Efficient Fault-Tolerant Collision-Free Data Aggregation Scheduling for Wireless Sensor Networks[☆,☆☆]

Arshad Jhumka[a,1,*], Matthew Bradbury[a], Sain Saginbekov[a]

[a]*Department of Computer Science*
University of Warwick
Coventry CV4 7AL, UK

[☆]This work was supported by a grant from the University of Warwick.
[☆☆]This is an extended version of a paper [18] that was published in the Proceedings of the Symposium on Reliable Distributed Systems (SRDS), 2010.
[*]Corresponding author
Email addresses: arshad@dcs.warwick.ac.uk (Arshad Jhumka), M.Bradbury@warwick.ac.uk (Matthew Bradbury), sain@dcs.warwick.ac.uk (Sain Saginbekov)
[1]Tel: +44 24 7657 3780, Fax: +44 24 7657 3024

Abstract

This paper investigates the design of fault-tolerant TDMA-based data aggregation scheduling (DAS) protocols for wireless sensor networks (WSNs). DAS is a fundamental pattern of communication in wireless sensor networks where sensor nodes aggregate and relay data to a sink node. However, any such DAS protocol needs to be cognisant of the fact that crash failures can occur. We make the following contributions: (i) we identify a necessary condition to solve the DAS problem, (ii) we introduce a strong and weak version of the DAS problem, (iii) we show several impossibility results due to the crash failures (iv) we develop a modular local algorithm that solves *stabilising weak DAS* and (v) we show, through simulations and an actual deployment on a small testbed, how specific instantiations of parameters can lead to the algorithm achieving very efficient stabilization.

Keywords: Wireless Sensor Networks, Data Aggregation Scheduling, Fault Tolerance, Crashes, Collision Freedom, Impossibility, Correctness

1. Introduction

Data gathering is a basic capability expected of any wireless sensor network (WSN). The usual way of performing data gathering is to have nodes send their measurements (possibly over multiple hops) to a particular node called a *sink*. This type of communication, called *convergecast*, is fundamental to WSNs. Convergecast generally works by constructing a logical tree (called a convergecast tree) on top of the physical topology, with the sink located at the root, and data is then routed to the sink along the tree. However, to save energy, the data is typically aggregated along the route at specific nodes. This means that an aggregator node needs to have all the values from its children before aggregating the data. However, due to the broadcast nature of the communication medium, data transmissions need to be mediated among the children (and possibly other nodes) to avoid message collisions and interference, events that typically lead to energy exhaustion and which could also bias the aggregation.

Mediating these transmissions can be achieved through the use of an appropriate media access control (MAC) protocol. For WSN applications that need fast response times (e.g., disaster recovery), timeliness is of utmost importance. To this end, we investigate *Time Division Multiple Access (TDMA)-based MAC protocols for data aggregation scheduling*, in which each node is allocated a specific time slot in which it can transmit its data. Another advantage of a TDMA schedule for WSNs is that the transceivers can be turned on only when needed, thus saving energy. There exist several algorithms for convergecast in multi-hop radio networks, e.g., [19, 15, 23] that can be used for WSNs. A common pattern in TDMA-based convergecast algorithms is the decomposition of the problem into two independent subproblems: (i) a logical tree construction, and (ii) time slot allocation along the constructed tree. For example, the tree in [19] is constructed based on the positions of the nodes in a 2-D plane. Various objectives of convergecast scheduling algorithms exist, e.g., minimizing time for completing a convergecast [15], maximizing throughput [23], which determine the slot assignment (i.e., the schedule) along the tree.

When slots are assigned to nodes for data aggregation, in what we term as *data aggregation scheduling* (DAS), a node will first aggregate the data obtained from its children before relaying it to its parent. The objective, in this case, is that a node can only transmit a message *after* collecting data from *all* of its children. However, whenever a node crashes in a WSN (e.g., due to energy depletion), the values from a whole subtree disappears. Thus, it is important for a DAS algorithm to be fault-tolerant to ensure that correct nodes have a proper path to the sink, in the sense that their parent transmits the aggregated data after their own transmission.

1.1. Contributions

Several work has addressed the problem of convergecast, with a subset of these addressing the problem of data aggregation scheduling. However, to the best of our knowledge, no work has investigated the problem of *DAS in presence of crash failures*, on which we focus in this paper. Crash failures in WSNs can be brought about by, for example, defective hardware or battery exhaustion. In this context, we make an in-depth study of DAS in presence of crash failures. We make a number of contributions in three categories:

- Theory
    1. We identify a necessary condition for solving the data aggregation scheduling problem. This condition provides the theoretical basis that explain the structure of several data aggregation scheduling.
    2. We provide two variants of the DAS problem, namely (i) strong data aggregation scheduling, and (ii) weak data aggregation scheduling.

3. We show that it is impossible to solve the strong data aggregation scheduling problem.
4. We show that it is impossible to solve the weak data aggregation scheduling problem in presence of crash failures.
5. We introduce the problem of stabilising weak data aggregation scheduling and show that, in general, there is no 2-local algorithm that solves the problem.

- Algorithm

1. We develop a modular d-local algorithm that solves weak DAS and achieves efficient stabilization, where d is the diameter of the affected area.

- Results and Validation

1. Using both simulation and an actual deployment on a small testbed, we show that, under appropriate parameterization, the d-local algorithm can achieve 2-local stabilization.

Our paper is structured as follows: We present our system and fault models in Section 2. We formalize the problems of strong and weak data aggregation scheduling in Section 3. In Section 4, we focus on variants of the weak data aggregation convergecast. In Section 5, we present and prove the various impossibility results. In Section 6, we provide a d-local algorithm that achieves efficient stabilization. We present the performance of our algorithm in Section 7. In Section 8, we survey related work in the area, and put our work in the proper context. We discuss the impact of the results in Section 9. We finally summarise the paper in Section 10.

2. Models: System and Faults

2.1. Graphs and networks:

A wireless sensor node is a computing device that is equipped with a wireless interface and is associated with a unique identifier. A wireless sensor network (WSN) consists of a set of wireless sensor nodes that communicate among themselves via their wireless interface. Communication in wireless networks is typically modelled with a circular communication range centered on a 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 then typically modelled as an undirected graph $G = (V, E)$ where V is a set of Γ wireless sensor nodes and E is a set of edges or links, each link being a pair of distinct nodes. Two nodes $m, n \in V$ are said to be 1-hop neighbours (or neighbours) iff $(m, n) \in E$, i.e., m and n are in each other's communication range. We denote by M the set of m's neighbours, and we denote by M^d, the d-hop neighbourhood of m. We say that two nodes m and n can collide at node p if $(p \in M) \wedge (p \in N)$[2]. In general, two nodes m and n can collide if they are in the 2-hop neighbourhood of each other. We then define the collision group of a node n as follows:

$$CG(n) = \{m \in V | ((n, m) \in E) \vee (2hopN(m, n))\},$$

where $2hopN(m, n)$ is a predicate that returns true if m, n are in each other's 2-hop neighbourhood.

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

We denote by Δ_m the degree of node m, i.e., the size of M. We also denote by Δ_G, the degree of G, i.e., $\Delta_G = max(\{\Delta_m, m \in V\})$. We also denote by η_G, the maximum of nodes at any hop distance in G. We assume a distinguish node $S \in V$, called a *sink*. A path of length k is a sequence of nodes $n_k \dots n_0$ such that $\forall j, 0 < j \leq k$, n_j and n_{j-1} are neighbours. We say a path $n_k \dots n_0$ is an S-path if $n_0 = S$. The path $n_k \dots n_0$ is said to be *forward* if $\forall i, j, 0 < i \leq j \leq k, n_i \neq n_j$. A path $n_k \dots n_0$ is called a *cycle* if the path is forward and $n_0 = n_k$. In this paper, we focus on forward S-paths (henceforth, paths). Specifically, we are only interested in paths from a node to the sink, hence forward S-paths. For an S-path $n_k \dots S$, we say that n_k has an S-path.

Given an undirected graph $G = (V, E)$: G is connected iff there exists a path in G between each pair of distinct nodes. In general, we are only interested in paths that end with the sink. We say that G is S-connected iff every node in G has an S-path, and we say that G is S^k-connected iff G is S-connected, and all nodes have k node-disjoint S-paths. Two paths are node-disjoint only if the end nodes of the two paths are the same while all other nodes differ. In this paper, when we mention disjoint paths, we mean node-disjoint paths. The distance between two nodes m and n in G is the length of the smallest path between m and n in G. We denote the distance between m and n by $d(m, n)$. The diameter D_G of G is equal to max$(\{d(m, n), m \in V \wedge n \in V\})$.

2.2. Distributed programs

2.2.1. Syntax

We model the processing on a WSN node as a process containing non-empty sets of variables and actions. A distributed program is thus a finite set of N communicating processes. We represent the communication network topology of a distributed program by an undirected connected graph $G = (V, E)$, where V is the set of N processes and E is a set of edges such that $\forall m, n \in V$, $(m, n) \in E$ iff m and n can directly communicate together, i.e., nodes m and n are neighbours.

Variables take values from a fixed domain. We denote a variable v of process n by $n.v$. Each process n has a special channel variable ch, denoted by $n.ch$, 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 channel variable, has a set of initial values. The *state* of a program is an assignment of values to variables 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 space of the program is the set of all possible value assignments to variables.

An action at a process p has the form $\langle name \rangle :: \langle guard \rangle \rightarrow \langle command \rangle$. In general, a guard is a state predicate defined over the set of variables of p and possibly of p's neighbours. When the guard of an action evaluates to $true$ in a state s, we say the action is enabled in s[3], and the corresponding $command$ can be executed, which takes the program from state s to another state s', by updating variables of process p. We assume that the execution of any action is atomic. A $command$ is a sequence of assignment and branching statements. Also, a $guard$ or $command$ can contain universal or existential quantifiers of the form: $(\langle quantifier \rangle \langle bound\ variables \rangle : \langle range \rangle : \langle term \rangle)$, where $range$ and $term$ are boolean constructs. A special $timeout(timer)$ guard evaluates to true when a timer variable reaches zero, i.e., the timer expires. A $set(timer, value)$ command can be used to initialise the timer variable to a specified value.

[3]By extension, we say a process p is enabled in state s if at least one of its actions is enabled in state s.

2.2.2. Semantics

We model a distributed program ϕ as a transition system $\phi = (\Sigma, I, \tau)$, where Σ is the state space, $I \subseteq \Sigma$ the set of initial states, and $\tau \subseteq \Sigma \times \Sigma$ the set of transitions (or steps). A computation of ϕ is a maximal sequence of states $s_0 \cdot s_1 \ldots$ such that $s_0 \in I$, and $\forall i > 0, (s_{i-1}, s_i) \in \tau$. A state s is terminal if there is no state s' such that $(s, s') \in \tau$. We denote reachability of state s' from s as $s \rightarrow^* s'$, i.e., there exists a computation that starts from s and contains s'.

In a given state s, several processes may be enabled, and a decision is needed to decide which one(s) execute. A *scheduler* is a predicate over the computations. In any computation, each step (s, s') is obtained by the fact that a non-empty subset of enabled processes in s atomically execute an action. This subset is chosen according to the scheduler. A scheduler is said *central* [9] if it chooses *only one* enabled process to execute an action in any execution step. A scheduler is said to be *distributed* [5] if it chooses *at least one* enabled process to execute an action in any execution step. Oan the other hand, a synchronous scheduler is a distributed scheduler where *all* enabled processes are chosen to execute an action in a step.

A scheduler may also have some fairness properties [10]. A scheduler is *strongly fair* if every process that is enabled infinitely often is chosen infinitely often to execute an action in a step. A scheduler is *weakly fair* if every continuously enabled process is eventually chosen to execute an action in a step. In this paper, we assume a *synchronous scheduler*, capturing a synchronous system where an upper bound exists on the time for a process to execute an action [21]. In other words, it can capture systems where processes execute in a kind of lock-step fashion. This assumption is not unreasonable as WSNs are often time-synchronized [25] to either correlate sensor readings from different devices or for time-based protocols such as TDMA [16].

2.2.3. Specification

A specification is a set of computations. Program ϕ *satisfies* specification ¶ if every computation of ϕ is in ¶. Alpern and Schneider [1] 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, whereas a liveness specification states that something good will eventually happen. Formally, the safety specification identifies a set of finite computation prefixes that should not appear in any computation. A liveness specification identifies a set of computation suffixes such that every computation has a suffix in this set.

2.2.4. Communication

An action with a $rcv(msg, sender)$ guard is enabled at process p when there is a message at the head of the channel $p.ch$, i.e., $p.ch \neq \langle\rangle$. Executing the corresponding command causes the message to be dequeued from the $p.ch$, while msg and $sender$ are bound to the content of the message and the identifier of the sender node. With a weak or strong fairness assumption, a stand alone $rcv(msg, sender)$ guard implies that the message msg will eventually be delivered. Differently, the $BCAST(msg)$ (or $send(msg)$) command causes message msg to be enqueued to the channel $q.ch$ of all processes q that are neighbours of p. We model synchronous communication as follows: after a process m sends a message to another process n in state s_i, the message is in $n.ch$ in state s_{i+1}. Overall, in this paper, we assume a synchronous system model.

2.3. Faults

A fault model stipulates the way programs may fail. We consider *crash* failures that causes a program to stop executing instructions. Formally, a crash failure can be modeled through the use of auxiliary variables, and appropriate fault (crash) actions that change the value(s) of the auxiliary variables [2]. For example, a process p can have an auxiliary variable *up*,

where $p.up = 0$ means process p has crashed, and $p.up = 1$ otherwise. A crash action changes the value of the auxiliary variable. From the example, the crash action will change variable *up* from 1 to 0. A crash occurs if a crash action is executed. Crash actions can interleave program actions and they might or might not be executed when enabled. We say a computation is crash-affected if the computation contains program and crash transitions. In this paper, we assume that the sink node does not crash. We also assume that any crash does not partition the network. A program is *self-stabilizing* iff, after faults have stopped occurring, the program eventually reaches a state from where its specification is satisfied.

2.4. Time Division Multiple Access

TDMA is a technique whereby the timeline is split into a series of time periods (or periods), and each period is divided into a series of time slots. In each period, every node is assigned a slot in which the node transmits its message. Slots have to be assigned carefully to avoid message collisions. Specifically, nodes that are in each others' collision groups cannot transmit in the same slot.

Definition 1 (Non-colliding slot). *Given a network G, a node $n \in G$, and a slot i in which n can transmit its payload, we say that i is* non-colliding for n *iff* $\forall m \in CG(n) \cdot m.slot \neq i$.

Definition 2 (F-affected node and F-affected area). *Given a network $G = (V, E)$ and a fault F that causes up to f crashes in G, a node is said to be* F-affected *(or affected) in a program state if the node will need to change its state to make the program state consistent. An* F-affected area *(or affected area) of a program state is a maximal set of connected F-affected nodes in that state.*

Definition 3 (d-local self-stabilizing algorithm). *Given a network $G = (V, E)$, a problem specification ¶ for G, and a self-stabilizing algorithm A that solves ¶. Algorithm A is said to be* d-local self-stabilizing *for ¶ iff, for every affected node n, messages that correct the state of the affected node originate from its N^d neighbourhood.*

3. Data Aggregation Scheduling Definitions

In this section, we present two variants of the data aggregation scheduling, namely (i) the strong data aggregation scheduling, and (ii) the weak data aggregation scheduling. Recall that we are focusing on forward S-paths, i.e., paths that end with a sink. For data aggregation scheduling, as argued before, nodes closer to the sink should transmit later than nodes further away from a sink (in what we call as proper slot assignment).

We first provide two definitions.

Definition 4 (Strongly and Weakly Proper Slot). *Given a graph $G = (V, E)$, a slot i allocated to a node n, i.e., $n \in \rho_i$, is said to be* strongly proper for n in G *if and only if* $\forall m \in N, n \cdot m \ldots S$ *is a path*: $\exists j > i : m \in \rho_j$. *Also, a slot i allocated to a node n, i.e., $n \in \rho_i$, is said to be* weakly proper for n in G *if and only if* $\exists m \in N, n \cdot m \ldots S$ *is a path*: $\exists j > i : m \in \rho_j$.

We will only say a slot is strongly or weakly proper if n and G are clear from the context.

3.1. Strong Data Aggregation Scheduling

We define the problem of strong data aggregation scheduling problem. The intuition here is, in a network where up to f crash failures can occur, there should be at least $(f + 1)$ disjoint paths from any node to the sink for the network to remain connected. Then, for strong DAS, we require that every path has a proper slot assignment, i.e., for every path $n \dots n_i \cdot n_{i+1} \dots S$ from a node n to sink S, the slot for n_{i+1} is greater than the slot for n_i. Formally, the strong data aggregation scheduling is defined as follows:

Definition 5 (Strong data aggregation scheduling). *Given a network $G = (V, E)$, a strong TDMA-based collision-free data aggregation schedule is a sequence of sets of senders $\langle \rho_1, \rho_2, \dots, \rho_l \rangle$ satisfying the following constraints:*

1. $\rho_i \cap \rho_j = \emptyset, \forall i \neq j$,
2. $\bigcup_{i=1}^{l} \rho_i = V \setminus \{S\}$,
3. $\forall n \in \rho_{1 \leq i \leq l-1} : \forall m \in N, n \cdot m \dots S$ *is a path:* $(m \in \rho_{j>i}) \vee (m = S)$,
4. $\forall \rho_{1 \leq j \leq l-1} : \forall m, n \in \rho_j : m \neq n \Rightarrow n \notin CG(m) \wedge m \notin CG(n)$.

There are four conditions that capture the strong data aggregation scheduling problem. These conditions are explained here:

1. The first condition stipulates that nodes are allocated at most one time slot in the schedule.
2. The second condition implies that all nodes (apart from the sink) will get at least one transmission slot. Taken together, conditions 1 and 2 state that every node (except the sink) will transmit in exactly one slot in the schedule.
3. The third condition captures the fact that whenever a node transmits a message, *all* of its neighbours that are closer to the sink will transmit in a later slot, and the condition captures the notion of *strongly proper slots*.
4. The last condition captures the condition for collision freedom, i.e., two nodes can transmit in the same slot only if they are not in each other's collision group.

Overall, nodes in ρ_1 transmit first, followed by those in ρ_2, and so on, until those in ρ_l. The parameter l is called the data aggregation *latency*. Observe that $|\rho_l| = 1$, since if $|P_l| \geq 1$, there will be a message collision at S.

Given a network $G = (V, E)$, an algorithm that produces a slot assignment that satisfies the above requirements for G is said to solve the *collision-free strong data aggregation scheduling* (or simply the strong data aggregation scheduling) problem for G, and the schedule is termed a *strong proper schedule for G*.

3.2. Weak Data Aggregation Scheduling

In this section, we formally define the problem of weak data aggregation scheduling.

Definition 6 (Weak data aggregation scheduling). *Given a network $G = (V, E)$, a weak TDMA-based collision-free data aggregation schedule is a sequence of sets of senders $\langle \rho_1, \rho_2, \dots, \rho_l \rangle$ satisfying the following constraints:*

1. $\rho_i \cap \rho_j = \emptyset, \forall i \neq j$,
2. $\bigcup_{i=1}^{l} \rho_i = V \setminus \{S\}$,
3. $\forall n \in \rho_{1 \leq i \leq l-1} : \exists m \in N, n \cdot m \dots S$ *is a path:* $(m \in \rho_{j>i}) \vee (m = S)$,
4. $\forall \rho_{1 \leq j \leq l-1} : \forall m, n \in \rho_j : m \neq n \Rightarrow n \notin CG(m) \wedge m \notin CG(n)$.

As for the case of strong data aggregation scheduling, conditions 1 and 2, taken together, state that every node (except the sink) will transmit in exactly one slot in the schedule. The third condition captures the fact that whenever a node transmits a message, *at least one* of its neighbours closer to the sink will transmit in a later slot, and the condition captures the notion of weakly proper slots. The last condition captures the condition for collision freedom, as in the case of strong data aggregation scheduling.

Given a network $G = (V, E)$, an algorithm that produces a schedule that satisfies the above requirements for G is said to solve the *collision-free weak data aggregation scheduling* problem for G, and the schedule is termed a *weak proper schedule for* G. Similarly, the parameter l is the latency of the data aggregation scheduling.

4. Data Aggregation Scheduling Specifications

Given the definition for weak data aggregation scheduling (Definition 6), there are possibly different ways of producing a weakly proper schedule. In this paper, we focus on two specific ways, namely (i) a deterministic way, and (ii) a stabilising way. These two possibilities capture two possible behaviours of algorithms that can generate a weakly proper data aggregation schedule. The specifications are described following the notion that a trace-based specification is the intersection of a safety specification and a liveness specification [1].

The two specifications we provide are similar in ideas to the notions of perfect failure detectors and eventually perfect failure detectors, a la Chandra and Toueg [6], although in a very different context. Specifically, the deterministic specification captures the fact that a node is only assigned a slot that is weakly proper and non-colliding, and that the slot is permanent, i.e., never changed. On the other hand, in the stabilising specification, we permit a node to change slots, with the exception that eventually a weakly proper and non-colliding slot will be permanently assigned.

4.1. Deterministic Weak Data Aggregation Scheduling

In this section, we provide the specification for *deterministic weak data aggregation scheduling* (Definition 7).

Definition 7 (Deterministic Weak DAS). *Given a network $G = (V, E)$, a program A solves the* deterministic weak data aggregation scheduling *problem (Definition 6) for G if every computation of A satisfies the following:*

1. Safety*: A node n in G is allocated a permanent slot only if the slot is weakly proper and non-colliding for n in G.*
2. Liveness*: Eventually, every node n in G is allocated a slot.*

Whenever G is obvious from the context, we will omit its use. Safety prevents a node from being assigned any arbitrary slot, while liveness requires every node to be eventually allocated a slot. Taken together, it captures the notion that all nodes will eventually be permanently allocated non-colliding weakly proper slots, which then generate the weak data aggregation schedule for G.

4.2. Stabilising Weak Data Aggregation Scheduling

We now define a weaker version of the deterministic specification, namely the stabilising version (Definition 8).

Definition 8 (Stabilising Weak Scheduling). *Given a network G, a program A solves the* stabilizing weak data aggregation scheduling *problem (Definition 6) for G if every computation of A satisfies the following:*

1. Safety*: Eventually, a node n in G is permanently allocated a slot only if the slot is weakly proper and non-colliding for n in G.*
2. Liveness*: Eventually, every node n in G is allocated a slot.*

The stabilising version of the data aggregation scheduling problem allows an algorithm A to make a finite number of mistakes during slot allocation (in contrast, no mistakes are allowed in the deterministic version). The mistakes may be due to the fact that the slots allocated are not weakly proper, which could be due to topology changes, incomplete information etc. Eventually, the mistakes will stop, for example when faults stop, and a weak data aggregation schedule will be eventually obtained.

In the next section, we investigate the possibilities and impossibilities of achieving strong and weak data aggregation convergecast schedules.

5. Research Contributions

In this section, we present some of the important research contributions we make in this paper. These are:

1. Our first contribution (Section 5.1) is the identification of a necessary condition that helps solve the strong or weak DAS problem.
2. We then show, in Section 5.2, that it is impossible to solve the strong data aggregation scheduling problem.
3. In Section 5.3, we show that it is impossible to solve the deterministic weak data aggregation scheduling problem in presence of crash failures.
4. We then show, in Section 5.4, that there exists no 2-local algorithm that solves the stabilising proper DAS problem.

5.1. Necessary Condition to Solve Strong and Weak Data Aggregation Scheduling

Before attempting to solve the data aggregation scheduling problem, we wish to determine the type of information required. In this context, our first main contribution is a necessary condition to solve the strong or weak DAS, as captured by Theorem 1.

Intuition When a node n has to decide on a slot, it needs to determine which of its neighbours are on a path to the sink. If a node cannot determine this, i.e., it cannot discriminate among its neighbours, then it is difficult for n to decide on a strongly or weakly proper slot, thus the data aggregation scheduling problem becomes difficult to solve.

Theorem 1 (Necessary condition). *Given a network $G = (V, E)$, there exists no deterministic algorithm A that solves the strong/weak DAS problem if $\exists n \in (V \setminus \{S\})$, n cannot choose a node $m \in N$ such that $n \cdot m \ldots S$ is a path.*

Proof

We assume the existence of the algorithm A that solves the strong/weak DAS problem when a node n cannot choose a node $m \in N$ such that $n \cdot m \ldots S$ is a path, and show a contradiction.

Given a graph $G = (V, E)$. Assume a node $n \in V \setminus \{S\}$ with neighbourhood N, and a unique node $m \in N$ with $n \cdot m \ldots S$ a path in G. Since A solves the weak/strong DAS problem, A identifies $m \in N$ as the neighbour on the path from n to S, with $m.slot > n.slot$.

Now, given another graph $G' = (V, E')$, where G' differs from G as follows: the nodes are labelled such that N in G' is exactly as in G, and that $n \cdot m \ldots S$ is no longer a path in G', but $m \cdot n \ldots S$ is. Since N is the same in G' as in G and A is

deterministic, A will again choose m as being on the path from n to S (since nodes cannot determine which of its neighbour is on the path to S). However, as $m.slot > n.slot$, then this violates the strong/weak DAS requirement. Hence, A cannot exist. □

Theorem 1 states that nodes need to be able to determine which node, among their neighbours, is on the path to the sink. This is only an abstract requirement, and there can be various possible implementations for this. In fact, Theorem 1 provides an insight into the design of existing convergecast and data aggregation scheduling algorithms [19, 15]. These algorithms in [19, 15] indeed provided different possible implementations of the abstract requirement identified in Theorem 1. In [19], the authors used classes of graphs (such as grids) and node locations to infer the direction of the sink and, hence, their neighbours on an S-path. In [15], the authors used hop distance to allow a node to determine which of its neighbours is on the path to the sink.

5.2. Strong Data Aggregation Scheduling

The scale and nature of wireless sensor networks often mean that failures of sensor nodes will not be rare events. Thus, it is important for data aggregation and message relay to proceed in spite of these failures. Therefore, solving the strong data aggregation scheduling problem is important as it enables any node to forward its aggregated data, in spite of various node (hence path) failures. Our second major contribution is to determine the feasibility of obtaining a strong data aggregation schedule, as captured by Theorem 2. The main implication of a strong data aggregation schedule is that, if a node has a strongly proper slot, then the schedule is inherently resilient to a certain number of crash failures.

Intuition Given a network $G = (V, E)$ in which every node has at least two disjoint paths to the sink. Consider three nodes $m, n, p \in V$. Now, assume a path r_1 for p is $p \cdot n \cdot m \ldots S$, while a path r_2 for m is $m \cdot n \cdot p \ldots S$. From the definition of strong data aggregation scheduling (Definition 5), if node p has a strongly proper slot, then in r_1, the slot of n is greater than that of p. However, in r_2, the slot of n is smaller than that of p, violating the strong data aggregation scheduling specification.

Theorem 2 (Impossibility - Strong DAS). *Given a network $G = (V, E)$ in which every node $n \in V$ has at least two disjoint paths to the sink. Then, there exists no deterministic algorithm that can generate a strong data aggregation schedule.*

Proof

We assume the existence of the algorithm A that can generate a strong data aggregation schedule, and then show a contradiction.

Consider three nodes $m, n, p \in V$, and they have $r_m, r_n, r_p \geq 2$ disjoint S-paths respectively. Assume that $m \cdot n \cdot p \ldots S$ is a path in G. Since A generates a strong data aggregation schedule, then the slot for n is greater than the slot for node m.

Now, since $r_m \geq 2$ and one path for m is $m \cdot n \cdot p \ldots S$, then there exists a path for p in G such that $p \cdot n \cdot m \ldots S$. On this path, since A generates a strong data aggregation schedule, then the slot assigned to n is less than that assigned to node m. This is a contradiction from the previous observation. Hence, A cannot exist. □

The implication of Theorem 2 is that it is impossible, in general, to generate a strong data aggregation schedule. On the other hand, it can be observed that a weak data aggregation schedule can be generated. However, understanding the generation of weak data aggregation schedules in presence of crash failures is important, so as to capture the resilience of such schedules, and of the complexity of generating them.

5.3. Deterministic Weak Data Aggregation Scheduling in presence of Crash

In this section, we investigate the possibility of solving the deterministic weak data aggregation scheduling problem in presence of crash failures, where nodes stop executing program instructions. To the best of our knowledge, no work has addressed the problem of weak data aggregation scheduling problems in the presence of crash failures. We show that it is impossible to solve the deterministic weak data aggregation scheduling problem in presence of crash failures.

Intuition From the specification of deterministic weak DAS, it is stated that nodes are only ever assigned weakly proper slots. However, when a node is being assigned a weakly proper slot and a crash failure occurs along the relevant path r (which makes the slot weakly proper), then the slot is no longer weakly proper as r no longer exists.

Thus, the third main contribution of the paper is captured by Theorem 3, which states that it is impossible to solve deterministic weak data aggregation scheduling in presence of crash failures. From Theorem 1, we assume that nodes can determine which of their neighbours are on the path to the sink.

Theorem 3 (Impossibility - Deterministic Weak DAS). *Given a network $G = (V, E)$ in which every node $n \in V$ has at least $(f + 1)$ disjoint paths to the sink, a fault model F where up to f crash failures can occur. Then, there exists no algorithm that can solve the deterministic weak data aggregation scheduling problem for G in the presence of F.*

Proof

Assume an algorithm A that solves the deterministic weak data aggregation scheduling problem for G, and show a contradiction.

Consider three nodes $m, n, p \in V$, and they have $r_m, r_n, r_p \geq f + 1$ S-paths respectively. Assume that $(m \cdot n_1 \cdot p_1 \dots S) \dots (m \cdot n_{r_m} \cdot p_{r_m} \dots S)$ are the r_m paths for m in G. Consider a state s in a computation σ of A, where node m is assigned a weakly proper slot (for all the paths $(m \cdot n_2 \cdot p_2 \dots S) \dots (m \cdot n_{f+1} \cdot p_{f+1} \dots S)$), i.e., node m has a slot which is less than those of nodes $n_2 \dots n_{f+1}$, resulting in state s'.

Now, consider a faulty computation σ' of A which is identical to σ up to state s. In state s, nodes $p_2 \dots p_{f+1}$ crash. Since σ' is identical to σ up to s and A is deterministic, then node m chooses a slot which is less than that of nodes $n_2 \dots n_{f+1}$, resulting is state s'. However, because of nodes $p_2 \dots p_{f+1}$ crashing, the paths $(m \cdot n_2 \cdot p_2 \dots S) \dots (m \cdot n_{f+1} \cdot p_{f+1} \dots S)$ no longer exist. Hence, the slot for node n is no longer weakly proper for G', which is identical to G except without nodes $p_2 \dots p_{f+1}$. Hence, A does not assign a weakly proper slot to m in σ', which is a contradiction. Hence, no such A exists. □

Theorem 3 states that, in presence of crash failures, it is impossible to solve the deterministic weak data aggregation scheduling problem. Specifically, this happens due to the fact that nodes can mistakenly be assigned slots that are not weakly proper, due to the crash failures, as the nodes are unaware of the failures when they are choosing slots. Thus, to allow for these possible mistakes to be corrected, we weaken the specification, and consider the stabilising version of the problem, viz. stabilising weak data aggregation scheduling problem. The area of self-stabilisation is quite mature [10]. A self-stabilising algorithm is one that eventually satisfies its specification, in the sense that the safety property is eventually satisfied. In the next section (Section 5.4), we investigate on the possibility of achieving stabilising weak data aggregation scheduling in presence of crash failures.

5.4. Stabilising Weak Data Aggregation Scheduling in presence of Node Crashes

Because mistakes can be made during slot assignment due to node crashes, the specification for stabilising weak data aggregation scheduling allows for this to occur only finitely. It is stated that, eventually, every node will have weakly proper

slots, allowing a finite number of mistakes to be made. From Theorem 1, we know that nodes only need to know at least one of their neighbours on a path to the sink. Thus, we investigate the problem of whether a 2-local algorithm can solve the stabilising weak data aggregation scheduling, i.e., achieving stabilising weak data aggregation scheduling using information about the 2-hop neighbourhood only. 1-local algorithms cannot be used as collision-free TDMA scheduling needs distance-2 colouring type algorithms. 2-local algorithms are important as they make use of local interactions only [17], thus helping reduce energy consumption, thereby prolonging network lifetime.

Intuition In the specification for stabilising weak data aggregation scheduling, nodes are allowed to make a finite number of mistakes when deciding on proper slots. However, since a crash can happen beyond the 2-hop neighbourhood of a node, then the information held by a node needs to be beyond its 2-hop neighbourhood. Hence, solving stabilising weak data aggregation scheduling by 2-local algorithms is impossible. Specifically, a node needs to have a network wide information about crashes to determine which of its neighbouring nodes has a path to the sink. This is captured by Theorem 4.

Theorem 4 (Impossibility). *Given a network $G = (V, E)$ in which every node $n \in V$ has at least $(f + 1)$ disjoint paths to the sink, a fault model F where up to f crash failures can occur. Then, there exists no 2-local algorithm that can solve the stabilising weak DAS problem in the presence of F.*

Proof:

We assume an algorithm A that solves the stabilizing weak data aggregation scheduling problem and that is 2-local, and show a contradiction.

Consider a node $m \in V$, and it has $r_m \geq f+1$ S-paths respectively. Assume that $(m \cdot n_1 \cdot p_1 \cdot c_1 \ldots S) \ldots (m \cdot n_{r_m} \cdot p_{r_m} \cdot c_{r_m} \ldots S)$ are the r_m disjoint paths for m in G. Consider a state s in a computation σ of A, where the following happens: (i) node m is assigned a weakly proper slot for the path $m \cdot n_1 \cdot p_1 \cdot c_1 \ldots S$ only, i.e., node m has a slot which is less than that of node n_1 and (ii) node c_1 crashes, resulting in state s'. Now, node m does not have a weakly proper slot, as path $m \cdot n_1 \cdot p_1 \cdot c_1 \ldots S$ does not exist anymore.

Since A solves the stabilizing weak data aggregation scheduling problem, then there exists a state $s'' > s'$ where node m will have a weakly proper slot for G' (similar to G without the crashed node c_1) again. To recompute a weakly proper slot, node m has learnt about the crash of node c_1. However, c_1 was 3 hops away from m, meaning A is a 3-hop algorithm. This is a contradiction. Hence, no such A exists. □

Since there is no 2-local algorithm that can solve the stabilising weak data aggregation scheduling, it is obvious that a d-local algorithm exists, where d is the diameter of the affected region, since every node needs to know about node failures as far away as where failures have occurred, i.e., up to d hops away.

6. A Crash-Tolerant Collision-Free Data Aggregation Scheduling Algorithm

In this section, we present a modular d-local algorithm that solves for stabilizing weak data aggregation scheduling. We will also show that, under specific parameterization, the algorithm can achieve 2-local stabilisation.

6.1. An Efficient d-local Algorithm for Solving Stabilising Weak DAS

Working of d-local algorithm (Figures 1 and 2): Figure 1 shows the variables of the algorithm, while Figure 2 shows the actions of the algorithm.

process j
variables

% Slots known to process j; id, set of neighbours, slot, hop
slots: set of (int, set of int, int, int) init $\emptyset$

% Slot of process j, Id of parent of a node, Hop of process j
slot, parent, hop: int init $\perp, \perp, \perp$

% Live processes in a round
live: set of int init $\emptyset$;

% Neighbours of process j
N[j]: set of ids init J;

% Start flag, Slot collection flag
start, c: bool init $1, 0$;

% A timer that sets the wave period
wave, *beacon*: timer init δ, λ;

constants

% Size of network
Δ: int;

Figure 1: Variables of d-local algorithm that solves stabilising Weak DAS for a general network.

actions

% **Diffusing slot information**
dissem:: **timeout**(*wave*)$\rightarrow$
slots := $\emptyset$;
if (**sink**(*j*) $\wedge$ (*start* = 1)) **then**
BCAST$\langle j, N[j], \Gamma, 0\rangle$;
start := 0;
elseif ($\neg$**sink**(*j*) $\wedge$ (*slot* $\neq \perp$)) **then**
BCAST$\langle j, N[j] \setminus \{parent\}, slot, hop\rangle$;
fi;
set($wave$, δ);

% **Sending neighbourhood info.**
neighbourS:: **timeout**(*beacon*)$\rightarrow$
BCAST$\langle j\rangle$;
set($beacon$, λ);

% **Collecting neighbourhood through beacon messages.**
neighbourC:: **rcv**$\langle n\rangle \rightarrow$
live := *live* $\cup\{n\}$;

% **Receiving information from neighbour nodes.**
receive:: **rcv**$\langle n, N, s, h\rangle \rightarrow$
c, live, slots := 1, *live* $\cup\{n\}$, *slots* $\cup\{(n, N, s, h)\}$;

% **Resolving slot conflicts.**
resolve:: **rcv**$\langle coll, S\rangle \rightarrow$
if ($j \in S \wedge$ *slot* $\neq \perp$) **then**
slot := *slot* $-$ rank(*j*, *S*) + 1;
fi;

% **All slots msgs processed.**
check:: **rcv**$\langle\rangle \rightarrow$
if ($c \neq 1 \wedge$ *parent* $\neq \perp \wedge$ *parent* $\notin$ *live*) **then**
slot, parent, hop := $\perp, \perp, \perp$;
BCAST$\langle crash, j\rangle$;
N[j] := *live*;
fi;
if ($c = 1$) **then**
if (*slot* $\neq \perp$) **then**
slots := *slots* $\cup$ (*j*, *N[j]*, *slot*, *hop*);
fi;
if ($\exists s \cdot |\{n|(n, N, s, h)\}| \geq 2$) **then**
$\forall s \cdot |\{n|(n, N, s, h)\}| \geq 2 :$
BCAST$\langle coll, \{n|(n, N, s, h) \in slots\}\rangle$;
elseif ($\neg$**sink**(*j*) $\wedge$ (*slot* $= \perp \vee$ *parent* $= \perp \vee$ *hop* $= \perp$)) **then**
hop := min$\{h|(n, N, s, h) \in slots\} + 1$;
slot := min$\{s|(n, N, s, hop - 1) \in slots\}$ $-$ rank(j, N);
parent := choose$\{n|(n, N, slot+\text{rank}(j, N), hop-1) \in slots\}$;
fi;
$c := 0$;
fi;
live := $\emptyset$;

% **Receiving crash information**
reset:: **rcv**$\langle crash, i\rangle \rightarrow$
if (*parent* $= i$) **then**
slot, parent, hop:= $\perp, \perp, \perp$;
BCAST$\langle crash, j\rangle$;
fi;

Figure 2: Actions of d-local algorithm that solves stabilising Weak DAS for a general network.

Variables Each process j runs a copy of the program, and the program has two timers: (i) a *wave* timer set to δ, and (ii) a *beacon* timer set to λ, and both δ and λ are parameters of the program. Each process will keep track of its *parent, hop, slot* values, while the variable *slots* stores information about the neighbours of process j. Variable *live* stores the set of processes that respond in a round (i.e., not crashed), while variable $N[j]$ stores the latest known neighborhood of process j.

Actions We now present the actions of the modular algorithm. It is modular in the sense that the crash handling aspect is separate from the data aggregation scheduling algorithm (see Figure 3).

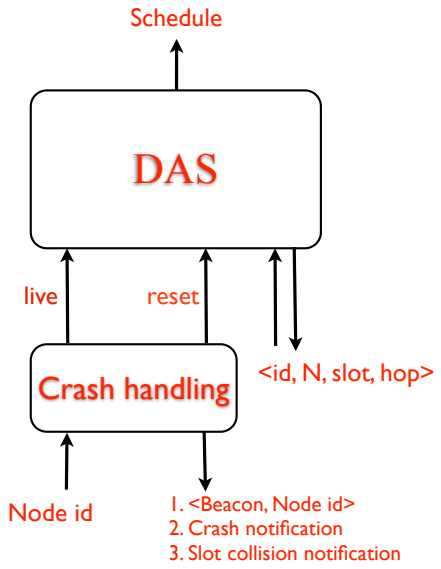


Figure 3: The Modular Fault-tolerant DAS Algorithm of Figure 2

·**Crash handling** Four actions are involved in the crash handling part:

1. *neighbourS*: This action is enabled every λ time units, where nodes exchange "I am alive" message beacons with only their id attached.
2. *neighbourC*: In this action, nodes collect the ids of those neighbours in a given round that send the "I am alive" messages.
3. *check*: In this action, a process j checks whether his elected parent is still alive. If not, node j resets its slot, hop and parent variables, and broadcasts a *crash* message.
4. *reset*: When a node j receives a *crash* message from a node i, it checks if the message is relevant by checking if i is his parent. If it is, then j resets its node (as its slot is no longer weakly proper), and forwards the *crash* message. The *crash* message is forwarded so that every node that is affected by the crash, i.e., those that are affected by the path disappearing, then these nodes can select a new parent.

·**DAS handling** The following actions are those that handle the DAS problem:

1. *dissem*: In this action, the process disseminates slot information. If the process is the sink, then it sends network information, whilst other nodes send updated information. A node sends its id, its neighborhood (minus its parent), its slot and its hop number from the sink, so that the direction of the sink can be determined.

2. *receive*: With this action, nodes collect information about its 2-hop neighbourhood. This is so to avoid nodes in a 2-hop neighbourhood to share the same slot, i.e., to avoid slot collision.
3. *resolve*: In this action, when a set of nodes is notified of a slot collision, the relevant nodes recompute their slot values based on the rank of their respective id in the attached collision set.
4. *check*: In this action, nodes that have their slot/hop/parent values set will check whether their parents are still up. If not, a crash message is broadcast. Next, a process j will check if its slot is colliding with that of other nodes in its 2-hop neighbourhood. If yes, a *collision* message is broadcast. Else, a process j collects values about its 2-hop neighbourhood and sets its slot/hop/parent variables by choosing the smallest hop, and having as a parent a node with the smallest hop. Then the slot is chosen according to the parent chosen.

The program contains two "waves": (i) a slot dissemination wave (DAS handling) and (ii) a crash notification wave (crash handling). Because a slot dissemination wave may be well underway when a crash happens, to minimise the number of nodes that have to reset their slots, we attempt to "slow down" the slot dissemination wave, while "speeding up" the crash notification wave. A similar idea was used in [8] in relation of tracking in WSNs, where erroneous information is overwritten via a "slowing down" process. In [8], the erroneous information is due to transient faults corrupting the state of the program. In this paper, the erroneous information is indirectly created through the node crashes.

The program works as follows: When the slot wave timer times out, each node sends the following information: $\langle j, J, slot, hop\rangle$ (i.e., its id, its neighbourhood, its slot number, its hop number) (action *dissem*). The sink sends a similar message with analogous information (action *dissem*). Node j collects all the messages received (action *receive*). It calculates a slot based on the shortest path to the sink using the hop values collected (action *check*). There can be slot collisions between nodes in the same collision group. However, these are eventually resolved (action *resolve*).

On the other hand, when the *beacon* timer expires (action *NeighbourS*), nodes send "I am alive" messages, and any crash information collected by a node is forwarded to its children (actions *neighbourC*, *check* and *reset*). The children reset their slot (since the path is no longer available), and inform their children about the crash (action *reset*). This process is iterative.

The idea then is to (i) detect crash failures fast enough (i.e., reduce detection latency), and (ii) propagate crash information fast enough so that the slot wave can be caught, thereby reducing the number of nodes requiring a reset. In our program, by setting a low value of λ (fast detection) and by forwarding *crash* notification instantly, and a high value of δ (the slot wave is "slowed down"), one can expect the number of nodes to reset their slots to be reduced.

6.2. Proof of Correctness

We now prove the correctness of the algorithm.

Lemma 1 (Correctness of DAS Handling). *Given a network $G = (V, E)$, where every node has at least $f + 1$ disjoint paths to the sink, where f is a system parameter. Eventually, all nodes $n \in (V \setminus \{S\})$ are assigned a weakly proper and non-colliding slot.*

Proof:

We prove this by induction over the *hop* value.

Init, hop = 1 All nodes start with $hop = \perp$. When *wave* timeouts, the sink disseminates $\langle j, J, slot, hop\rangle$. Its 1-hop neighbours receive the message in action *receive*. In action *check*, all the 1-hop neighbours choose a slot value less than N,

which is trivially weakly proper for G. The slots are also non-colliding for G as all nodes have unique ids, making their rank in the sink's neighbourhood unique. Hence, no two nodes in the 2-hop neighbourhood will have the same slot.

hop = k We assume that all nodes at $1 \leq hop \leq k$ have weakly proper and non-colliding slots for G.

Induction Hypothesis All nodes at $1 \leq hop \leq k+1$ will eventually have weakly proper and non-colliding slots for G.

When the *wave* timer times out, $\forall n \in V \cdot n.hop = k$, n disseminates $\langle n, N \setminus \{parent\}, n.slot, k\rangle$. When a node j in the 1-hop neighbourhood of node n at $hop = k+1$ is not set, then, in action *check*, node j will choose $j.slot = n.slot - rank(j, N) < n.slot$. If there is any slot collision with another node l, i.e., $l.slot = j.slot$, then, in action *resolve*, the value of $j.slot$ or $l.slot$ gets smallerm meaning that $j.slot < n.slot$. Hence, process j has a weakly proper slot for G. If the slot is not a non-colliding one, the nodes sharing the same slot will reduce the value of their slots by two different numbers (action *resolve*), hence j will eventually obtain a non-colliding slot. On the other hand, if $n.slot$ collided with another node, then if n reduces it slot such that its childrens' slots are affected, then the children will choose again to ensure weakly proper property. The weakly proper slot property is maintained during slot collision resolution as the slot values of colliding nodes are reduced. The non-colliding property is eventually guaranteed through the action *resolve*

Since nodes at $hop = 1$ have weakly proper and non-colliding slots and from the induction hypothesis, we conclude that all nodes at $1 \leq hop \leq D_G$ will have weakly proper and non-colliding slots for G. □

Lemma 2 (Correctness of crash handling of d-local algorithm). *Given a network $G = (V, E)$, where every node has at least $f+1$ disjoint paths to the sink, and a fault model F where up to f crash failures can occur. Then, only the nodes that have the crashed nodes on their path to the sink will reset their slot/hop/parent values.*

Proof

When a node n is detected to have crashed by node j such that $j.parent = n$ (through actions *neighbourS, neighbourC, check*), a *crash* message is sent by j. When a child of j receives the notification, it resets its state in action *reset*, and re-broadcasts the *crash* message. Only children of nodes that are resetting that will reset, until a node is reached that has not yet set its slot/parent/hop values. □

Lemma 3 (Crash recovery of d-local algorithm). *Given a network $G = (V, E)$, where every node has at least $f+1$ disjoint paths to the sink, and a fault model F where up to f crash failures can occur. All nodes that have had their slots reset will eventually be assigned new weakly proper and non-colliding slots.*

Proof

After a node resets its state, it continues to collect neighbourhood information (actions *dissem, receive*), as there is at least one path still active. Using actions *check and resolve*, it eventually assigns itself a weakly proper and non-colliding slot.

□

Theorem 5 (Correctness of d-local algorithm). *Given a network $G = (V, E)$, where every node has at least $f+1$ disjoint paths to the sink, and a fault model where up to f crash failures can occur. Then, the program of Figure 2 solves the stabilising weak data aggregation scheduling problem.*

Proof

This follows from Lemmas 1, 2 and 3. □

Now, we present a result that shows that the algorithm of Figure 2 can achieve efficient stabilization. In fact, the algorithm of Figure 2, though d-local, can be made to have a near-constant recovery overhead, under appropriate parameterization.

To show this, we define the notion of *perturbation area* as follows:

Definition 9 (Perturbation area). *Given a network $G = (V, E)$ where every node has at least $f + 1$ disjoint paths to the sink, and a fault model F where up to f crashes can occur. The* perturbation size *is the number of nodes in the F-affected area.*

From Definition 2, an affected area of a graph G is a subgraph of G where all the nodes in the subgraph will have to change their state. In the proposed algorithm, all the affected nodes (those in the affected area) will change their slot values to $\perp$.

Theorem 6 (Bounded perturbation area). *Given a network $G = (V, E)$, where every node has at least $f + 1$ disjoint paths to the sink, and a fault model where up to f crash failures can occur. Then, the program of Figure 2 solves stabilizing weak data aggregation scheduling with a perturbation size of $O(\Delta_G^{\frac{\lambda}{\delta}})$.*

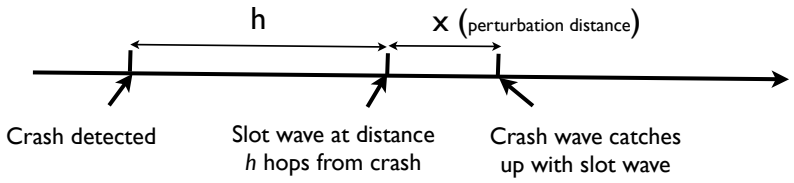


Figure 4: State of protocol when a crash is detected.

Proof

Assume a node n crashes and assume that the slot wave is at a distance h hops away (i.e., the latest node m to choose its slot based on n (n is an ancestor of m) is h hops away from n - see Figure 4). In the time the crash information propagates, the slot information, at *wave* period δ, propagates too over a distance x (which we call perturbation distance).

Let the time it takes for the crash information to catch up with the slot wave be denoted by τ, and let the crash propagation speed be ϵ hops per second.

Then, we have the following:

$$\tau = \lambda + \frac{x + h}{\epsilon} = x\delta$$

This means that in τ time units, the wave will travel $\lceil \frac{\tau}{\delta} \rceil = x$ hops. At the same time, a child of n needs λ time units to detect the crash and then the crash wave will travel $h + x$ hops (to catch up with the slot wave). The time taken will be $\frac{h+x}{\epsilon}$.

Simplifying the equation, we have

$$x = \frac{\epsilon\lambda + h}{\epsilon\delta - 1}$$

Since ϵ is likely to be big, then

$$x \approx \frac{\lambda}{\delta}$$

Since the degree of the graph G is Δ_G, then the perturbation area is $O(\Delta_G^{\frac{\lambda}{\delta}})$. □

Now, given that the objective is to minimise x, if $\delta = \lambda$, then the size of the perturbation area may be big, depending on how ϵ compares with δ. However, by setting δ to be much larger than λ (and allowing ϵ to be large - note that this parameter is outside the control of the protocol and simulation) (i.e., slowing down the "slot" wave), the size of the affected area can be bounded, and approaches $O(1)$ - with the result that the algorithm approaches 2-local stabilizing.

6.3. Performance Analysis and Comparison

In this section, we present a analysis of the performance of our algorithm and we subsequently make a comparison with an algorithm, presented in [28], which is sufficiently close to that presented in the paper.

Performance Analysis: Time and Message Complexities, Energy Efficiency and Latency

The time (in terms of rounds) to execute the algorithm is O(D_G) as all nodes at a given hop count make decisions concurrently and there is a maximum of D_G hops in G. For the communication cost, which is measured in terms of the number of messages, the following is taken into account: (i) number of slot waves, (ii) number of beacons and (iii) number of collision messages. The number of slot waves is $O(\eta_G D_G)$, since nodes at hop = 1 will transmit for D_G rounds. The number of beacons is $O(\Gamma D_G)$, since the number of beacons is a multiple of the number of slot waves. The number of collision messages to be generated has the same complexity, giving the communication cost a complexity of $O(\Gamma D_G)$ (or $O(\Gamma log \Gamma)$).

The latency of our algorithm is $O(D_G(\Delta_G + \alpha_G))$, where α_G is the maximum size of a collision group in the network. The smallest slot for nodes at hop = 1 is bounded by $\Gamma - \Delta_G - \alpha_G$. The smallest slot in the network will then be bounded by $\Gamma - (\Delta_G + \alpha_G)D_G$, giving a latency bound of $(\Delta_G + \alpha_G)D_G$.

Performance Comparison

We compare our work with that presented in [28], which is the closest to the one we present here. Both the time and message complexities of our algorithm is similar to those of [28], though the algorithm in [28] is not fault-tolerant. The algorithm in [28] is later adapted to make it tolerate crashes, however no performance analysis is available for comparison to include beacons and adaptivity messages. Energy-efficiency is closely related to the number of message transmissions and, thus, our algorithm is similar in efficiency to that of [28].

The latency of the algorithm in [28] is $O(24D_G + 6\Delta_G + 16)$, compared to $O(\Delta_G + \alpha_G)D_G$ for that presented here. The algorithm of [28] is based on the specific property of a particular algorithm that generates connected dominating sets, i.e., a specific clustering algorithm.

7. Experiments: Setup and Results

In this section, we first present the simulation setup we use to test the performance of our algorithm, and the results are presented subsequently.

7.1. Simulation Setup

For our simulations, we used JProwler, a Java-based, event-driven simulator for wireless sensor networks [13]. JProwler simulates the transmission/propagation/reception delays of Mica2 motes and the operation of the MAC-layer. We used two network topologies: (i) a grid topology, and (ii) a ring topology. In both cases, a node is equidistant from its neighbours. The signal strength is set such that the degree of a node in the network is between 2 (for rings left and right neighbours) and 4 (for grids, top, bottom, left and right neighbours). We implemented the whole algorithm, and crashed specific nodes during execution.

Network setup Our implementation of the protocol (Figure 2) for handling crash failures when setting up a WSN under JProwler is a per node, message-passing distributed program. Our experimental data consists of running our simulations on (i) $g \times g$ grids, where $g = 11, 15, 21, 25$, and (ii) rings of size $4g - 4$, where $g \in \{11, 15, 21, 25\}$.

When inducing crash failures, we have to choose (i) which node to crash, and (ii) the time at which to crash the node. Note that only one node can be crashed since the grids and rings we generated for our simulations only offer two disjoint paths to the sink.

Choosing Node to Crash In our simulations, we choose the set of all nodes that are within 4 hops of the sink, which we denote by Ω. The nodes in Ω are those that we crash. The reason for choosing these nodes is to allow "maximum" impact on the rest of the network.

Crash Failures In our simulation, we crash one node n from the Ω (i.e., we crash a node $n \in \Omega$) at certain times after the node n has had its values set, and these times are 5, 10, 15, 20 and 25 seconds. Specifically, one crash scenario consists of (i) choosing a crash time $\tau \in \{5, 10, 15, 20, 25s\}$, and (ii) choosing one node $n \in \Omega$ that is within 4 hops from the sink that will be crashed.

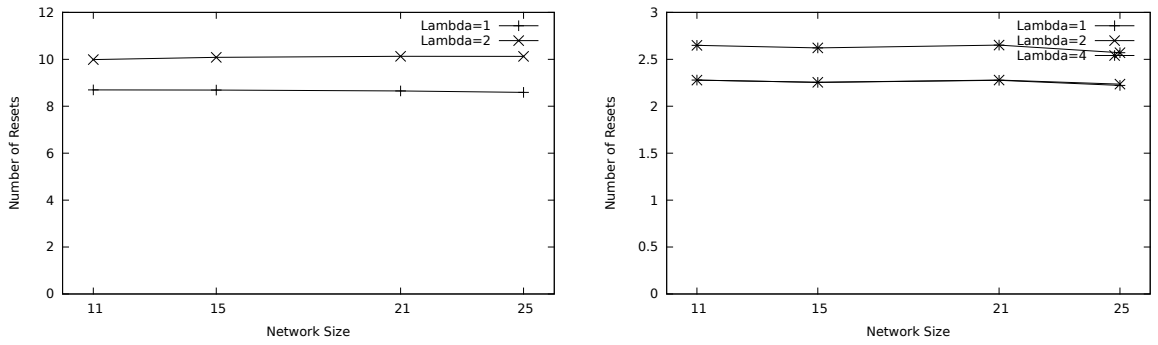
Data Collection and Metrics Each crash scenario (choosing a node to crash and a crash time) is executed 20 times. We then take the average number of nodes reset from all of these runs. These simulations are then repeated for various settings of network size, δ and λ (from the protocol). We analyse the data and focus on the average number of resets (action *reset*) when crash failures occur. This metric captures the locality property of the algorithm.

7.2. Simulation Results

In this section, we present the results detailing the performance of the protocol presented in Figure 2. We focus on the number of resets, which captures the size of the perturbation area (hence the locality property).

7.2.1. Grid Networks: Number of Resets

As can be observed from Figure 5, as δ increases, then the number of resets decreases. Specifically, as $\frac{\lambda}{\delta}$ becomes smaller, the size of the perturbation area decreases. This is due to the fact that the slot wave is "slowed" down a lot, the crash information wave has a greater chance of catching up with the slot wave. Also, a λ increases, detection of crashes takes more time, giving the chance for the slot wave to propagate further, thereby causing an increase in the number of resets. Thus, from our observation, a low value of λ and a high value of δ will cause a very small number of resets. In fact, in Figure 5(d), we observe that the number of resets is 0.



(a) Grid: $\delta = 4$

(b) Grid: $\delta = 8$

(c) Grid: $\delta = 16$

(d) Grid: $\delta = 32$

Figure 5: Grid: Number of Resets vs Network Size

7.2.2. Ring Networks: Number of Resets

As can be observed from Figure 6, as δ increases, then the number of resets decreases. This is again due to the fact that the slot wave is "slowed" down a lot, the crash information wave has a greater chance of catching up the slot wave. Also, a λ increases, detection of crashes takes more time, giving the chance for the slot wave to propagate further, thereby causing an increase in the number of resets. Thus, from our observation, a low value of λ and a high value of δ will cause a very small number of resets.

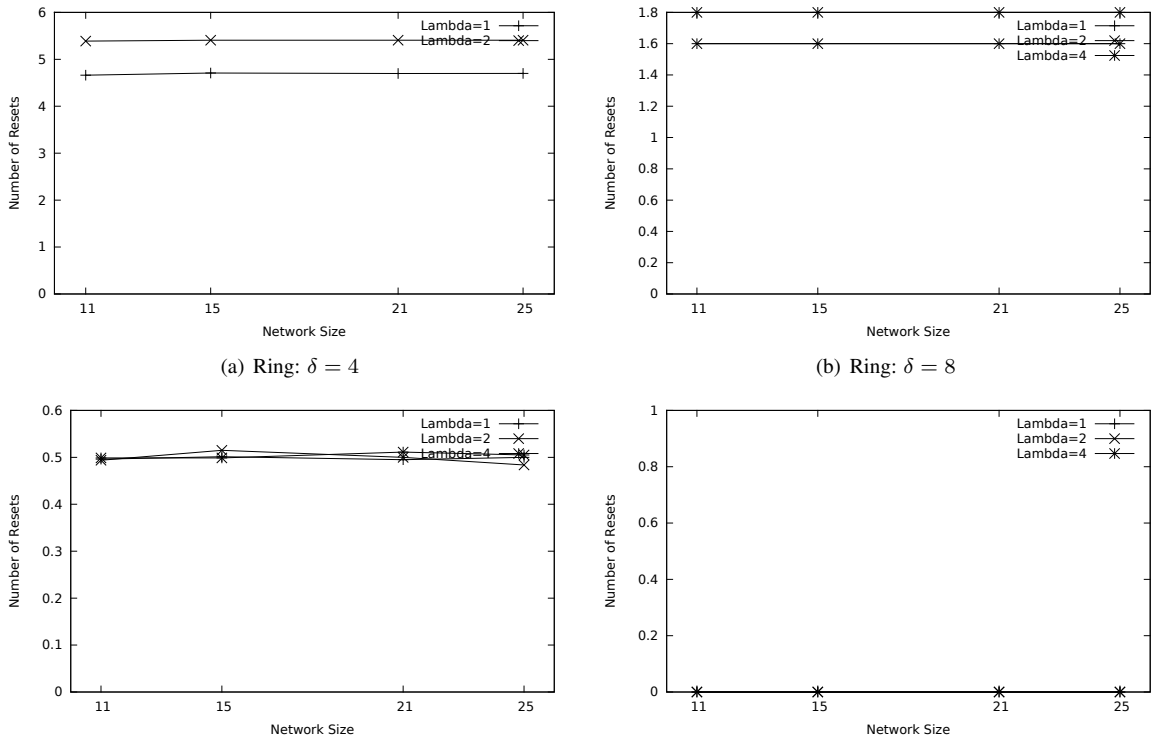
Comparing Figures 5 and 6, it can be observed that the number of resets is greater in grid networks than in ring networks. This is due to the fact that, in grid networks, a node has a higher number of neighbours than in ring networks.

7.2.3. Grid Networks: Completion Latency

For the sake of completeness, we present some results that show the modularity of our proposed algorithm, in that the DAS aspect completes in parallel with the correction aspect (for stabilisation). From Figure 7, we observe that all the graphs are almost identical. Hence, irrespective of the size of the network, the completion time is similar, indicating that the DAS completes irrespective of the time and location of the crash. This captures the modularity of our algorithmm.

7.3. Experimental Results

To assess whether the execution of the protocol is as predicted by our mathematical analysis, we also conducted a deployment on a small testbed of 10 CM5000 sensor motes, which are based on the TelosB platform. We perform an indoor deployment, setting the transmission power of each node to a very low level 2. Our implementation of the protocol requires less than 1KB in memory.



(a) Ring: $\delta = 4$

(b) Ring: $\delta = 8$

(c) Ring: $\delta = 16$

(d) Ring: $\delta = 32$

Figure 6: Ring: Number of Resets vs Network Size

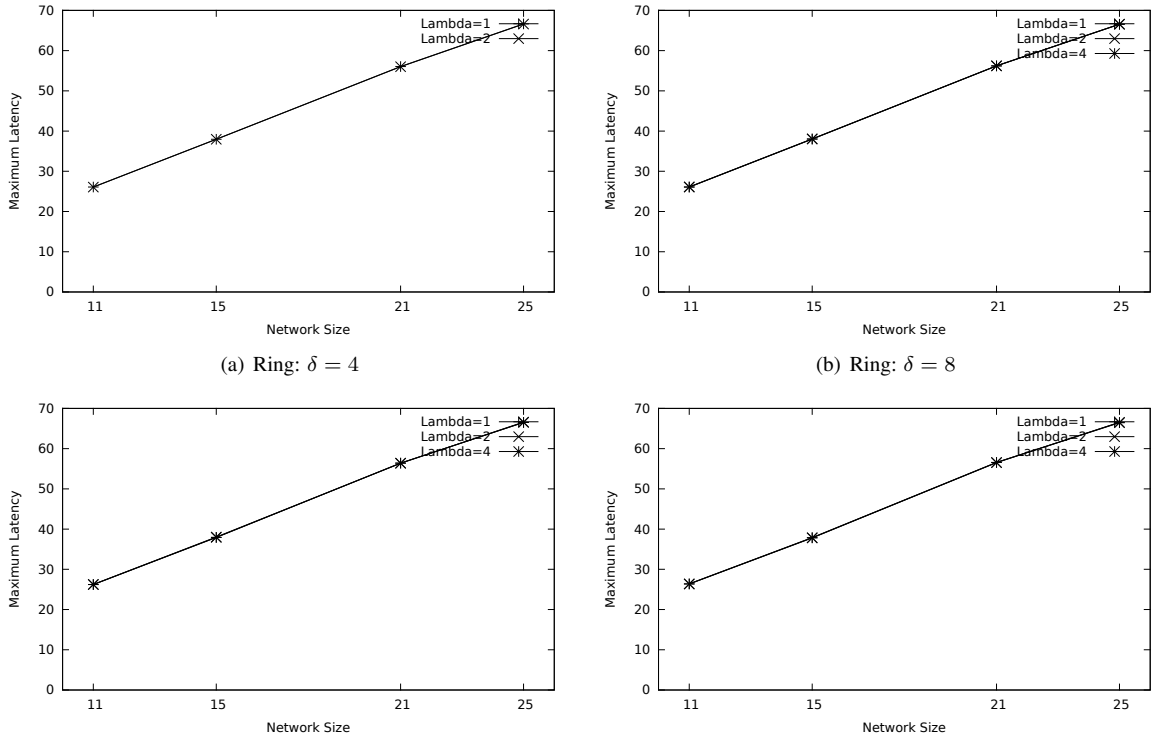
The network deployed is as depicted in Figure 8. As in the simulations, we are interested in the number of resets, where a low number will prove the locality of the protocol. We set the beacon period to $\lambda = \{5s, 7s\}$, and the slot wave period $\delta = \{10s, 20s, 40s, 14s, 28s, 56s\}$, i.e., $\delta = \{2, 4, 8\}\lambda$. We first allowed the system to run for 5 beacon periods to allow nodes to gather their neighbourhoods. In every run, node 1 was crashed a random time after it has broadcast its state. The reason for crashing node 1 is that it allows for maximum impact to be observed and the reason for crashing the node after it broadcasts is to observe how the state of the children are affected. This combination allows for a "worst-case" situation. Each run was repeated 5 times and, for each run, the number of resets is counted, and the average taken. For each value of beacon (and wave period) considered, the number of resets is 2 (hence the average is 2), where nodes 3 and 4 would reset due to their parent, i.e., 1, crashing. This confirms the locality property of the protocol.

8. Related Work

In this section, we survey the area of data aggregation scheduling (e.g., [26]) and convergecast algorithms (e.g., [23]). Since traditional convergecast is very different to data aggregation convergecast, we focus on works about data aggregation scheduling. Most works on DAS or convergecast have focused on optimizing some metric, for example latency or throughput. There exists both centralised and distributed algorithms for DAS.

Centralized algorithms A number of centralised algorithms [26, 24, 7, 22] have been proposed in the literature where the goals was to optimize some objectives, e.g., latency [26], aggregation time [7], concurrency [22]. Further, none of these algorithms are fault-tolerant, in contrast to the work presented here.

Distributed algorithms A number of distributed algorithms for solving DAS [28, 27, 20, 15] have been proposed. How-



(a) Ring: $\delta = 4$

(b) Ring: $\delta = 8$

(c) Ring: $\delta = 16$

(d) Ring: $\delta = 32$

Figure 7: Grid: Completion Time vs Network Size

ever, none of the above work is known to tolerate any types of failures such as crash or transient faults. The approach presented in [28] presents an adaptive strategy for their DAS in the presence of crash failure. To achieve this transformation, they introduce beacons to detect node crashes and nodes execute steps to obtain new parents. The authors of [28] claim fault locality of their algorithm, though there is no proof of correctness of the property. Further, the authors also claim that, in the presence of a number of nodes failing, they re-execute their algorithm again, in contrast to our approach here.

To the best of our knowledge, the only known works that focuses on fault tolerant DAS/convergecast are those by Jhumka [18] and by Arumugam and Kulkarni [19]. The work presented in [18] focused on DAS in the presence of a single crash failure. In this work, we extend the work of [18] by (i) generalizing the result to deal with f faults, (ii) we propose an algorithm that work on general networks, and (iii) we provide simulation results to show the efficiency of the proposed algorithm. In [19], the authors develop a stabilising convergecast algorithm. However, the authors of [19] made assumptions which we do not, such as (i) each node knows their exact location in the network, (ii) the network is a grid, though they provided hints as to how to relax these constraints, and (iii) the fault model is one that corrupts the state of the network, whereas we assume crash failures in this work.

The area of graph colouring is closely related to the area under investigation here, and have been extensively studied before. The problem of graph colouring involves assigning a set of colours to nodes or links under various constraints. For example, the authors of [4] showed how to use 2-distance colouring to obtain a TDMA schedule for a wireless sensor network, where 2-distance colouring is required so that slots are assigned in such a way to avoid collisions between nodes. However, most of the work in graph colouring may not be easily applied to our problem here, since DAS requires an ordering on the colours to be assigned to nodes.

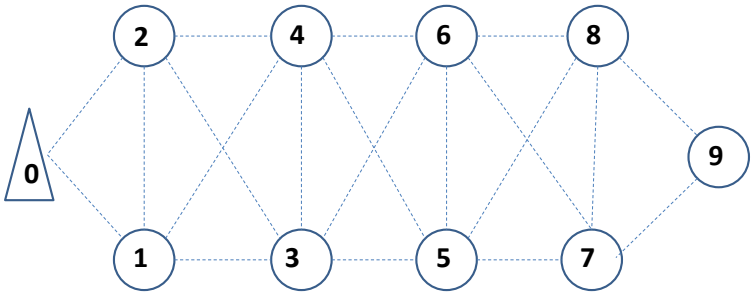


Figure 8: Network used in deployment, with node 0 being the sink.

Finally, since our approach involves developing a tree structure, the seminal work by Madden *et al.*[14] on aggregation service during queries adopted a similar approach. However, the tree structure in [14] is continuously maintained by the root (sink), by periodically broadcasting relevant information to make the structure tolerant to node failures or topology changes. Thus, maintaining the tree structure involves the whole network (i.e., $O(N)$, where N is the network size) and is, thus, not energy-efficient. This maintenance is also performed even though there may not have been any change in the network topology.

In other related work regarding fault tolerance in wireless ad hoc networks, we note the comparison-based techniques developed in [11, 12] for detecting faulty nodes through an assignment of tasks to nodes, though the types of faults assumed is different from this work. The area of self stabilisation [10] has been widely applied to wireless sensor networks to tolerate transient faults, e.g., [17, 3, 8].

9. Discussion

In this section, we will discuss some of the issues raised by our work and the design decisions made in our work.

The first observation is that the nodes periodically broadcast their state information, which is not optimal in terms of the number of messages. We emphasise that the objective of the paper is to conduct a study on DAS in presence of crash failures, and not on protocol optimisations. However, the reason for periodic broadcasts is for the need to detect slot collisions. Since a node can have $O(\Delta_G)$ slot collisions, and with a diameter of D_G, the number of broadcasts can be quite high. Several optimisations are possible to reduce the number of messages transmitted. For example, the protocol can be optimised by having nodes broadcast a special message only when they are resetting their state. This will enable potential new parents to send their state information to those nodes with reset state only during crashes. Another possible optimisation is to have slot information about the 2-hop neighbourhood piggy-backed onto messages, such that whenever a node is choosing a slot, it has all the relevant slot information readily available.

The protocol that has been proposed can be used for DAS purposes (as in this paper). However, since the protocol builds

a tree structure over the physical network to achieve this, the protocol can also be used to setup an aggregation structure, similar to that in [14]. Given that the protocol is crash-tolerant, the overlay will always be maintained in the network, thereby maintaining the efficiency of any WSN application.

10. Conclusions and Future Work

In this paper, we have investigated the problem of data aggregation scheduling in presence of crash failures. The work is novel in its kind, as it is the first work that properly investigates the possibility of designing data aggregation schedules in presence of crash failures. We have made the following important contributions:

- We have identified a necessary condition for solving the data aggregation scheduling problem. This condition provides the theoretical basis that explain the structure of several DAS protocols.
- We have introduced two variants of the DAS problem, namely (i) strong DAS, and (ii) weak DAS.
- We have shown that it is impossible to solve the strong DAS problem.
- We have shown that it is impossible to solve deterministic weak DAS problem in presence of crash failures.
- We introduced the problem of stabilising weak DAS and showed that, in general, there is no 2-local algorithm that solves the problem.
- We developed a modular d-local algorithm that solves the stabilizing weak DAS, where d is the diameter of the perturbation area. We show that, under appropriate parameterization, the size of the area can be reduced to $O(1)$.

As future work, we plan on investigating the problem of DAS in presence of transient faults. We also plan to work on DAS problems where there are multiple sinks in the network.

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21, 1985.

[2] A. Arora and S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.

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

[4] M. Arumugam and S. S. Kulkarni. Self-stabilizing deterministic tdma for sensor networks. *AIAA Journal of Aerospace computing, information and communication (JACIC), 2006*, 2006.

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

[6] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[7] X. Chen and et al. Minimum data aggregation time problem in wireless sensor networks. In *Proc. Mobile Ad Hoc and Sensor Systems (MASS)*, pages 133–142, 2005.

[8] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *Proceedings of 8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004.

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

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

[11] M Elhadef, A Boukerche, and H Elkadiki. Diagnosing mobile ad-hoc networks: two distributed comparison-based self-diagnosis protocols. In *Proceedings of the 4th ACM international workshop on Mobility management and Wireless Access*, 2006.

[12] M. Elhadef, A. Boukerche, and H. Elkadiki. A distributed fault identification protocol for wireless and mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 68(3):321–335, 2008.

[13] G. Simon et al. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. In *Proceedings of IEEE Aerospace Conference*, pages 1339–1346, 2003.

[14] S. Madden et al. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proceedings of 5th Symposium of Operating Systems Design and Implementation*, 2002.

[15] S. Gandhama, Y. Zhangb, and Q. Huang. Distributed time-optimal scheduling for convergecast in wireless sensor networks. *Computer Networks (Elsevier)*, 52(3):610–629, 2008.

[16] A. Jhumka and S. Kulkarni. On the design of mobility-tolerant tdma-based media access (mac) protocol for mobile sensor networks. In *Proc Int Conf on Distributed Computing and Internet Technology (ICDCIT)*, 2007.

[17] A. Jhumka and L. Mottola. On consistent neighbourhood in wireless sensor networks. In *Proceedings Symposium on Reliable Distributed Systems (SRDS)*, pages 199–208, 2009.

[18] Arshad Jhumka. Crash-tolerant collision-free data aggregation scheduling for wireless sensor networks. In *Proceedings of Symposium on Reliable Distributed Systems*, pages 44–53, 2010.

[19] S. Kulkarni and M. Arumugam. *SSTDMA: A self-stabilizing MAC for sensor networks*. Sensor Networks Operations, 2006.

[20] Y. Li and et al. An energy-efficient distributed algorithm for minimum-latency aggregation scheduling in wireless sensor networks. In *Proc. ICDCS*, 2010.

[21] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.

[22] Baljeet Malhotra, Ioanis Nikolaidis, and Mario A. Nascimento. Aggregation convergecast scheduling in wireless sensor networks. *Wireless Networks*, 17:319–335, 2011.

[23] N.Lai and et al. On maximizing the throughput of convergecast in wireless sensor networks. In *Proc. Advances in Grid and Pervasive Computing*, pages 396–408, 2008.

[24] S.C-H.Huang and et al. Nearly constant approximation for data aggregation scheduling in wireless sensor networks. In *Proc. IEEE INFOCOM*, 2007.

[25] B. Sundararaman, U. Buy, and A. Kshemkalyani. Clock synchronization in wireless sensor networks: A survey. *Ad-Hoc Networks*, 3:281–323, 2005.

[26] P.J. Wan and et al. Minimum-latency aggregation scheduling in multihop wireless networks. In *Proc. ACM MobiHoc*, 2009.

[27] X. H. Xu and et al. An improved approximation algorithm for data aggregation in multihop wireless sensor networks. In *Proc. FOWANC*, 2009.

[28] B. Yu, J. Li, and Y. Li. Distributed data aggregation scheduling in wireless sensor networks. In *Proceedings INFOCOM*, 2009.