4. According to the algorithm, the only slot of v_1 will be 18, i.e., v_1.slot=18. The first and second slots

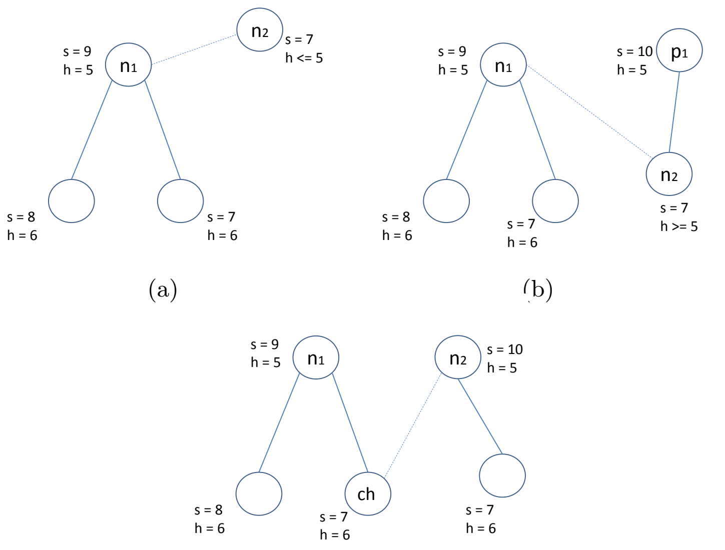


Figure 6.7: Scheduling cases. s: slot number, h: hop number. A line between nodes show the existence of communication link between them

of v_2, v_3 and v_4 will be $\{17, 19\}$, $\{16, 20\}$ and $\{15, 21\}$ respectively, e.g., v_2.slot1=17 and v_2.slot2=19. The *maxslot* of v_1 will be 2 less than its slot, that is 16. And, as v_1 has only one child, the *minslot* will be $16 - 1 = 15$. While the *maxslot* of v_2 will be $17 - 2 = 15$ and *minslot* will be $15 - 2 = 13$. The children of v_2 with IDs 10 and 7 will take their slots from the range (*maxslot*, *minslot*+1), i.e., (15, 14), according to their ranks. In this case, the slot of node with ID=10 will take slot 15, and the node with ID=7 will take slot 14.

If a node detects a slot conflict, depending on the *priority*, which is basically the hop distance, slot and rank in that order, the node decides whether to change the slots of its children. And this makes the scheduling collision-free. A node n_1 tells its children to change their slots if

- n_1 finds a neighboring node n_2 that share the same slot with its child and n_2 has a smaller or equal hop distance to n_1's hop distance (lines 20-30 of Figure 6.8, 6.9). Figure 6.7(a) illustrates this case.

- n_1 finds a neighboring node n_2 that share the same slot with its child and n_2's parent slot is bigger than n_1's slot (lines 37-43 of Figure 6.9). Figure 6.7(b) illustrates this case.
- one of n_1's child *ch* tells to change because *ch* share the same slot with a child of n_2 that has bigger slot than n_1's slot. The variable *otherslot* is used for this purpose (lines 16-19, 31-35 of Figure 6.8, 6.9). Figure 6.7(c) illustrates this case.

After all the nodes are assigned their slots, all nodes' slot values are sent to Δ_1 where it computes the minimum of the slots, and broadcasts that value to the nodes. The nodes after receiving the minimum value can compute their slots by taking the difference of its slot and the minimum value, and adding 1.

Correctness of EECF

Lemma 5 (Invariant of EECF) *Given a network $G = (V, A)$, with a set of virtual sinks $VS \subset V$ that link 2 sinks Δ_1, Δ_2, then the following is an invariant of EECF-DAS:*

$\forall m \in V ::$

$I0\ m.slot \neq \infty \wedge m.slot2 \neq \infty \Rightarrow m.vsink$

$I1 : \wedge\ m.slot \neq \infty \wedge m.slot2 = \infty \Rightarrow \neg m.vsink$

$I2 : \ \wedge\ \ m.slot < m.slot' \Rightarrow m.slot < m.parent.slot$

$I3 : \ \wedge (m.slot \neq m.slot') \Rightarrow (\exists n \in 2HopN(m) \cdot m.slot' = n.slot') \vee (\exists y \cdot y.parent = m.parent : y.slot' = n.slot')$

Proof. $I0, I1$ follow trivially from the program (statements 2-7). $I2$ is satisfied by statements on lines $8-13$ (which ensures that $m.slot < m.parent.slot$) and $20-40$, in case of slot collisions. $I3$ is handled through any case from statements $20-40$ to resolve slot collisions. □

Process i

Variables of i
$slot, slot2, maxslot, minslot, otherslot, parentslot, s, x \in \mathbb{N}$;
Init:$slot, slot2, maxslot, minslot, otherslot, parentslot$:=$\infty$; $s, x := 0$
$vsink \in \{0, 1\}$ %vsink=1 if i is a virtual sink
$rank(id)$: a function that returns the number of greater or equal values to id in $children$ of id's parent.
P : 9-tuple $\langle id, hop, slot, slot2, maxslot, minslot, otherslot, parentslot, vsink\rangle$
$SLOT$: Packet type
Constants of i
$threshS$

% When the neigbouring virtual sink of Δ_1 enters the $SCHEDULE$ state, it sets its $slot, slot2 := |V|$ and starts **broadcast**$(SLOT, threshS)$

```
Import from BTF % Import the variables of BTF (Fig 6.4, 6.5)
state=SCHEDULE
1 upon rcv⟨$SLOT, \alpha : P$⟩
% If the src and $i$ are virtuals sinks and $i$ has not assigned a slot yet,
% set states accordingly
2  if($slot = \perp \wedge vsink = 1 \wedge \alpha.vsink = 1$)
3    $slot := \alpha.slot - 1$
4    $slot2 := \alpha.slot2 + 1$
5    $maxslot := slot - 2$
6    $minslot := maxslot - |children|$
7    broadcast($SLOT, threshS$)

% If the src is the parent of $i$ and src's maxslot has been changed,
% set states accordingly
8  elseif($parent.id = \alpha.id$)
9     $parentslot := \alpha.slot$
10    if($slot \neq \alpha.maxslot - rank(i)$)
11      $slot := \alpha.maxslot - rank(i)$
12      $maxslot := slot - 1$
13      $minslot := maxslot - |children|$
14      broadcast($SLOT, threshS$)
% If the src is a potential parent
15  elseif($\alpha.hop = hop - 1$)

% If $i$ has the same slot as a child of src and src's slot is larger than
% $i$'s parent slot, or they are equal and src's id is larger than $i$'s parent id,
% then notify $i$'s parent about this (See lines 31-35 of Fig 6.9)
16    if(($\alpha.slot > parentslot \vee (parentslot = \alpha.slot \wedge \alpha.id \geq parent.id)$)
17                 $\wedge(\alpha.maxslot \geq slot \wedge slot > \alpha.minslot)$)
18      $otherslot := \alpha.minslot$
19      broadcast($SLOT, threshS$)

% If one of $i$'s children shares the same slot with the potential parent,
% set states of $i$ accordingly
20    elseif($maxslot \geq \alpha.slot \wedge \alpha.slot > minslot$)
21      $maxslot := \alpha.slot - 1$
22      $minslot := maxslot - |children|$
23      broadcast($SLOT, threshS$)
24    endif
```

Figure 6.8: Data aggregation scheduling algorithm-partA

```
% If $i$'s and src's hops are equal and one of $i$'s child shares the same slot
% with the src, set states of $i$ accordingly
25 elseif($\alpha.hop = hop$)
26   if($maxslot \geq \alpha.slot \wedge \alpha.slot > minslot$)
27     $maxslot := \alpha.slot - 1$
28     $minslot := maxslot - |children|$
29     broadcast($SLOT, threshS$)
30   endif

% If the src is a child of $i$ and has detected a collision, then set states
% of $i$ accordingly (See lines 16-19 of Fig. 6.8)
31 elseif($\alpha.id \in children$)
32   if($\alpha.otherslot < maxslot$)
33     $maxslot := \alpha.otherslot$
34     $minslot := maxslot - |children|$
35     broadcast($SLOT, threshS$)
36   endif

% If the hops of src and $i$'s children are equal and a child of $i$ shares the
% same slot with the src and $i$'s slot is smaller than src's parent slot or,
% if the slots are equal, $i$'s id is smaller than src's parent id,
% then set states of $i$ accordingly
37 elseif($\alpha.hop = hop + 1$)
38   if(($\alpha.parentslot > slot \vee (\alpha.parentslot = slot \wedge \alpha.id > i)) \wedge$
39                 $(\alpha.slot \leq maxslot \wedge \alpha.slot > minslot)$)
40     $maxslot := \alpha.slot - 1$
41     $minslot := maxslot - |children|$
42     broadcast($SLOT, threshS$)
43   endif
44 endif

broadcast($SLOT, x$)
1 $s := 0$
2 while($s < x$)
3   bCast($SLOT, \alpha : P$)
4   $s := s + 1$
```

Figure 6.9: Data aggregation scheduling algorithm-partB

Theorem 6 *Given a network $G = (V, A)$ with 2 sinks Δ_1, Δ_2, then (BTF ; EECF-DAS) solves the EECF-DAS problem with a schedule $\mathbb{S}$ s.t. $\sum_{n \in V} f_n^{\mathbb{S}} = l - 2$, where l is the length of the shortest path between Δ_1 and Δ_2.*

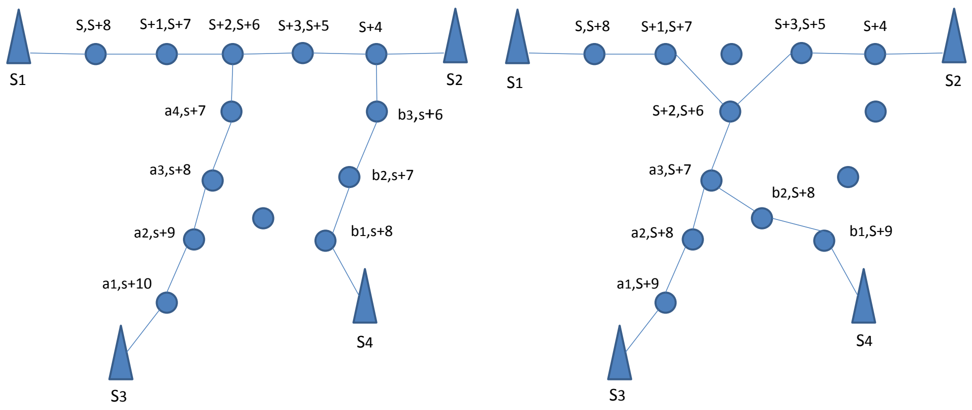
Proof. Follows directly from Lemmas 4 and 5. $\square$

Our data aggregation convergecast scheduling algorithm solves the collision problem, but it does not address the node failure problem. However, when a node has potential parents, it is easy to tolerate if the node's parent fails. Since our scheduling algorithm does not allow a child n, to have a time slot that is later than any potential parent's time slot, n can be connected to any of its potential parent, without changing its slot. But, this might disrupt the contiguousness property of the algorithm. Moreover, this does not work when the parent node (failed node) is one of the nodes on the shortest path. As we will mention in Chapter 7, we are planning to work on the node failure problem in the future.

6.4.3.2 Slot Assignment for Cluster-Based DAS

In this section, we present the slot assignment algorithm that is based on the clusters formed [120] (see Section 6.4.2.2).

After forming a data aggregation tree, every node computes its *competitor set*, a set of nodes that collide with the node. The general idea of this scheduling algorithm is as follows: Initially, all leaf nodes are in the READY state and non-leaf nodes are in the NOT-READY state. A node in READY state assigns a slot only if all nodes that are in READY state and with larger IDs in its competitor set have already assigned their slot. The node takes the smallest available slot in its competitor set. If all children of a node have completed assigning their slot, the node goes to the READY state. A node after assigning its slot broadcasts to notify the nodes in its competitor set about the change. When the sink receives notifications from all of its children, the scheduling algorithm stops.



(a) Sinks are connected to the shortest path between s_1 and s_2

(b) Sinks are connected with each other with the smallest number of nodes

Figure 6.10: Example: (a) Backbone formed by our algorithm and (b) Backbone with smaller number of nodes. Letters and numbers are slots. Slots labeled with a_i and b_i are used to send data toward the path between s_1 and s_2.

6.4.4 Data Aggregation Convergecast Scheduling in WSN with more than 2 sinks

Now, the question is can our algorithm be applied to networks with more than 2 sinks? The answer is yes, with a bit of modification. However, this will not give an optimal solution to the problem given in Section 6.2.1. Consider the network with 4 sinks given in Figure 6.10. The nodes labeled with numbers and letters are the nodes with two slots, where numbers and letters are slot numbers. Assume that our algorithm connects s_1 and s_2 with a shortest path p as shown in Figure 6.10(a). As our algorithm connects all other nodes with shortest paths to p, sinks s_3 and s_4 are also connected to p with shortest paths, say p_2 and p_3. Now, if we assign two slots to the nodes on paths p, p_2 and p_3, as shown in Figure 6.10(a), and a single slot to each of the other nodes, it is clear that data aggregation convergecast scheduling could be done. In this case, the number of nodes with more than 1 slot will be 11. However, as can be seen from Figure 6.10(b), there exists a solution where the number of nodes with more than 1 slot is 9. Note that, the number of nodes that have more than 1 slot is equal to one less than the number of nodes in the formed tree (backbone) that connects four sinks. Therefore, as mentioned earlier, the solution for the problem is relevant to the minimal Steiner tree problem, which has been shown to be NP-complete [58].

6.5 Experimental Setup

In this section, we present the simulation and testbed setup used to evaluate the working and performance of EECF-DAS.

6.5.1 Simulation Setup

We perform TOSSIM [78] simulations to evaluate our EECF-DAS algorithm. We evaluated it on networks of sizes 400, 600, 800 and 1000 nodes. We constructed the networks such that a node has a communication radius of 10m, 15m and 20m, for two nodes in the communication range were given a link gain of -65 dBm [2]. Each node is given a noise model from the "casino-lab" noise trace file, which is taken in the Casino Lab of Colorado School of Mines. The nodes were uniformly randomly distributed on a 100m$\times$100m surface. We varied the distance between two sinks: i) two sinks were deployed at two diagonally opposite corners, ii) two sinks were deployed such that the distance between them is 4 hops and iii) the distance between them is 6 hops.

To compare the performance of EECF-DAS with the cluster-based DAS (CDAS) protocol proposed in [120], we also simulated EECF-DAS and CDAS (see Section 6.4.2.2) in Java. We used Java because in [120], the authors have not clearly stated how a node communicates with a node in its competitor set, and they simulated CDAS using C++. We chose CDAS because it is claimed to be one of the algorithms with the lowest latency. However, as CDAS was intended for a network with only one sink, we adapted it to make it work in a network with two sinks.

For comparison purposes, we adapted CDAS into two different ways: i) we ran DAS twice, one for each sink, and called this adapted algorithm 2DAS, and ii) we first run a shortest path algorithm to form a shortest path (as in EECF) between the two sinks and, instead of assigning Δ_1 as the root of the dominating tree, as in [114], we assigned each virtual sink (a node on the shortest path) as a root of a dominating tree and called this adapted algorithm SP-DAS. SP-DAS shows that our proposed approach is modular. We ran each case ten times and computed the average.

6.5.2 Testbed Setup

We have run the EECF-DAS algorithm on Indriya testbed (see Section 4.5.2.1 for characteristics of Indriya). We select the node with $ID = 1$ as sink 1 and $ID = 46$ as sink 2. To increase the diameter (largest hop number) of the network, we set the transmission power to 7. The number of hops between sink 1 and sink 2 is 5.

The constant values used in the algorithm are given in Table 6.1. The values are used to send corresponding packets more than once as there could be packet losses, though it does not solve the packet losses problem completely. The values could affect the total number of transmitted packets to complete the scheduling.

Constant name	**Value**
threshJ	15
threshS	10

Table 6.1: Parameter values used in experiments.

In simulations and testbed experiments, we compare the latency, which is equal to the largest slot of the nodes in the network, and the number of time slots at which each node should be awake to transmit and receive packets.

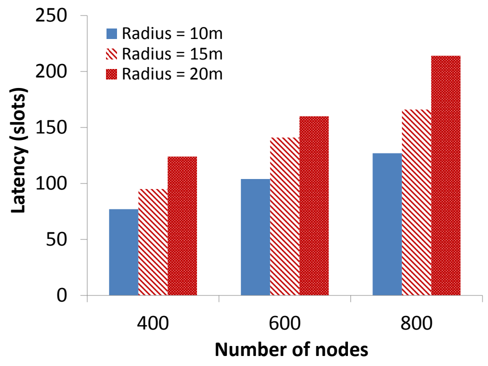
6.6 Simulation and Testbed Results

In this section, we present the results obtained from the simulation and testbed experiments.

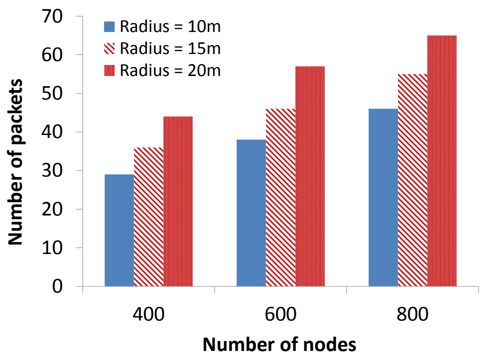
6.6.1 Simulation Results

The latency and message overhead of EECF

Figure 6.11 shows the latency and the number of packets transmitted to complete the scheduling when simulated with TOSSIM. In Figure 6.11(a), the average latency is shown. We can see that the latency is low relative to the number of nodes in the network, although EECF considers contiguousness of slots. Also we see that the latency is directly related to the neighborhood size/network density. Figure 6.11(b) shows the average number of packet transmissions per node to complete the scheduling. From the figure we see that it increases linearly as the neighborhood size/network density increases. This shows that our algorithm is linear and can work with networks of bigger size.

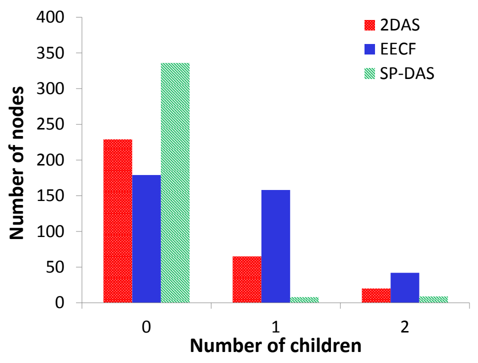


(a) The average latency.

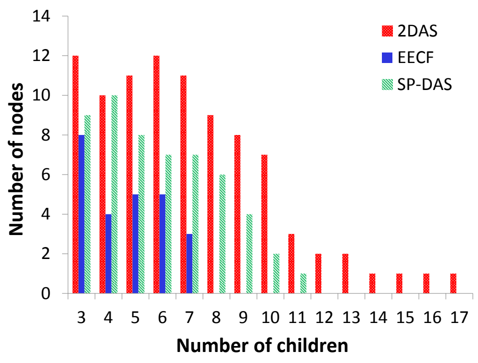


(b) The number of packet transmissions per node.

Figure 6.11: The average latency and number of packet transmissions with different network sizes and transmission ranges (EECF)



(a) The number of nodes with 0,1 and 2 children.



(b) The number of nodes with ≥3 children.

Figure 6.12: Comparison of number of children of EECF, SP-DAS and 2DAS (400 nodes, transmission range=10m)

Comparing EECF, SP-DAS and 2DAS

Figures 6.12 and 6.13 show the latency and the number of children obtained from the three algorithms run on a network where two sinks were placed at opposite corners of the network.

Number of children (sinks are at opposite corners of the network) Figure 6.12 shows the number of children of networks constructed by EECF, SP-DAS and 2DAS. As can be observed, 2DAS has more nodes that have large number of children, which implies that nodes in 2DAS should be awake more time slots than EECF and SP-DAS. The figure also shows that our balancing algorithm decreases the number of nodes that have large number of children.

Latency (sinks are at opposite sides of the network) Figure 6.13 shows the number of slots required for data aggregation convergecast in one round in networks of different sizes and transmission ranges. From the figure we can see that, in some cases, EECF and SP-DAS aggregate 2 times faster than 2DAS.

From the figures 6.12 and 6.13 we can conclude that the algorithms that use the shortest path as a backbone (EECF and SP-DAS) show better results in terms of latency and balance in number of children.

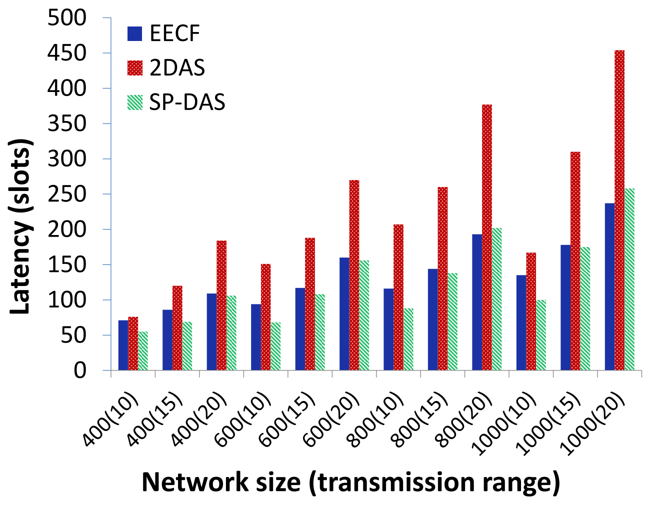
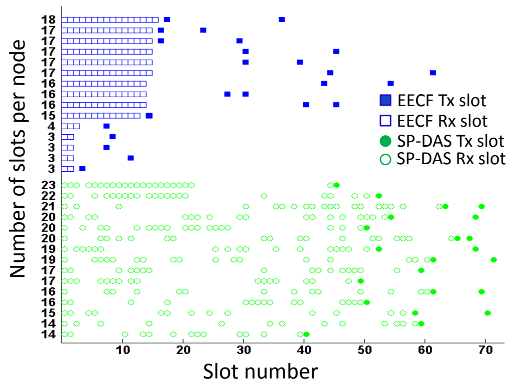


Figure 6.13: Comparison of latencies (in slots) of EECF, SP-DAS and 2DAS with different network sizes and transmission ranges.

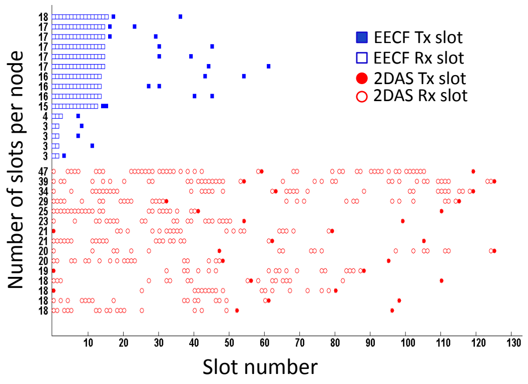
Slot distribution (sinks are at opposite sides of the network) Figure 6.14 shows the transmission and reception time slots of the 15 nodes of EECF, SP-DAS and 2DAS that have maximum number of slots in a network of 400 nodes with a transmission range equal to 15m. The filled slots indicate the transmission slots and empty ones indicate reception slots. From the figure we can see that the nodes scheduled by EECF have contiguous reception slots, while reception slots of SP-DAS and 2DAS are usually separated. It can also be observed that the nodes scheduled with EECF needs to switch from the *sleep* to the *active* mode at most 3 times. While in 2DAS and SP-DAS the number of switches could be large and can alternate every slot. From the figure we see that in 2DAS and SP-DAS the maximum number of switches is 20 and 13 respectively.

From the figure we can infer that by balancing trees and assigning contiguous slots to children we can increase the sleeping time of nodes and reduce the number of sleep-active transitions, thereby reducing the energy consumption of nodes.

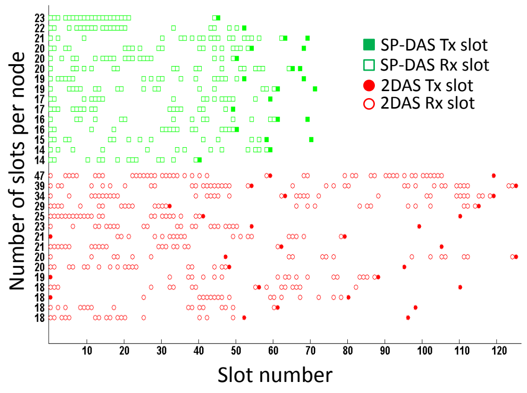
It can be observed from Figure 6.14(a) that, as both of the algorithms use a



(a) EECF and SP-DAS



(b) EECF and 2DAS

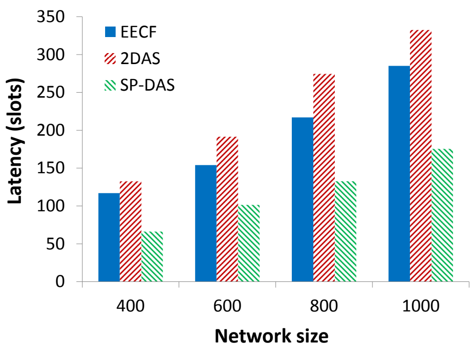


(c) SP-DAS and 2DAS

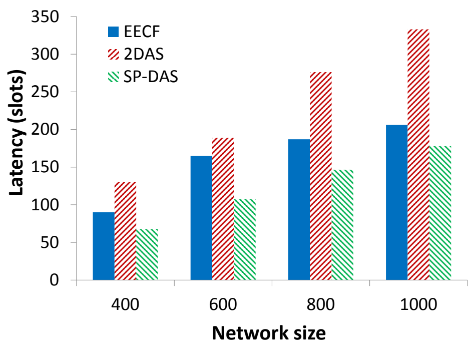
Figure 6.14: Distribution of slots of 15 nodes that have maximum number of slots of EECF, SP-DAS and 2DAS (400 nodes, transmission range = 15m)

shortest path in their aggregation tree, the number of nodes with 2 transmission slots are equal. However, in the figure, there are 9 nodes in EECF and only 7 nodes in SP-DAS with 2 transmission slots. It is because in SP-DAS there could be nodes that have more children than that of the nodes on the shortest path.

Latency (sinks are 4 and 6 hops away) Figures 6.15(a) and 6.15(b) show the latency obtained from EECF, SP-DAS and 2DAS run on networks where two sinks were placed such that the distance between them is 4 and 6 hops. From these two figures and from Figure 6.13 we can notice that as the distance between two sinks decrease, the latency obtained from EECF increases. However, the latencies obtained from SP-DAS and 2DAS remain almost same. This is because 2DAS and SP-DAS use CDS (Section 6.4.2.2) to build the structure, which depends on the locations of nodes. Therefore, SP-DAS is more efficient in terms of latency than EECF if the distance between sinks is small.



(a) Distance between two sinks is 4 hops.



(b) Distance between two sinks is 6 hops.

Figure 6.15: Comparison of EECF, SP-DAS and 2DAS (transmission range=15m)

Slot distribution (sinks are 4 and 6 hops away) Figure 6.16 shows the transmission and reception time slots of the 15 nodes of EECF, SP-DAS and 2DAS that have maximum number of slots in a network of 400 nodes with a transmission range equal to 15m where the distance between two sinks is 4 hops and 6 hops. From the figure we see that slots in EECF are contiguous and the number of nodes that have large number of receive slots (children) is around the number of hops between two sinks. This corroborates the fact that EECF reduces energy cost by reducing the number of sleep-active transitions and the number of nodes with large number of slots independent of the distance between two sinks.

6.6.2 Testbed Results

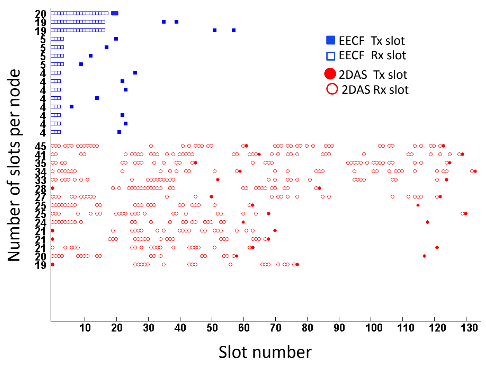
We have run EECF-DAS on Indriya about 100 times to check (i) the number of children per node and (ii) the latency (number of slots needed per round).

Number of children Figure 6.17 shows the the number of children obtained from EECF algorithm. As can be observed, most nodes (about 90%) have 0,1 or 2 children. And the remaining nodes, including virtual sinks, have between 3 and 7 children. The values show that by balancing the number of children we can decrease the number of nodes that have large number of children, which in turn means that most of the nodes wake up for only short period of time. The obtained result supports the results obtained from simulation experiments.

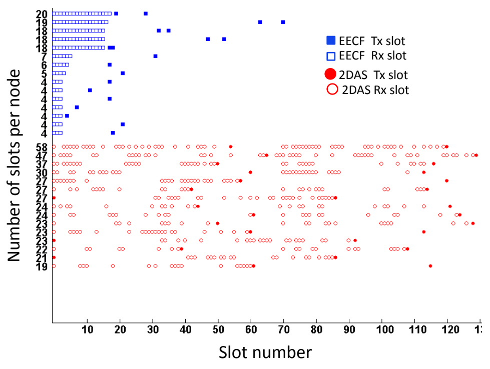
Latency and Slot distribution Figure 6.18 shows the transmission and reception time slots of the 15 nodes of EECF. The filled slots indicate the transmission slots and empty slots indicate reception slots. As in simulation results, EECF assigned contiguous slots to nodes, which reduces the number of sleep-active transitions, thereby reduces the energy cost.

Table 6.2 shows the number of slots required for data aggregation convergecast in one round. For 100 nodes of Indriya testbed, EECF assigns about 36 time slots in average.

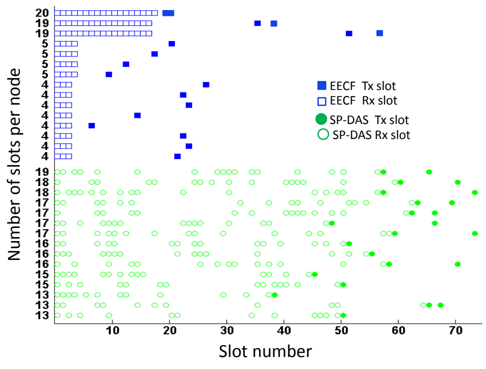
One point to note from the figures is that, there are a few nodes that are not on the shortest path and whose number of children is about the same as of the nodes on the shortest path. This is because the Indriya testbed has 3D topology and the nodes are not perfectly uniformly located. Thus, as discussed in Section 6.4.2.1,



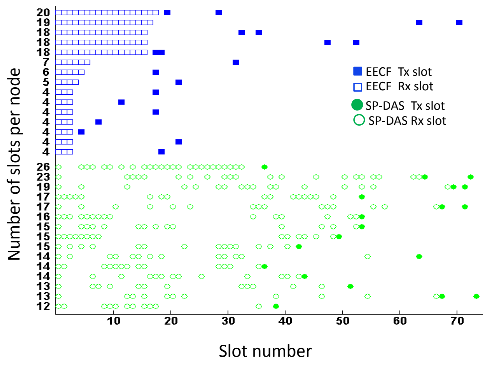
(a) Distance between two sinks is 4 hops.



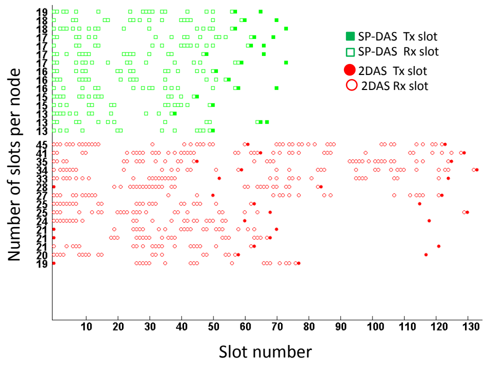
(b) Distance between two sinks is 6 hops.



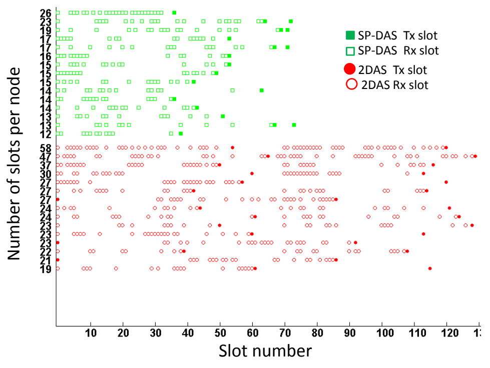
(c) Distance between two sinks is 4 hops.



(d) Distance between two sinks is 6 hops.



(e) Distance between two sinks is 4 hops.



(f) Distance between two sinks is 6 hops.

Figure 6.16: Distribution of slots of 15 nodes that have maximum number of slots of EECF, SP-DAS and 2DAS (400 nodes, transmission range = 15m)

BTF may not guarantee the balance of children depending on the network topology.

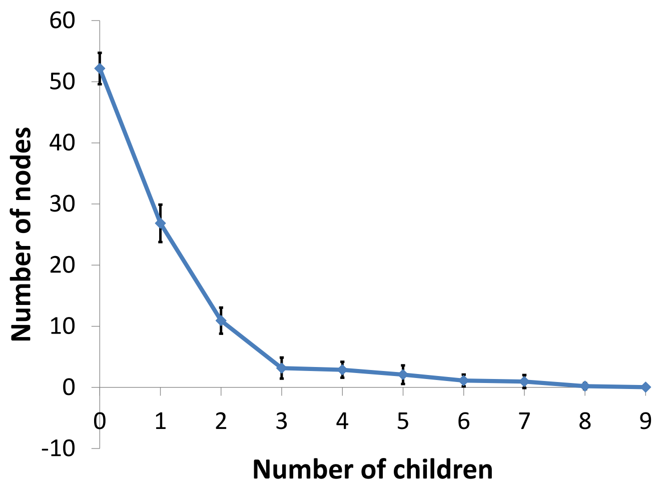


Figure 6.17: Number of children per node

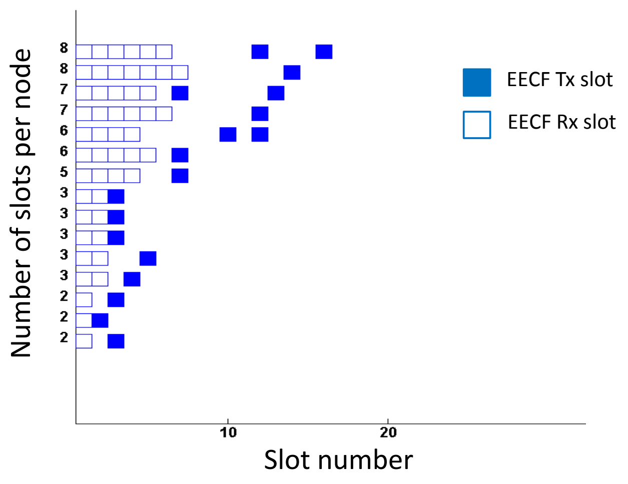


Figure 6.18: Testbed results: Distribution of slots of 15 nodes that have maximum number of slots of EECF

	Latency (Slots)
AVG	36.6
STD	6.0

Table 6.2: Average latency: the average number of slots used to convergecast in one round.

6.7 Related Work

In this section, as the algorithms related to data aggregation convergecast scheduling were presented in Section 2.2, we survey balancing tree algorithms only.

In [48], authors propose a distributed algorithm, similar to ours, with a goal to maximize the network lifetime. The idea is to balance the number of children according to a function of three variables: number of children, number of neighbors and the hop distance in that order. So, a node chooses a parent first with fewer children, then fewer neighbors, and finally, with the shortest distance. Different from our algorithm in that our algorithm first chooses a node with the smallest hop distance as a parent. Hence, in this algorithm nodes could choose a parent with a longer distance.

In [25], authors propose an Adjustable Convergecast Tree (ACT) algorithm that builds load-balancing tree which is based on shortest path tree. ACT is designed for convergecast without data aggregation. Therefore, in the algorithm, when balancing, each node should know the number of all descendant nodes. As we assume a data aggregation technique while convergecasting, this algorithm is not suitable for our algorithm. Further, in [101], authors show that ACT will not balance trees in some cases and propose an Efficient Balancing Tree algorithm (ECT). Their algorithm is based on the theorem given in [84] that states that the average number of children of node tends to 1 as the hop number of node increases. However, in their algorithm, they only balance nodes that have more than 1 child and have potential parents with 0 children. This algorithm does not work correctly when potential parents have more than 0 children. For example, if a node's parent has 5 children and its potential parent has 2 children, the node does not connect to the potential parent to make the number of children 4 and 3, respectively. Hence, the algorithm does not balance in this case.

In [44], authors propose Load-Balanced Data Aggregation Tree(LBDAT), an algorithm that is based on Maximal Independent Set (MIS) and Connected MIS under probabilistic network model which assumes lossy links. However, their algorithm is centralized.

6.8 Conclusion

In this chapter, we have addressed the problem of data aggregation scheduling (DAS) in WSNs with two sinks. There are a few works that address DAS problem in WSNs with multiple sinks. However, all of them consider many-to-one communication. To the best of our knowledge, this is the first work that deal with the DAS problem in WSNs with more than one sink that use many-to-many communication.

Before presenting our algorithm, we have formalized the DAS problem. Then, we have showed that it is impossible to have a schedule in which all nodes have only one slot. Consequently, there should be some nodes in the network that have more than one slot. We have also proved that the nodes that have multiple slots should form a path that connects the sinks. As a result, to have a schedule that have minimum number of nodes with multiple slots, only the nodes on the shortest path between two sinks should have multiple slots.

Based on the theoretical results as mentioned above, we have proposed a data aggregation scheduling algorithm. Our algorithm is based on first forming a shortest path between two sinks, building balanced tree rooted at each node on the shortest path and then allocating time slots to nodes such that every node's data is aggregated towards two sinks. Further, the scheduling algorithm assigns slots to nodes such that every node's children have contiguous slots. This method makes the convergecast more energy efficient and the energy consumption load more balanced as the energy consumption is directly related to the number of message transmissions and receptions, and to the active time of a node.

We have performed simulation and real-world testbed experiments to evaluate the performance of our algorithm. The experimental results show that our balanced tree formation algorithm and scheduling algorithm work correctly and make a schedule with low latency compared to an algorithm that have been developed for a WSN with a single sink. Moreover, our algorithm assigns contiguous slots to children so as to reduce the number of sleep-active transitions. Reducing the number of sleep-awake transitions reduces energy cost.

The results also show that our approach is modular in the sense that different structures could be formed to have different properties. For example, rather than balancing the number of children among parents and building balanced trees rooted at each node on the shortest path, we could build minimum spanning trees, linear trees or other type of structure to have other properties. For example, if we build linear trees, then delay could be high, however, all nodes will receive and transmit at most once which conserves energy.

CHAPTER 7

Conclusion and Future Work

Improvements in WSN technology have made WSN applications possible in different fields. However, due to stringent constraints of sensor nodes, developing protocols for WSN applications may not be easy. Sensor nodes are limited in memory, computation power and energy, and prone to failures. Therefore, protocols designed for WSNs should consider these characteristics of sensor nodes. Furthermore, the characteristics of wireless links and the environment where the network is deployed should be taken into account.

Many protocols have been developed to address the challenges brought by WSNs. However, there still exist problems that should be addressed. For example, there exist several data dissemination protocols, however, these protocols do not consider transient faults that can corrupt values stored in the memory and packets. If the protocols heavily depend on such values, then the corruption of these values may negatively impact on the protocols. Another area that we should work on is WSNs that contain more than one sink. A sink may stop communicating with nodes for some time due to reasons such as link failures and energy deficiency. In a WSN with a single sink, the outage of the sink results in the loss of the network. Hence, for WSNs to be more reliable, we should deploy at least two sinks, and data generated by nodes should be collected (convergecast) by all/many of the sinks. To reduce the number of collisions, many convergecast scheduling protocols have been proposed. However, the convergecast scheduling is done from many nodes to *one* sink.

Therefore, in this thesis, we have presented protocols that address the problems mentioned above. In particular, we have developed two algorithms that make data dissemination protocols tolerant to transient data faults that may corrupt values stored in the memory and messages. The algorithms are local which means that they are energy efficient and scalable, and thus can be used by integrating to a dissemination protocol without incurring big overhead in networks of different sizes. We have also developed a code update maintenance algorithm that is efficient in terms of latency and energy. The algorithm reduces energy consumption by minimizing the number of transmitted packets. These improvements are most prominent in networks with permanent and high number of asymmetric links. And finally, we have designed a data aggregation convergecast scheduling protocol that collects data from many nodes to two sinks. The proposed data aggregation algorithm reduces the energy cost by reducing the number of sleep-active transitions, minimizing the number of nodes that have multiple slots, and by balancing the number of slots. The remaining part of this chapter summarizes the contributions we made in this thesis and presents future directions.

7.1 Summary of Contributions

The main contributions presented in this thesis are as follows:

- We formalised the concept of code dissemination in WSN, and provide three refined specifications, viz.: strong, consistent and best effort code dissemination. Also we showed that (i) there is no deterministic algorithm that solve strong code dissemination in the presence of transient faults, and (ii) there is no deterministic 1-local algorithm that solve strong code dissemination in the presence of a stronger class of transient faults, called *detectable* faults.

- We presented two novel f-local algorithms called (i) BestEffort-Repair and (ii) Consistent-Repair that, when added to any fault-intolerant code dissemination protocol, solve (i) BestEffort code dissemination and (ii) Consistent code dissemination, and we proved the correctness of both protocols. Moreover, we ran real-world experiments and simulations to show their correctness and performance, especially the locality property of the protocols. Further, a case study where the two protocols were added to an existing code dissemination algorithm, called Varuna, was presented. The results of case study showed that both BestEffort-Repair and Consistent-Repair induce small overhead on Varuna in the presence of detectable transient faults. While Varuna alone in the presence of a transient fault resulted in all the nodes downloading the wrong code.

- We presented an energy efficient and fast code update maintenance algorithm called *Triva*. We conducted real-world experiments and simulations, and showed the performance of Triva over algorithms of this kind. The experiments showed that Triva is more energy efficient, faster and has considerable advantage when the network is event-based and when asymmetric links exist in the network.

- Finally, we presented an efficient data aggregation convergecast scheduling algorithm, which is, to the best of our knowledge, the first specifically designed for WSNs that have more than one sink. In particular, we designed a scheduling algorithm for WSNs with two sinks. We showed its correctness. Before scheduling the transmission and reception slots of nodes, the algorithm first connects two sinks by a shortest path and builds trees rooted at each node on the path. The scheduling is done in a way such that children of a node have contiguous transmission slots. We performed simulation and real-world experiments and showed the efficiency of the algorithm.

7.2 Future Work

We believe the following research questions could take the research done in this thesis further:

- BestEffort-Repair and Consistent-Repair tolerate transient data faults in code dissemination; future research may include *developing a dissemination protocol that tolerate Byzantine faults, where sensors and networks may behave in unexpected ways, without using cryptographic functions.*

- We have presented an efficient data aggregation convergecast scheduling algorithm for WSNs with two sinks. However, WSN applications with more than two sinks exist. Therefore, it is very interesting for us to see *an efficient solution for the data aggregation convergecast scheduling problem for WSNs with three or more sinks.*

- In our data aggregation convergecast scheduling algorithm, we have addressed the collision problem, however, we have not addressed the node failure and packet loss problem. So, one of the future works could be *developing a node failure tolerant data aggregation converegast scheduling protocol for WSNs with multiple sinks.* As in WSNs the energy consumption is important, addressing this problem as local as possible, i.e., using as small neighbours as possible, is desired.

Bibliography

[1] Tinyos. http://docs.tinyos.net.

[2] Tossim: Configuring a network. http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSSIM.

[3] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[4] K. Akkaya, M. Demirbas, and R. S. Aygun. The impact of data aggregation on the performance of wireless sensor networks. *Wireless Communications and Mobile Computing*, 8(2):171–193, Feb. 2008.

[5] I. F. Akyildiz and I. H. Kasimoglu. *Ad Hoc Networks*, 2(4):351–367, 2004.

[6] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[7] I. F. Akyildiz and M. C. Vuran. *Wireless Sensor Networks.* John Wiley & Sons, Ltd, 2010.

[8] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[9] V. Annamalai, S. Gupta, and L. Schwiebert. On tree-based convergecasting in wireless sensor networks. In *Wireless Communications and Networking (WCNC 2003)*, pages 1942–1947, 2003.

[10] A. Arora, M. Demirbas, and S. Kulkarni. Graybox stabilization. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2001)*, pages 389–398, 2001.

[11] A. Arora et al. A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605–634, 2004.

[12] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the* 18^{th} *International Conference on Distributed Computing Systems (ICDCS)*, 1998.

[13] N. Baccour, A. Koubaa, L. Mottola, M. Zuniga, H. Youssef, C. Boano, and M. Alves. Radio link quality estimation in wireless sensor networks: a survey. *ACM Transactions on Sensor Networks*, 8(4):34, 2012.

[14] C. R. Baker et al. Wireless sensor networks for home health care. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02*, AINAW '07, pages 832–837, Washington, DC, USA, 2007. IEEE Computer Society.

[15] S. Bapat and A. Arora. Stabilizing reconfiguration in wireless sensor networks. In *Proc. of IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC) 2006)*, pages 52–59, 2006.

[16] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In T. F. Abdelzaher, M. Martonosi, and A. Wolisz, editors, *SenSys*, pages 43–56. ACM, 2008.

[17] G. Barrenetxea, F. Ingelrest, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange. Sensorscope: Out-of-the-box environmental monitoring. In *IPSN*, pages 332–343. IEEE Computer Society, 2008.

[18] M. Bertier, A.-M. Kermarrec, and G. Tan. Message-Efficient Byzantine Fault-Tolerant Broadcast in a Multi-Hop Wireless Sensor Network. In *The 30th International Conference on Distributed Computing Systems*, Genoa, Italie, June 2010.

[19] J. Beutel, K. Römer, M. Ringwald, and M. Woehrle. Deployment techniques for wireless sensor networks. In G. Ferrari, editor, *Sensor Networks: Where Theory Meets Practice*, Heidelberg, 2009. Springer.

[20] V. Bhandari and N. H. Vaidya. On reliable broadcast in a radio network. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005*, pages 138–147, 2005.

[21] Y. Bo and J. Li. Minimum-time aggregation scheduling in multi-sink sensor networks. In *SECON*, pages 422–430. IEEE, 2011.

[22] J. E. Burns, M. G. Gouda, and R. E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[23] A. Cerpa, J. Elson, D. Estrin, and L. Girod. Habitat monitoring: application driver for wireless communications technology. In *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, 2001.

[24] K. Chakrabarty, S. Member, S. S. Iyengar, and E. Cho. Grid coverage for surveillance and target location in distributed sensor networks. *IEEE Transactions on Computers*, 51:1448–1453, 2002.

[25] T.-S. Chen, H.-W. Tsai, and C.-P. Chu. Adjustable convergecast tree protocol for wireless sensor networks. *Computer Communications*, 33(5):559–570, 2010.

[26] X. Chen, X. Hu, and J. Zhu. Minimum data aggregation time problem in wireless sensor networks. In *MSN*, pages 133–142, 2005.

[27] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 235–246. ACM, 2009.

[28] H. Choi, J. Wang, and E. A. Hughes. Scheduling for information gathering on sensor network. *Wireless Networks*, 15(1):127–140, Jan. 2009.

[29] M. Cinque, D. Cotroneo, C. D. Martino, S. Russo, and A. Testa. Avr-inject: A tool for injecting faults in wireless sensor nodes. In *IPDPS*, pages 1–8. IEEE, 2009.

[30] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 205–218, New York, NY, USA, 2007. ACM.

[31] Crossbow Technology Inc. Mote in-network programming user reference, www.tinyos.net/tinyos-1.x/doc/xnp.pdf. 2003.

[32] T. Dang, N. Bulusu, W. chi Feng, and S. Park. Dhv: A code consistency maintenance protocol for multi-hop wireless sensor networks. In U. Roedig

and C. J. Sreenan, editors, *EWSN*, volume 5432 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[33] M. Demirbas and S. Balachandran. Robcast: A singlehop reliable broadcast protocol forwireless sensor networks. In *27th International Conference on Distributed Computing Systems Workshops (ICDCS 2007 Workshops)*, pages 54–, 2007.

[34] A. Deshpande, C. Guestrin, and S. Madden. Resource-aware wireless sensor-actuator networks. *IEEE Data Engineering Bulletin*, 28(1):40–47, 2005.

[35] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[36] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda. Indriya: A low-cost, 3d wireless sensor network testbed. In *TRIDENTCOM*, volume 90 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 302–316. Springer, 2011.

[37] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

[38] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: Enabling Sustainable and Scalable Outdoor Wireless Sensor Network Deployments. In *Information Processing in Sensor Networks IPSN2006*, Apr. 2006.

[39] S. C. Ergen and P. Varaiya. Tdma scheduling algorithms for wireless sensor networks. *Wireless Networks*, 16(4):985–997, May 2010.

[40] A. Farahat and A. Ebnenasir. Exploiting computational redundancy for efficient recovery from soft errors in sensor nodes. In *SEKE*, pages 619–624. Knowledge Systems Institute Graduate School, 2011.

[41] F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Grenoble, France, Sept. 2004.

[42] E. N. Gilbert and H. O. Pollak. Steiner minimal tree. *SIAM Journal on Applied Mathematics*, 16:1–29, 1968.

[43] V. Handziski, A. Köpke, A. Willig, and A. Wolisz. Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proceedings of the 2Nd International Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality*, REALMAN '06, pages 63–70, New York, NY, USA, 2006. ACM.

[44] J. S. He, S. Ji, Y. Pan, and Y. Li. Constructing load-balanced data aggregation trees in probabilistic wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2013.

[45] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Symposium on Operating Systems Principles*, 2001.

[46] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, pages 174–185. ACM, 1999.

[47] F. Hermans, H. Wennerström, L. McNamara, C. Rohner, and P. Gunningberg. All is not lost: Understanding and exploiting packet corruption in outdoor sensor networks. In *EWSN*, pages 116–132, 2014.

[48] C.-K. Huang, G. Y. Chang, and J.-P. Sheu. Load-balanced trees for data collection in wireless sensor networks. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 474–479, Washington, DC, USA, 2012. IEEE Computer Society.

[49] S. C.-H. Huang, P.-J. Wan, C. T. Vu, Y. Li, and F. F. Yao. Nearly constant approximation for data aggregation scheduling in wireless sensor networks. In *26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA, INFOCOM*, pages 366–372, 2007.

[50] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM.

[51] Ö. D. Incel, A. Ghosh, B. Krishnamachari, and K. Chintalapudi. Fast data collection in tree-based wireless sensor networks. *IEEE Transactions on Mobile Computing*, 11(1):86–99, 2012.

[52] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *In Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[53] A. Jhumka. Crash-tolerant collision-free data aggregation scheduling for wireless sensor networks. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010)*, pages 44–53, 2010.

[54] A. Jhumka, M. Hiller, and N. Suri. Approach for designing and assessing detectors for dependable component-based systems. In *Proceedings of the 5th IEEE*

International Symposium on High Assurance Systems Engineering, pages 69–78, 2004.

[55] A. Jhumka and L. Mottola. Neighborhood monitoring and view consistency enforcement in wireless sensor networks. Technical Report TR2013.1321, Politecnico di Milano, 2013.

[56] G. Jolly and M. Younis. An energy-efficient, scalable and collision-free mac layer protocol for wireless sensor networks. *Wireless Communication and Mobile Computing*, 5(3):285–304, May 2005.

[57] N. R. Junior, M. A. M. Vieira, L. F. M. Vieira, and O. Gnawali. CodeDrip: Data Dissemination Protocol with Network Coding for Wireless Sensor Networks. In *Proceedings of the 11th European conference on Wireless sensor networks (EWSN 2014)*, Feb. 2014.

[58] R. M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[59] R. Kawano and T. Miyazaki. Distributed data aggregation in multi-sink sensor networks using a graph coloring algorithm. *Advanced Information Networking and Applications Workshops, International Conference on*, 0:934–940, 2008.

[60] B.-K. Kim, S.-K. Hong, Y.-S. Jeong, and D.-S. Eom. The study of applying sensor networks to a smart home. In *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume 01*, NCM '08, pages 676–681, Washington, DC, USA, 2008.

[61] S. Kim, S. Kim, and D. S. Eom. A robust and space-efficient stack management method for wireless sensor network os with scarce hardware resources. *International Journal of Distributed Sensor Networks (IJDSN)*, 2012, 2012.

[62] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 254–263, New York, NY, USA, 2007. ACM.

[63] Y. Kim, T. Schmid, Z. M. Charbiwala, J. Friedman, and M. B. Srivastava. Nawms: nonintrusive autonomous water monitoring system. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 309–322, New York, NY, USA, 2008. ACM.

[64] Y. Kim, H. Shin, and H. Cha. Y-mac: An energy-efficient multi-channel mac protocol for dense wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 53–63, Washington, DC, USA, 2008. IEEE Computer Society.

[65] C.-Y. Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 275–282, New York, NY, USA, 2004. ACM.

[66] C.-Y. Koo, V. Bhandari, J. Katz, and N. H. Vaidya. Reliable broadcast in radio networks: the bounded collision case. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC '06, pages 258–264, New York, NY, USA, 2006. ACM.

[67] E. Kranakis, D. Krizanc, and A. Pelc. Fault-tolerant broadcasting in radio networks. *Journal of Algorithms*, 39(1):47–67, 2001.

[68] B. Krishnamachari, D. Estrin, and S. B. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 575–578, Washington, DC, USA, 2002. IEEE Computer Society.

[69] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 64–75, New York, NY, USA, 2005. ACM.

[70] S. Kulkarni and L. Wang. Energy-efficient multihop reprogramming for sensor networks. *ACM Transactions on Sensor Networks*, 5(2):16:1–16:40, 2009.

[71] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A system for coarse grained memory protection in tiny embedded processors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 218–223, New York, NY, USA, 2007. ACM.

[72] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ransactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[73] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, Washington, DC, USA, 2006. IEEE Computer Society.

[74] A. Lédeczi, A. Nádas, P. Völgyesi, G. Balogh, B. Kusy, J. Sallai, G. Pap, S. Dóra, K. Molnár, M. Maróti, and G. Simon. Countersniper system for

urban warfare. *ACM Transactions on Sensor Networks*, 1(2):153–177, Nov. 2005.

[75] H. Lee and A. Keshavarzian. Towards energy-optimal and reliable data collection via collision-free scheduling in wireless sensor networks. In *INFOCOM*, pages 2029–2037, 2008.

[76] H. Lee, A. Klappenecker, K. Lee, and L. Lin. Energy efficient data management for wireless sensor networks with data sink failure. *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, 0:210, 2005.

[77] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[78] P. Levis, N. Lee, M. Welsh, and D. 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.

[79] P. Levis, N. Patel, D. Culler, and S. 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*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[80] C.-J. M. Liang, R. Musaloiu-E, and A. Terzis. Typhoon: A reliable data dissemination protocol for wireless sensor networks. In *European Conference on Wireless Sensor Networks (EWSN)*, January 2008.

[81] K. Lin and P. Levis. Data discovery and dissemination with dip. In *IPSN*, pages 433–444. IEEE Computer Society, 2008.

[82] R. P. Liu, Z. Rosberg, I. B. Collings, C. Wilson, A. Y. Dong, and S. Jha. Overcoming radio link asymmetry in wireless sensor networks. In *Proceedings of the IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2008, 15-18 September 2008, Cannes, French Riviera, France*, pages 1–5. IEEE, 2008.

[83] H. Lu, C. H. Foh, and J. Cai. Scalable data dissemination protocol for wireless sensor networks. In *Networks (ICON), 2012 18th IEEE International Conference on*, pages 471–476, Dec 2012.

[84] M. Macedo. Are there so many sons per node in a wireless sensor network data aggregation tree? *IEEE Communications Letters*, 13(4):245–247, 2009.

[85] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Operating Systems Review*, 36(SI):131–146, Dec. 2002.

[86] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM.

[87] A. Maurer and S. Tixeuil. On byzantine broadcast in loosely connected networks. *CoRR*, abs/1209.1358, 2012.

[88] A. Maurer and S. Tixeuil. On byzantine broadcast in planar graphs. *CoRR*, abs/1301.2875, 2013.

[89] L. Mottola and A. A. S. G.P. Picco. Figaro: Fine-grained software reconfiguration for wireless sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks*, pages 286–304, 2008.

[90] L. Mottola and G. P. Picco. Muster: Adaptive energy-aware multisink routing in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 10(12):1694–1709, 2011.

[91] L. Mottola, G. P. Picco, M. Ceriotti, c. Gună, and A. L. Murphy. Not all wireless sensor networks are created equal: A comparative study on tunnels. *ACM Transactions on Senor Networks*, 7(2):15:1–15:33, Sept. 2010.

[92] M. Nesterenko and S. Tixeuil. Ideal stabilization. *CoRR*, abs/0906.1947, 2009.

[93] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 151–162, New York, NY, USA, 1999. ACM.

[94] J. Paek, B. Greenstein, O. Gnawali, K.-Y. Jang, A. Joki, M. A. M. Vieira, J. Hicks, D. Estrin, R. Govindan, and E. Kohler. The tenet architecture for tiered sensor networks. *ACM Transactions on Sensor Networks*, 6(4), 2010.

[95] R. K. Panta, M. Vintila, and S. Bagchi. Fixed cost maintenance for information dissemination in wireless sensor networks. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 54–63, 2010.

[96] L. Paradis and Q. Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15(2), 2007.

[97] P. Paritosh, M. Kirk, R. Alistair, H. Ong, and J. Hart. Glacial environment monitoring using sensor networks. In *Real-World Wireless Sensor Networks, Stockholm, Sweden, 20 - 21 Jun 2005.*

[98] J. Polastre, R. Szewczyk, A. Mainwaring, D. Culler, and J. Anderson. Wireless sensor networks. chapter Analysis of Wireless Sensor Networks for Habitat

Monitoring, pages 399–423. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[99] V. Raghunathan, C. Schurgers, S. Park, M. Srivastava, and B. Shaw. Energy-aware wireless microsensor networks. In *IEEE Signal Processing Magazine*, pages 40–50, 2002.

[100] J. M. Reason and J. M. Rabaey. A study of energy consumption and reliability in a multi-hop sensor network. *ACM Mobile Computing and Communications Review*, 8:84–97, 2004.

[101] M. R. Saadat and G. Mirjalily. Efficient convergecast tree for data collection in wireless sensor networks. In *20th Iranian Conference on Electrical Engineering (ICEE)*, pages 1534–1539, 2012.

[102] E. Shih, B. H. Calhoun, S.-H. Cho, and A. Chandrakasan. Energy-efficient link layer for wireless microsensor networks. In *IEEE Computer Society Workshop on VLSI*, pages 16–21, 04/2001 2001.

[103] L. Sitanayah, K. N. Brown, and C. J. Sreenan. Multiple sink and relay placement in wireless sensor networks. In *Workshop on AI in Telecommunications and Sensor Networks, at ECAI 2012), Montpelier, France, August 2012.*

[104] W. Song, R. Huang, B. Shirazi, and R. LaHusen. Treemac: Localized tdma mac protocol for real-time high-data-rate sensor networks. *Pervasive Mobile Computing*, 5(6):750–765, Dec. 2009.

[105] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA, Center for Embedded Networked Computing, 2003.

[106] V. Subramonian, H. ming Huang, S. Datar, and C. Lu. C.lu, priority scheduling in tinyos - a case study. Technical report, Washington University, 2002.

[107] R. Szewczyk, J. Polastre, A. M. Mainwaring, and D. E. Culler. Lessons from a sensor network expedition. In *EWSN*, pages 307–322, 2004.

[108] P. Thulasiraman, S. Ramasubramanian, and M. Krunz. Disjoint multipath routing to two distinct drains in a multi-drain sensor network. In *INFOCOM*, pages 643–651. IEEE, 2007.

[109] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless micro-sensor network models. *ACM Mobile Computing and Communication Review*, 6(2), 2002.

[110] T.Masuzawa and S.Tixeuil. Stabilizing locally maximizable tasks in unidirectional networks is hard. In *Proceedings of International Conference on Distributed Computing Systems*, 2010.

[111] G. Tolle and D. E. Culler. Design of an application-cooperative management system for wireless sensor networks. In *European Conference on Wireless Sensor Networks*, pages 121–132, 2005.

[112] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 51–63, New York, NY, USA, 2005. ACM.

[113] M. Valero, M. Xu, N. Mancuso, W.-Z. Song, and R. A. Beyah. Edr^2: A sink failure resilient approach for wsns. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada.* IEEE, 2012.

[114] P.-J. Wan, K. M. Alzoubi, and O. Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. *Mobile Network Applications*, 9(2):141–149, Apr. 2004.

[115] Q. Wang, Y. Zhu, and L. Cheng. Reprogramming wireless sensor networks: challenges and approaches. *IEEE Network*, 20(3):48–55, 2006.

[116] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[117] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the International Conference on Information Processing in Sensor Networks IPSN*, 2006.

[118] Q. Wu, N. S. V. Rao, X. Du, S. S. Iyengar, and V. K. Vaishnavi. On efficient deployment of sensors on planar grid. *Computer Communications*, 30(14-15):2721–2734, Oct. 2007.

[119] Y. Wu, J. A. Stankovic, T. He, and S. Lin. Realistic and efficient multi-channel communications in wireless sensor networks. In *INFOCOM*, pages 1193–1201, 2008.

[120] B. Yu, J. Li, and Y. Li. Distributed data aggregation scheduling in wireless sensor networks. In *28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil, INFOCOM*, pages 2159–2167, 2009.

[121] G. V. Záruba, S. Basagni, and I. Chlamtac. Bluetrees-scatternet formation to enable bluetooth-based ad hoc networks. In *IEEE International Conference on Communications, ICC 2001, June 11-14, Helsinki, Finland*, pages 273–277, 2001.

[122] H. Zhang, A. Arora, Y. ri Choi, and M. G. Gouda. Reliable bursty convergecast in wireless sensor networks. *Computer Communications*, 30(13):2560–2576, 2007.