Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 1539642, 18 pages
https://doi.org/10.1155/2018/1539642

Research Article

Increasing Aggregation Convergecast Data Collection Frequency through Pipelining

Evandro de Souza and Ioanis Nikolaidis

Department of Computing Science, University of Alberta, Edmonton, AB, Canada T6G 2E8

Correspondence should be addressed to Ioanis Nikolaidis; nikolaidis@ualberta.ca

Received 1 March 2018; Accepted 4 July 2018; Published 19 July 2018

Academic Editor: Dongkyun Kim

We consider the problem of increasing the data collection frequency of aggregation convergecast. Previous studies attempt to increase the data collection frequency by shortening the completion of a single data collection cycle. We aim at increasing the frequency at which data collection updates are collected by the use of pipelining and, consequently, increasing the overall data collection frequency and throughput. To achieve this, we overlap the propagation schedule of multiple data snapshots within the same overall schedule cycle, thus increasing parallelism through pipelining. Consequently, the effective data collection time of an individual snapshot may span over multiple, successive, schedule cycles. To this end, we modify the aggregation convergecast model, decoupling schedule length, and data collection delay, by relaxing its precedence constraints. Our solution for this new problem involves the unconventional approach of constructing the schedule before finalizing the exact form of the data aggregation tree, which, in turn, requires that the schedule construction phase guarantees that every node can reach the sink. We compare our results using snapshot pipelining against a previously proposed algorithm that also uses a form of pipelining, as well as against an algorithm that though lacking pipelining, exhibits the ability to produce very short schedules. The results confirm the potential to achieve a substantial throughput increase, at the cost of some increase in latency.

1. Introduction

A commonplace application of Wireless Sensor Networks (WSNs) is the collection of values, e.g., one measurement from each sensor, to a sink node. The underlying protocol is termed *convergecast*. To execute convergecast, it is useful to route packets along paths from sensors to the sink and to enforce a transmission schedule, such that concurrent transmissions do not result in collisions (at the intended receivers).

A particular variety of the data collection problem calls for the extraction of a subset of values to summarize/describe the entire set of measurements across all sensors. This is loosely termed *aggregation*. For a given network, several different spanning trees can be used to route data and perform aggregation on the way to the sink, i.e., different *aggregation trees*. The *aggregation convergecast scheduling* problem is concerned with determining the best aggregation tree in terms of schedule length required to collect a *single* set of data (a "snapshot") from the nodes to the sink. The output of aggregation convergecast scheduling is a schedule of transmissions and a corresponding, spanning, aggregation tree. The produced schedule is a "one-shot" schedule; i.e., it dictates all necessary transmissions to collect a single "snapshot" of data to the sink. It is therefore implied that, for any ordinary application in which we are interested in continuous data acquisition, this schedule is periodically repeated. We can therefore assume that this periodic repetition is taking place back-to-back, and we can think of each execution of the schedule as a "cycle."

To construct the schedule we need to consider the nature of wireless media, which imposes restrictions due to the collisions/interference when transmissions from multiple neighboring nodes towards the same receiver occur concurrently. We will call the restrictions arising out of the impact of interference *resource constraints*, because they enforce access decisions of the shared wireless resource.

Additionally, in aggregation convergecast, leaf nodes transmit their measurements to their parent nodes. Interior nodes of the tree perform the aggregation operation on the

values arriving from their children (and their own value) and transmit the aggregation result to their parent node. The constraints of forcing parent nodes to wait for the results from *all* their designated children to be received, before they produce and transmit the aggregation result, will be called *precedence constraints*.

The solution approach used in the relevant literature is to decompose the problem into two phases: the first one constructs an aggregation tree (i.e., determines the routing), and the second one determines the transmission time of each node (i.e., scheduling). Both phases rely on heuristics. Two important aspects are, sometimes implicitly, assumed: (a) that it is necessary to define an aggregation tree *first* in order to obtain a schedule and (b) that the **strict** enforcement of *precedence constraint* has to be satisfied within the timespan of a schedule cycle. Some solutions obtain the schedule and tree simultaneously but never schedule first and aggregation tree later.

Aggregation convergecast schedule has, thus far, been interested in a single, isolated, collection of all sensor data to the sink, and hence focused on completing the collection in the shortest amount of time. While this approach is desirable for some classes of applications, it does not serve other applications, such as the continuously running data aggregation queries. Continuously running queries can benefit from the ability to collect data samples more *frequently* from the entire sensor field, in order to allow a finer temporal reconstruction of the observed phenomenon. Such applications demand higher sampling rate, i.e., a higher rate of snapshots collected per unit of time and hence higher throughput data collection. In fact, we may even be willing to accept a longer lead-in time for the first snapshot to be collected as long as the snapshot collections can be performed at a higher rate.

To accomplish the stated goal, we will relax the enforcement of the precedence constraints within a single schedule cycle. At the same time, we introduce a form of pipelining, allowing multiple, separate, data snapshots to be independently propagated and gradually, separately, aggregated within the network. The distinct snapshots do not "mix"; i.e., they propagate in FCFS order and are aggregated separately from each other. Since the solution proposed considers satisfying the totality of precedence constraints over a timespan of multiple, consecutive, scheduling cycles, we will call the constituent successive repetitions of the overall transmission schedule as the *microschedules*. A characteristic of a microschedule common with the original definition of aggregation convergecast schedule is that, within a microschedule, each node transmits *exactly once*. The difference is that the order of transmissions scheduled in one microschedule are, typically, inadequate to satisfy all the dependency constraints. Multiple successive microschedules are needed to complete a single snapshot aggregation. The upside is that, within a microschedule, several snapshots may be in the process of being concurrently collected/aggregated. Additionally, as will be later seen, the production of shorter microschedules implies smaller intersnapshot time and, hence, higher overall throughput.

The intuition in support of pipelining being advantageous is based on the spatial reuse of the medium in a multihop wireless network. A simplified example of the benefits of pipelining is shown in Figure 1. The throughput of a solution without using pipelining is 1/7, as seen in Figure 1(b) (one completed snapshot for every 7 slots). The throughput increases to 1/4 in Figure 1(c) using pipelining. In Figure 1(c), node 13 transmits the aggregation (for one particular snapshot), at time T11, while node 12, which feeds 13, transmits after node 13 (at time T12), violating the *precedence constraint*. However, if we consider that the transmissions of nodes 12 and 13 refer to different (in this case consecutive) snapshots, the *precedence constraints* between nodes 12 and 13 **for the same snapshot** are satisfied, albeit over two consecutive (micro)schedule cycles. The tree topology depicted in Figure 1 is, relatively speaking, trivial to pipeline. As we will see in subsequent sections, our snapshot pipelining technique applies to arbitrary connected graphs.

The contributions of our study with respect to aggregation convergecast scheduling are the following:

(i) We extend and propose a new optimization model for aggregation convergecast by the following: incorporating the notion of *snapshots* in the model, relaxing the restriction of single data collection cycle, decoupling throughput from delay, and accounting for the buffering occurring in the network.

(ii) We present a new paradigm for the two-phase approach model in aggregation convergecast, by inverting the phases (scheduling first and routing latter) while adopting a *node activation* scheduling model (compared to the use of *link activation model*).

The current paper is a significantly expanded version based on our earlier work [1]. Specifically, it contributes towards the following directions:

(i) It introduces a formal model of the underlying optimization problem. This model requires the introduction of the new concepts of: *snapshot*, *microschedule*, *aggregation sojourn metric*, and the *snapshot sequence constraint* (Section 3.2);

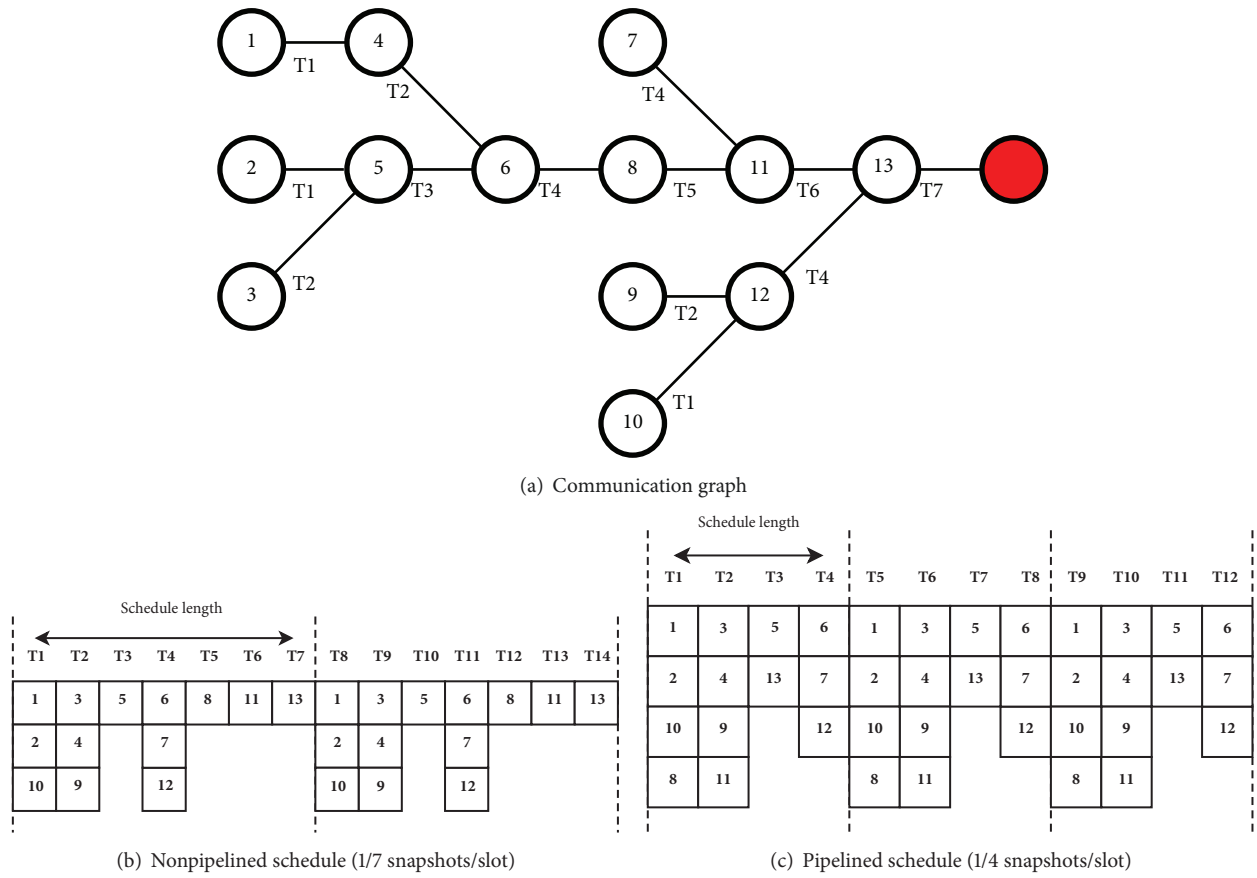
(ii) It provides added explanation through examples of snapshot pipeling (Section 3.3);

(iii) It presents the complexity analysis for the proposed heuristic algorithm (Section 4.1);

(iv) It performs a comparison against the optimal solution for small networks, and the refinement of a corresponding constraint programming model (Section 5.2);

(v) It supplements the performance results with a study of energy consumption characteristics of the solutions produced by the heuristic (Section 5.3).

The remaining of the paper is organized as follows. Section 2 presents related work. Section 3 contains the aggregation convergecast model, definitions, and related concepts. The proposed algorithm is described and discussed in Section 4. Extensive simulations and consequent results and findings are presented in Section 5. Finally, Section 6 concludes the paper.



(a) Communication graph

(b) Nonpipelined schedule (1/7 snapshots/slot)

(c) Pipelined schedule (1/4 snapshots/slot)

FIGURE 1: Throughput improvement using pipelining in an example tree topology. In (a) the edge labels (T*i*) correspond to time slots in (b) and (c). The numbers in (b) and (c) indicate which node transmits. "Stacked" transmissions occur within the same time slot. The microschedule length in (c) is four slots long. Note the increased network utilization, i.e., more concurrent transmissions, in the pipelined schedule compared to the nonpipelined schedule.

2. Related Work

An extensive body of work exists on the aggregation convergecast problem. The problem complexity has been determined by Chen et al. [2, 3] and, recently, revisited by Bramas et al. [4], where it is shown to be NP-hard. Several different heuristics have been devised to address the problem. Some authors adopt a centralized approach [2], while others offer a distributed solution [5]. The interference model varies from one author to the next. The protocol interference model is used in [6], while the physical interference model is adopted in [7]. Most works adopt a single channel convention, but multifrequency solutions have also been investigated [8].

The construction of the aggregation tree is the most studied subject. Shortest Path Tree (SPT) is the preferred choice in [2, 9], while a Connected Dominating Set (CDS) based tree is used in [7]. It has also been suggested that trees satisfying different criteria be combined in a single logical topology. One example is the combination of SPT with a Minimum Interference Tree (MIT), proposed in [10]. Another possible criterion guiding the construction of the aggregation tree is the tree's impact on energy use, as used, e.g., in [11].

We consider three relevant aspects for the purpose of comparison with the current work: the relaxation of precedence constraints; the sequence (order of steps) to obtain a solution (node scheduling first, aggregation tree later); and the use of pipelining. With respect to the first aspect, most works explicitly force parent nodes to only transmit after they have received the transmission from all their designated children [9], while other works use the same restriction implicitly [12, 13]. On the second aspect, to the best of our knowledge, with the exception of our earlier work [1], no previous algorithms decide first about the node's transmission time/schedule and later about the aggregation tree. On the third aspect, we did not locate in the literature a solution proposing pipelining, except in [10], even though their use of pipelining first commits to an aggregation tree and subsequently creates a pipelined schedule over this tree. We did not find a single work that jointly combines the three aspects mentioned.

Even though the work described in [14] bears similarity to our work, it primarily deals with *network capacity*, and the scheduling and routing are obtained after a cell-based network partition is performed, essentially overlaying a grid topology on the communication graph, and subsequently using Compressive Data Gathering (not used in our work). The work of [15] proposes the algorithms LPIPE (line topology) and TPIPE (tree topology) to deal with pipeline

scheduling, but their model does not use any form of aggregation.

Since the closest to our work is [10], in the interest of completeness, we provide a summary of its operation. The aggregation tree constructed is called a Bounded-Degree Minimum-Radius Spanning Tree (BDMRST). The aggregation tree uses bicriteria optimization to combine a Shortest Path Tree and a Minimum Interference Tree. The optimal aggregation tree construction is formulated as follows: given a threshold on the maximum node degree, the objective is to minimize the maximum number of hops (radius) in the tree. Subsequently, the aggregation tree is used as the basis for multichannel scheduling. The WSN deployment area is divided into a set of square grid cells. First, frequencies are assigned to receivers of the tree on each cell, and then a greedy time slot assignment scheme is employed for each cell. The scheduling algorithm does not require the precedence constraint; instead, a pipeline is established for the sink to receive aggregated data from all nodes.

3. Network Model

Let a WSN aggregation convergecast application be defined as an application over a network connectivity graph G. The network connectivity is captured by an undirected connected graph $G = (V, E)$, where V represents the set of vertices/nodes and E represents the set of edges. That is, there are $n = |V|$ nodes. The edges capture the *physical topology* and are assumed to be bidirectional. We assume, for the sake of this work, that all vertices of V contribute to the data collection. The collection occurs along a subset of the edges, E', which defines a subgraph $T = (V, E')$, the *aggregation tree*, and a single node $v_s \in V$ which is the sink that receives the collected and aggregated data. All nodes have precise knowledge about time, such that each node v_i transmits exactly on its allowed time slot, r. No partial transmission or segmentation is allowed. Every transmission from node v_i begins and ends during the time slot r. Each time slot r has a duration of one time unit.

All nodes operate on a single channel. Each node v_i includes a transceiver capable of half-duplex communication. The, so-called "primary" interference represents the inability of a half-duplex transceiver to concurrently receive and transmit. Further, we also assume the *protocol interference model*, as in [10], according to which a potential receiver node can successfully receive a transmission *only if one, and only one*, of its neighboring nodes (as per the connectivity graph) is transmitting. If multiple of its neighboring nodes transmit simultaneously, the result is a collision and none of the concurrent transmissions are received. This is in contrast to the *physical interference model* where the outcome of the simultaneous reception of transmissions from multiple neighbors depends on the relative received signal strengths of the transmissions. The protocol interference model collisions are sometimes called *secondary interference*.

Definition 1. Let a snapshot s^k be defined as the union of the set of values sensed by the network sensors at the same instant in natural time. k captures the snapshot sequence order, and it corresponds to the values of the sensors at a particular time instant, t. A snapshot does not imply any aggregation operator (e.g., MAX, MIN, and AVG). The aggregation operator is defined by the application.

Definition 2. The collection delay Δ is the difference between the time when a snapshot (simultaneous sensor measurements across all nodes) is taken and the time when the sink node has received all the (aggregated) data related to this snapshot.

Definition 3. Precedence constraint is a restriction according to which once a node transmits aggregated data of a particular snapshot, it can no longer be the destination for any transmissions related to the same snapshot.

Definition 4. An aggregation convergecast schedule employs **snapshot pipelining** if the nodes can begin to transmit data of the next snapshot s^{k+1} before the previous snapshot s^k has been completely received by the sink.

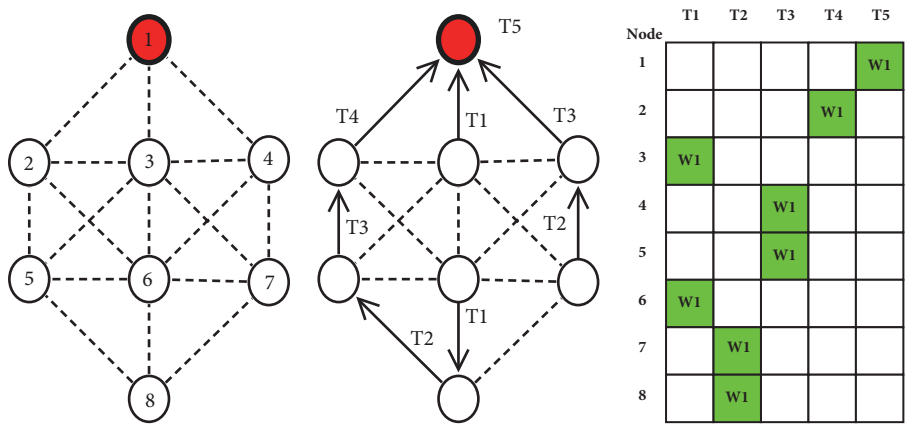
Definition 5. The time difference, as perceived by the sink, between the complete reception of the snapshot s^k and the complete reception of the next snapshot s^{k+1} represents the **intersnapshot** delay δ. It also represents the **microschedule** length.

For the system to be stable in the queuing sense, the rate at which snapshots are created and introduced into the network should match the rate at which they are delivered to the sink. Hence, δ is a property of the particular schedule which, in turn, defines how frequently the snapshots can be generated and delivered, i.e., a metric of throughput. After a possible initial *transient* when the first snapshots start propagating through the network, we reach a steady state, whereby the pipeline is fully utilized and snapshots periodically reach the sink. It is trivial to note (by contradiction) that in steady state the *intersnapshot* delay is equal to the schedule length; i.e., $\delta = l$.

Definition 6. A microschedule is a transmission schedule that describes the overlap, across time, of the concurrently progressing data collection and aggregation of multiple, but separate, data snapshots. The system continuously repeats the microschedule. A microschedule must fulfil the following requirements: (1) each node transmits once and only once, (2) the resulting schedule satisfies the snapshot sequence constraint, and (3) the resulting schedule satisfies the reachability constraint.

A specific description of (2) and (3) is given in Definitions 8 and 9. Intuitively, the *snapshot sequence constraint* is the FCFS property of the snapshots as they propagate through individual nodes and up to the sink, while *reachability constraint* is the property that there exists a path of collision-free transmissions such that from any node we can reach the sink, *even if this traversal would involve multiple successive microschedules to complete*.

Pipelining is achieved by having more than one snapshot propagating through the network during each *microschedule*



(a) Base graph (b) Conventional solution (c) Microschedule

FIGURE 2: Schedule solution for an 8-node graph example without pipelining ($\Delta = \delta = 5$).

period, and hence $\Delta \geq \delta$ $(= l)$, noting that, in previous works, $\Delta = \delta = l$.

Definition 7. The pipeline throughput, C, is the ratio of the amount of data transmissions involved in a single snapshot collection, over the intersnapshot delay, i.e., $C = n/\delta$, where n is the number of nodes. It can be understood as the data collected per unit of time by the sink node.

In the following, the count n includes the sink node, and δ includes a slot in which a pseudotransmission of the sink node to itself is introduced to delineate the end of each snapshot collection epoch. Without loss of generality, one could use $n - 1$ for the metric to count "real transmissions," i.e., transmissions as those in Figure 1. However, the inclusion of sink pseudotransmission makes for a more elegant presentation, and it is adopted from this point onwards. That is, the total number of nodes includes the sink, and a self-transmission from sink to itself is included in the slot immediately following the last reception from its children. This self-transmission, being virtual, is not subjected to any collision/interference constraints.

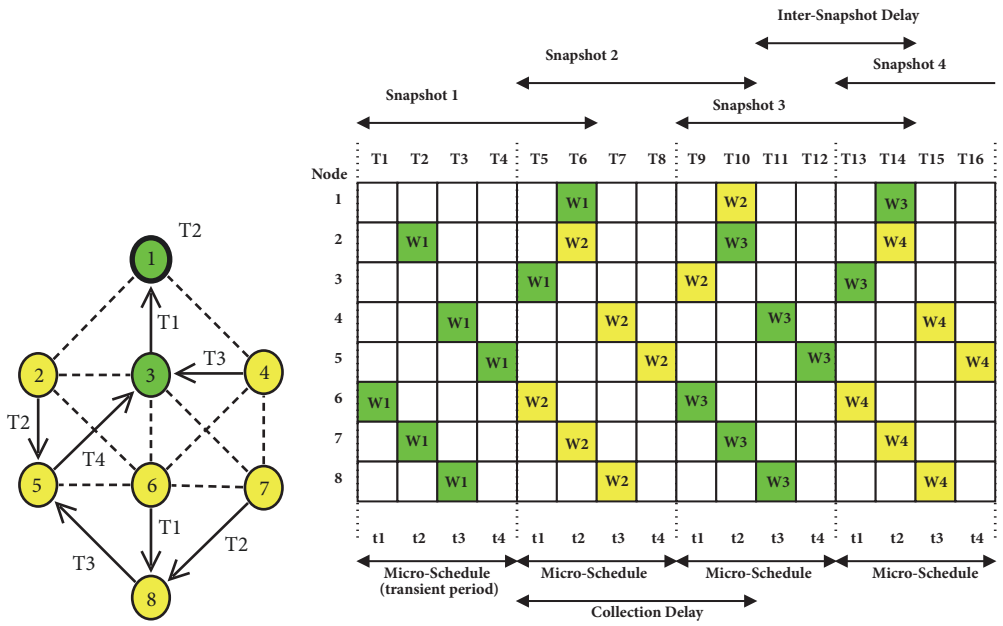
As an example, consider the topology shown in Figure 2(a), where node 1 is the sink node. The optimal solution for the conventional aggregation convergecast problem is shown in Figure 2(b). The aggregation tree is represented by solid arcs between the nodes. The schedule is illustrated by the label next to each arc of the aggregation tree (except for the sink node pseudotransmission at time T5). Essentially T5 indicates the first time slot at which the aggregated data can be made available. The conventional schedule (the same of the microschedule without pipelining) is presented in Figure 2(c). The colors and the Wi notation indicate distinct snapshots ("waves"). Clearly, in Figure 2(c) there is only one snapshot being collected.

Figure 3 presents a solution for the same topology using the snapshot pipelining technique. The aggregation tree and the node transmission time are shown in Figure 3(a). Figure 3(b) provides the detailed view of how the scheduling will unfold. The complete data gathering is executed across two consecutive microschedules. The first (initial) microschedule, on the left side in Figure 3(b), is a transient period. The next microschedule (moving to the right) already reaches steady state. Snapshot 1 (W1) completes, and snapshot 2 (W2) executes its initial part of aggregation. The third repetition of the microschedule is then executed, snapshot 2 (W2) completes, and snapshot 3 (W3) executes its initial part, and so on.

From the sink node perspective, there is a throughput increase. Instead of completing a data collection every $\delta = 5$ time slots (Figure 2(c)), it is now possible to receive a snapshot every $\delta = 4$ time slots (Figure 3(b)).

3.1. A Solution Strategy. Our overall solution strategy is to reduce the schedule length, l, by constructing a *tight* microschedule and subsequently constructing an aggregation tree that *fits* with the schedule. That is, we invert the construction order (tree first, schedule next) followed by existing literature. The complication introduced by our choice is the need to schedule without knowing who should be the recipient of each transmission, i.e., not knowing which node will be the parent of the transmitting node in the aggregation tree. We are therefore forced to use the *node activation model*, according to which, when a node transmits, there should not be transmissions by **any** of its one- and two-hop neighbors. The resulting flexibility to decide the parent of relationship later comes at the cost of reduced throughput [16]. As we will see in the following, any such reduction is apparently compensated by the throughput increase gained by pipelining.

In summary, we define the microschedule $S = \{S(1), S(2), \ldots, S(r), \ldots, S(l)\}$ to be the sequence of concurrent transmissions $S(i)$ taking place in slot i. $S(i)$ is the set of nodes that transmit in slot i, also called the set of *active* nodes in the i–th slot. Assume $v_j \in S(i)$ is a node transmitting in slot i. We denote by a_{v_j} the outgoing arc towards the sink of the transmission of node v_j. It follows that the aggregation tree is defined by $T = \bigcup_{v \in V - \{v_s\}} a_v$. In the conventional aggregation convergecast schemes, a_v was fixed by means of



(a) Solution using pipelining (b) Pipelined schedule

FIGURE 3: Pipelined schedule solution for the graph in Figure 2(a) ($\Delta = 6$; $\delta = 4$).

a precomputed aggregation tree, while, in the proposed snapshot pipelining, a_v is determined by a particular spanning tree construction phase subsequent to the scheduling phase.

An additional complication is that we need to ensure the FCFS delivery of snapshots and that there exists at least one interference-free path (sequence of transmissions) along which sensed data can be forwarded from any node to the sink—even if this path involves a sequence of transmissions over multiple successive microschedules. This is accomplished by enforcing a very mild requirement during the schedule construction, which we call the *reachability constraint*, according to which no interference from scheduled transmissions is possible to disrupt the ability of any node in the network to have at least one path to the sink. Note that this constraint does not imply any optimality with respect to the length of any such paths. However, as the reader may have noticed, long paths are not fundamentally against our objectives because we are ready to trade increased latency for higher throughput.

Definition 8. A schedule S respects the **snapshot sequence constraint** if, for every node v_i, node v_i transmits the next snapshot $(w + 1)$, if the last time it had transmitted, it had transmitted the previous (w) snapshot (per-node FCFS order of snapshots.)

Definition 9. A schedule S respects the **reachability constraint** if there exists at least one directed path $P_{v \rightsquigarrow s}$ from each node v_i to the sink node v_s where each intermediate node in the sequence of the path can receive the transmission without collisions.

The constraints ensure that we avoid schedules where snapshots are incompletely transmitted, or cases where a node that accumulates packets from more than one snapshot transmits them out of order. As we will see in the next section, we also need to capture the accumulation (buffering) in the nodes, which is directly related to how many time slots a child's packet spends between the time it is received at a parent node and the instant the parent sends out the aggregated packet for that snapshot. To this end, we define the following.

Definition 10. The **aggregation sojourn time** is the interval (in slots) starting when an individual packet of snapshot w is received from a child node by its parent node and ends when the aggregated packet for snapshot w is transmitted by the parent.

The definition of aggregation sojourn time extends to the sensor sample taken for a snapshot by a node until the time it transmits it out, either unaggregated (for leaf nodes) or as part of aggregation (for interior tree nodes). The aggregation sojourn time is a proxy for the backlog that builds up at a node as it starts accumulating data. Since many snapshots can be simultaneously collected in the network, the buffer requirements to store them at a node accumulate across all ongoing (not yet transmitted out) snapshots. Even the conventional aggregation convergecast schemes involve accumulation of data prior to transmitting the aggregated output—however, to our knowledge, no previous work models the relevant backlog. The conventional understanding is that only one space (memory buffer) is necessary to perform aggregation from a node's children and that the time it stays in memory does not matter. This is a simplistic assumption since, usually, the instant at which the aggregation (seen as an algorithmic operation) can take place is usually left unspecified. In the worst case, it cannot take place until *all* the constituent to-be-aggregated messages have been received, thus requiring

Table 1: Table of notation.

n	Number of nodes
v_i	Node i
v_s	Sink node
δ	Micro-schedule length in slots
t_{v_i}	Node v_i transmission slot ($\in \{1, \ldots, \delta\}$) within a micro–schedule
$a^t_{v_i,v_j}$	Is arc (v_i,v_j) transmitting at time t? (binary)
w_{v_i}	Node v_i's snapshot index in the micro-schedule relative to other snapshots within the same micro-schedule
b_{v_j}	Node v_j aggregation sojourn time total
Δ	Collection delay in slots
N_{v_i}	1-hop neighbors of node v_i
$r(v_i)$	Recipient of the transmission of node v_i

storage for all of them until the instant of aggregation. We adopt as part of our assumptions that this, worst case, holds. The need to store the messages from multiple snapshots, such that each snapshot is separately aggregated, further increases the memory demands/backlog.

3.2. Formal Model. We formally define the pipelined aggregation convergecast problem using the notation listed in Table 1. Let $G = (V, E)$ be an undirected connected graph as defined in Section 3. The term $a_{v_iv_j}$ represents the existence of the arc $\overrightarrow{v_iv_j}$ if $a_{v_i,v_j} = 1$; otherwise the arc does not exist $a_{v_i,v_j} = 0$. The sink node pseudotransmission takes place at time t_{v_s}, defined as the slot immediately following the reception of the sink's last child transmission.

w_{v_i} is used as a (relative) indicator of which snapshot is transmitted by node v_i when it transmits at time t_{v_i}. The snapshot transmitted by the sink w_{v_s} is the reference relative to which the other w_{v_i} are defined. Trivially, we can set the reference to a fixed value, e.g., $w_{v_s} = 1$ and express all the remaining w_{v_i} relative to that. For example, in Figure 3(b), note that $t_{v_s} = t_1 = 2$ and $t_8 = 3$, and also note that if $w_{v_s} = w_1 = i$ then $w_8 = i + 1$. The additional requirements to setting w_{v_i} for all the transmissions in a microschedule is that the values used are in sequence, and they start from w_{v_s}. Additionally, no transmissions with $w_{v_i} = w_{v_s}$ are allowed from slot t_{v_s} onwards (since the snapshot w_{v_s} has been completed at that point). Through the definition of w_{v_i}, we can indirectly define Δ as

$$\Delta = \left(\max\left(w_{v_i}\right) - w_{v_s}\right) * \delta + t_{v_s} \quad \forall v_i \in V \tag{1}$$

For example, applying the formula on Figure 3(b) gives $\Delta = (2 - 1) * 4 + 2 = 6$. Moreover, through w_{v_i} we can define the total backlog (via Definition 10) due to all snapshots in progress for all packets stored on behalf of those snapshots at node v_i (in units of *time* × *space*):

$$b_{v_j} = \left[\left(\max\left(w_{v_i}\right) - w_{v_j}\right) * \delta + t_{v_j}\right] + \sum_{v_i \in V} a^{t_{v_i}}_{(v_i,v_j)} * \left[\left(w_{v_i} - w_{v_j}\right) * \delta - \left(t_{v_i} - t_{v_j}\right)\right] \quad \forall v_i \in V \tag{2}$$

The first component of b_{v_j} captures the storage needs for the data sourced at node v_i, i.e., its own sample, until the node can transmit the aggregated packet. The second component of b_{v_j} expresses the sum of the sojourns of the *incoming* packets from the node's children (if any).

Combining the above notation, with the constraints disallowing interference/collisions, leads to the following optimization problem formulation:

$$\begin{aligned}
&\textbf{minimize} \quad Q = c_\Delta * \Delta + c_\delta * \delta + c_b * \max\left(b_{v_j}\right) \\
&\textbf{subject to:} \quad \sum_{v_j \in N_{v_i}} \sum_{t=1}^{T} a^t_{(v_i,v_j)} = 1 \quad \forall i \in \{2, \ldots, n\} \\
&\sum_{v_j \in N_{r(v_i)}} a^{t_{v_i}}_{(r(v_i),v_k)} = 0 \quad \forall i \in \{1, \ldots, n\} \\
&\sum_{(v_j \in N_{r(v_i)}, j \neq i)} \sum_{v_k \in N_{v_j}} a^{t_{v_i}}_{(v_j,v_k)} = 0 \quad \forall i \in \{1, \ldots, n\} \\
&\sum_{(v_i,v_j) \in E(S)} \sum_{t=1}^{T} a^t_{(v_i,v_j)} \leq |S| - 1 \\
&\forall S \subset V,\ S \neq V,\ S \neq \emptyset;\ \forall i \in \{1, \ldots, n\} \\
&w_{v_i} = \begin{cases} w_{v_j}; & \text{if } t_{v_i} < t_{v_j},\ a^{t_{v_i}}_{(v_i,v_j)} = 1 \\ w_{v_j} + 1; & \text{if } t_{v_i} > t_{v_j},\ a^{t_{v_i}}_{(v_i,v_j)} = 1 \\ 1; & \text{if } v_i = v_s \end{cases} \\
&a^t_{v_i,v_j} \in \{0, 1\}
\end{aligned} \tag{3}$$

The first three constraints are similar to the conventional aggregation convergecast problem [9]. They capture, respectively, the single transmission per-node constraint, disallowing a recipient node to be transmitting in the same slot it has been assigned to receive (half-duplex) and inhibiting interference at the recipient node when it is supposed to receive. As we relaxed the precedence constraint, a different condition is added to enforce the reachability constraint. The formulation is based in a *Subtour Elimination Formulation* [17] for spanning tree problems. Subtour elimination leans on the observation that a spanning tree (aggregation tree in our problem) has *no simple cycles* and has $(n - 1)$ edges. It can be demonstrated that the selected edges do not contain a cycle if, for any set $S \neq \emptyset$ and $\forall S \subset V$, the number of edges with endpoints in S is less than $(|S| - 1)$. The second constraint necessary for the *Subtour Elimination* is covered by the first constraint. The last constraint links the formulation to the concept of snapshots. It states that the children of a node are (in terms of advancing from snapshot to snapshot) in the same snapshot as their parent, if the transmission time of the child is such that it transmits **before** the parent in the microschedule (as it was in the conventional aggregation convergecast problem), or they are handling the immediately

next snapshot, if the child transmits **after** the parent node v_j in the microschedule. A special case is the sink node, which defines the first (reference) snapshot. This a necessary constraint because it ties to which snapshot each node is currently handling, according to the transmission time of each node relative to its parent. It is not difficult to observe that the conventional aggregation convergecast problem will have only one snapshot, as the children always transmit before their parents. This last restriction imposes the *snapshot sequence constraint* described in Definition 8.

The objective function captures three competing goals of the aggregation convergecast model, using correspondingly three weights $(c_\Delta, c_\delta, c_b)$. The traditional goal is to minimize the delay (Δ in our notation). Using the pipeline we decouple throughput from delay, and throughput is captured by δ which is also the microschedule length. The smaller δ is, the higher the throughput is. The third element (neglected in previous studies) is the cumulative aggregation sojourn time b_{v_j} perceived by each node in the network (and, indirectly, the backlog at each node). It is desirable that this backlog be as small as possible. Here, it is captured by minimizing the largest total aggregation sojourn across all nodes. The reduction of the maximum total aggregation sojourn time has a number of desirable properties: (i) it tends to **decrease the number of node's children** since the larger the number of children, the greater the backlog and, equally, since the node does not need to receive the transmission of many children, its (receive) communication energy consumption will be smaller; and (ii) it tends to result in a **faster data collection** since the smaller the backlog is, the faster the aggregation at a node completes. A consequence of each node having a single parent, to which the result of aggregation is sent, is that the transmission communication energy is constant and independent of the number of children nodes. Note that a side-effect of the optimization is determining the aggregation convergecast tree. Finding an optimized topology was not part of the original aggregation convergecast work by Chen et al. [2, 3] and it subsequently led other authors to create tailor-made topologies to handle secondary optimization criteria, such as energy. Here, we indirectly guide the aggregation tree construction by accounting for its impact on the total aggregation sojourn time. However, this could also decrease the number of concurrently collected snapshots, because the more the snapshots in transit, the larger the backlog at the nodes.

3.3. Examples. We present two examples where the use of snapshot pipelining has an impact on the data gathering performance. The solution shown for both examples is the optimal solution for the specific instance. The first example is a chain topology, as depicted in Figure 4, and the second a tree topology. Node 1 is the sink node in both cases.

The results presented in Tables 2 and 3 reveal the advantage of using snapshot pipelining. Even though the conventional and pipelined approaches present similar results for collection delay and aggregate sojourn, the use of three snapshots allows the snapshot pipelining solution to *"fold"* the schedule and triple the throughput. Figure 5 depicts how the data collection schedule is *"unfolded."* After a transient from T1 to T6, the microschedule is in steady state (from T7 to T9) and repeats (T10 to T12, T13 to T15, etc.). The snapshots forge forward towards the sink node.

Table 2: Aggregation sojourn comparison for chain topology.

Node	1	2	3	4	5	6	7	8	9
Conventional b_{v_j}	10	9	8	7	6	5	4	3	1
Pipelined b_{v_j}	10	9	8	7	6	5	4	3	1

In the case of the binary tree example described in Figure 6, alongside the time slots Ti for the transmissions, we also indicate the aggregation sojourn values Bb_{v_i} of each node. The objective function value can be seen in Table 4. The pipelined solution exhibits the same collection delay as the conventional approach, but its throughput is 44.5% higher.

4. A Pipeline Scheduling Algorithm

The formulated pipelined aggregation convergecast problem includes, as one of its subcases, the original aggregation convergecast problem which is known to be NP-hard. Here, we derive a heuristic (Algorithm 1) whose primary purpose is the increase of throughput. Initially, the input graph must be transformed into a directed graph, with two arcs per edge. Another preliminary action is the removal of all outgoing arcs from the sink, because the sink does not transmit. These actions are carried out by the function *TransfGraph* in Line 1.

Line 2 determines the order in which the vertices will be processed. We consider three heuristics. Each heuristic may employ multiple decision criteria. The first heuristic (*ACSPIPE_1*) applies three criteria in the following order: (a) select the vertex whose outgoing arc has the largest number of common neighbors between itself and the destination vertex, the rationale being to identify vertices inside large cliques on the graph and schedule them earlier, because they could remove more conflicting arcs (explained latter), and break ties, based on (b) giving preference to vertices further away from the sink, the rationale being to remove more arcs far from the sink first and leave the region close to the sink (where all paths must inescapably converge) with more path options, and break remaining ties, (c) selecting first vertices with higher ID. The second heuristic (*ACSPIPE_2*) is a variation of the first one whereby the criteria of distance from the sink are applied first (criterion (b)) and then the criterion of the number of common neighbors (criterion (a)) and then, in the event of a tie, the order is decided based on higher ID (criterion (c)). Finally, the last heuristic (*ACSPIPE_3*) is to randomly select vertices.

Once a vertex order is defined, the algorithm processes nodes following the order selected, trying to assign each vertex to transmit at the earliest possible time slot. This is executed in Lines 3 to 8. A vertex v_i is taken from the queue to be processed. A tentative schedule for node v_i is stored in $sched(v_i)$, representing the slot index within the schedule cycle. The Boolean variable *reachable* is used to indicate if the sink node is still reachable by all vertices on the graph.

Once a tentative schedule (i.e., slot in which to transmit) is selected for vertex v_i, it is time to verify if this selection does not break the reachability restriction. First, the algorithm

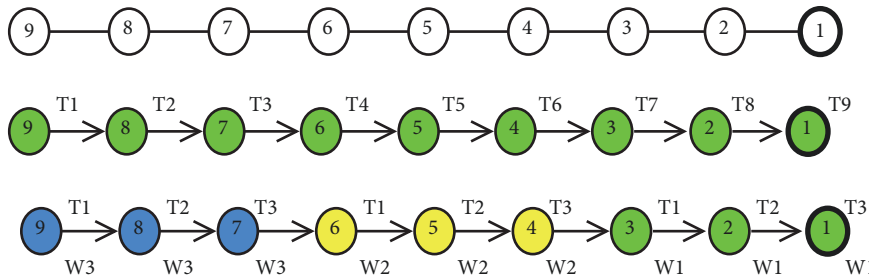


Figure 4: Chain topology: conventional and pipelining solutions.

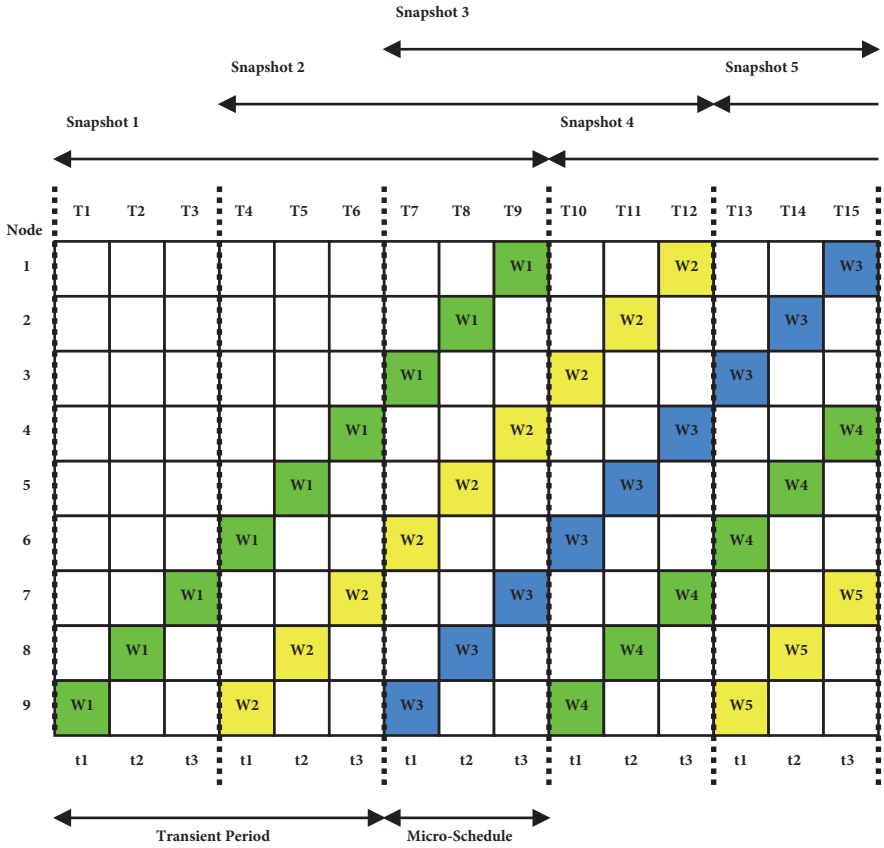


Figure 5: Chain topology: snapshot pipelining schedule.

Table 3: Objective function value for chain topology.

	Snapshots	Δ	δ	$\max\left(b_{v_j}\right)$	Q for $c_\Delta = c_\delta = c_b = 1$
Conventional	1	9	9	10	28
Pipelined	3	9	3	10	22

Table 4: Objective function value for tree topology.

	Snapshots	Δ	δ	$\max\left(b_{v_j}\right)$	Q for $c_\Delta = c_\delta = c_b = 1$
Conventional	1	9	9	12	30
Pipelined	2	9	5	12	26

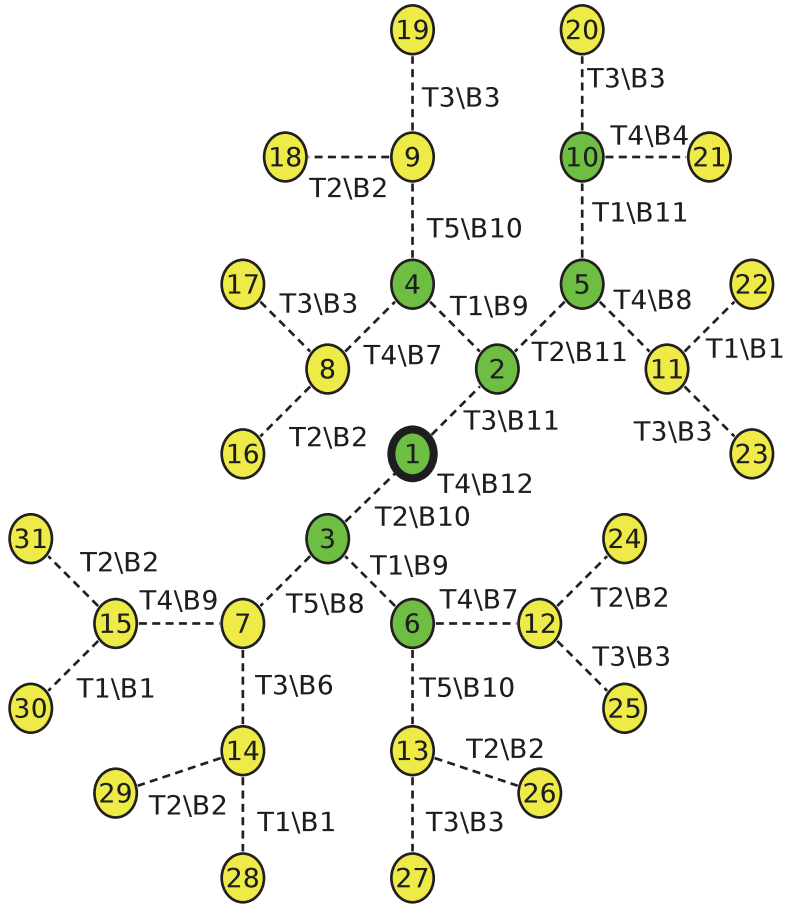


Figure 6: Tree topology: snapshot pipelining solution.

determines the conflicting arcs. Conflicting arcs are those that would not be possible to use because their activation would violate the primary or secondary interference constraints. The conflicting arcs are determined by the function *ConflictArcs* in Line 9. The interference is applied here to restrict the paths available to reach the sink instead of restricting the time slots available for vertex transmission.

Line 10 is used to verify if the removal of the conflicting arcs, identified in Line 9, would break the reachability constraint. Function *Reachability* checks if the sink node is reachable from each v_i's 1-hop and 2-hop neighbors.

Let $\Gamma_1(v_i)$ be the set of v_i's one-hop neighbors and $\Gamma_2(v_i)$ the set of two-hop neighbors. Let the set of nodes in v_i's local region (inclusive of v_i) be defined as $\mathcal{N}(v_i) = \{v_i \cup \Gamma_1(v_i) \cup \Gamma_2(v_i)\}$.

Definition 11. Convergecast Reachability is the property according to which all vertices $u \in V$ in the directed graph $G'(V, A)$ can reach the sink node.

Lemma 12. *A directed graph G', which satisfies the convergecast reachability, maintains this property when vertex v_i is additionally scheduled, if and only if all $w \in \mathcal{N}(v_i)$ can reach the sink after v_i is scheduled. (This can be trivially verified by inspecting the reachability of the nodes in $\mathcal{N}(v_i)$.)*

Proof. A vertex transmission only affects incoming arcs to the vertices in $\{v_i \cup \Gamma_1(v_i)\}$, because these are the vertices affected by the primary and secondary conflicts. An incoming arc to $\Gamma_1(v_i)$ comes from at most vertices in $\Gamma_2(v_i)$. An interruption in a directed path $P_u = \{\overrightarrow{uw}, \overrightarrow{w, z_1}, \overrightarrow{z_1, z_2}, \ldots, \overrightarrow{z_k, sink}\}$ (starting in vertex u) also interrupts all directed paths where P_u is a subset. Only vertices for which all their possible paths to the sink cross vertices in $\mathcal{N}(v_i)$ may have their reachability affected, as they would otherwise have an alternative path that does not involve $\mathcal{N}(v_i)$. Assuming that at least one vertex of its path is in $\mathcal{N}(v_i)$, then checking if all vertices in $\mathcal{N}(v_i)$ maintain their reachability to the sink is sufficient to guarantee that the remaining vertices in V also preserve their reachability to the sink. The contrary is also true, because $\mathcal{N}(v_i) \subset V$. Therefore, a directed graph G' maintains the convergecast reachability property by inspecting only whether the vertices in $\mathcal{N}(v_i)$ maintain their reachability to the sink subject to v_i's schedule. □

Lemma 12 is useful in simplifying the reachability verification. Instead of verifying all network nodes, only $w \in \mathcal{N}(v_i)$ nodes need to be checked. The reachability verification can be performed using Breadth First Search (BFS), based on Lemma 12. Each vertex in $\mathcal{N}(v_i)$ is selected as root. If no

Input: $G(V, E)$, *sink*
Output: T, *sched*, W
1: $G'(V, A) \longleftarrow TransfGraph(G)$
2: $V_o \longleftarrow OrderVertices(G')$
3: **for** ($i = 1$ *to* $|V_o|$) **do**
4: $reachable \longleftarrow$ **false**
5: $sched(v_i) \longleftarrow 0$
6: **while** ($reachable =$ **false**) **do**
7: $reachable \longleftarrow$ **true**
8: $sched(v_i) \longleftarrow sched(v_i) + 1$
9: $A^c \longleftarrow ConflictArcs(G', sched, v_i)$
10: $reachable \longleftarrow Reachability(G', A^c, v_i)$
11: **if** ($reachable =$ **true**) **then**
12: $G' \longleftarrow G'(V, A \setminus A^c)$
13: **end if**
14: **end while**
15: **end for**
16: **for** ($j = 1$ *to* $|A(G')|$) **do**
17: $p[arc_j] \longleftarrow SchedDifference(arc_j)$
18: **end for**
19: $G^* \longleftarrow G'(V, A^T)$
20: $T_{MCA} \longleftarrow MinCostArborescence(G^*, p, sched)$
21: $W \longleftarrow SelectSnapshots(T_{MCA}, sched)$
22: $T \longleftarrow (T_{MCA})^T$

Algorithm 1: Pipelined aggregation convergecast.

reachability violation occurs, the tentative schedule is valid and is committed. If the reachability restriction still holds, all the conflicting arcs A^c, identified before, are removed from graph G'. The removal is executed in Lines 11 to 13. If a reachability violation happens, no arc is removed, and the same vertex v_i is processed again with a new tentative schedule (at the next slot). If the search for a schedule slot for v_i without producing reachability violation turns up fruitless, i.e., all slots thus far defined for the schedule have been exhausted, then the schedule length expands by one slot (by virtue of Line 8) and vertex v_i is trivially scheduled for transmission in that slot.

Once a schedule is defined for all vertices, we need the aggregation tree, preferably one that minimizes the delay. The intermediate result so far is a directed subgraph with the transmission time for each vertex. Each vertex might have more than one path to reach the sink node. After node v_i transmission, the information content is only forwarded further to the sink after node v_i's destination transmission. This slack time between the source vertex u transmission and the destination node v transmission can be used to differentiate between two or more possible candidates, say v' and v'', i.e., whether to use arc $\overrightarrow{uv'}$ or arc $\overrightarrow{uv''}$. Therefore, we calculate the slack times for all nodes in Lines 16-18 and use them as the weight of each arc. If the destination vertex is scheduled after the source vertex, the time difference will be $p[\overrightarrow{uv}] = sched(v) - sched(u)$. If vertex v is scheduled before vertex u, we make use of the fact that the information collection is periodic and that the schedule is periodically repeated. If the destination vertex has already transmitted in the current cycle, the time slack lasts until v's transmission on the *next* cycle. Therefore, the time slack will be $p[\overrightarrow{uv}] = \max(sched) - sched(u) + sched(v) + 1$. Here $\max(sched)$ indicates the length of the microschedule cycle δ.

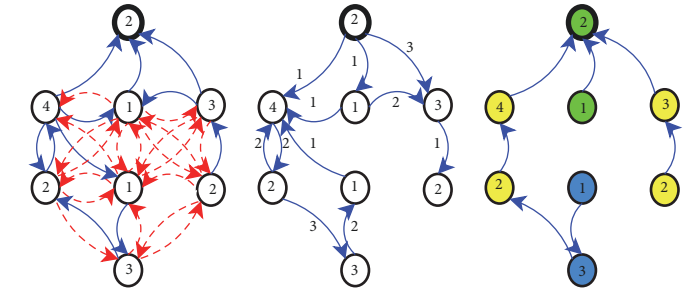
Each arc now has a weight associated with it expressing the time elapsed from the arc's source transmission to the time it is forwarded by the arc's destination (or the delay associated with the "use" of this arc). Our objective now is to find an aggregation tree that minimizes the overall delay. How can such minimum weight tree be obtained? The traditional algorithms (*Kruskal* and *Prim* [18]) to obtain a spanning tree T use undirected graphs. Therefore these algorithms cannot be applied directly. The problem is related to computing a rooted directed spanning tree. The rooted directed spanning tree is a graph which connects, without any cycle, all nodes with $n-1$ arcs (each node except the root) and each node has one and only one incoming arc. This formulation belongs to a class of *branching* problems, also known as *minimum cost arborescence* (MCA) [19]. An algorithm for solving this problem has been proposed by Edmonds [20]. The MCA algorithm is capable of obtaining a result even if the input graph has cycles.

Specifically, convergecast can be seen as an *in-branching* problem [19], and a slight modification on Edmonds' algorithm is enough to obtain an aggregation tree. The modification consists in changing the direction of the arcs obtained after Line 18, using the same weights obtained before. This operation is executed in Line 19. In Line 20, the minimum cost arborescence algorithm is used and an outward tree is obtained.

The snapshot for each node is selected at Line 21. The *SelectSnapshots* function consists in traversing T_{MCA} in *level-order* using BFS and selecting the snapshot each node belongs to based on node's schedule and node's parent schedule. The last step is executed in Line 22 and consists of reverting back the arc's direction. In the end, we have the schedule obtained in Line 15, the snapshots defined in Line 21, and the aggregation tree obtained on Line 22.

An example execution of the algorithm is presented in Figure 7, as it is applied to the input graph of Figure 2(a). The result of the first phase is depicted in Figure 7(a), with the time slot indicated inside the node. The number inside the circles indicates the time slot selected for each node transmission. The dotted arcs indicate the conflicting arcs removed during the schedule selection. Figure 7(b) represents the status after execution of Line 19, where the directed graph is ready to be used by MCA. The number next to each arc indicates the time slack. The final result of the pipeline aggregation convergecast algorithm is depicted on Figure 7(c). The aggregation convergecast execution schedule is depicted in Figure 7(d). In this example, the propagation of a snapshot spans two complete microschedule cycles plus two time slots in a third microschedule; therefore, up to three snapshots are propagating at the same time through the network.

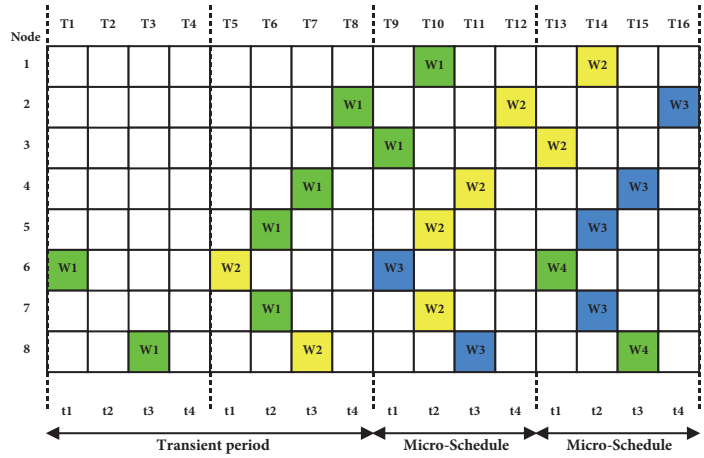
4.1. Complexity Analysis. The run-time complexity of *pipelined aggregation convergecast* is as follows: Line 1 has run-time complexity of $\mathcal{O}(|V|)$. Line 2 may have different run-time complexity depending on the heuristic used. The first heuristic *ACSPIPE_1* is the more demanding, because it has



(a) Graph scheduled. (Line 15)

(b) MCA input. (Line 19)

(c) Final result. (Line 22)



(d) Pipeline execution schedule

FIGURE 7: Execution of Algorithm 1.

to traverse each edge of the graph and get the first and second neighborhood of each of its endpoints in order to define how many common neighbors the endpoints have. Therefore, its run-time complexity is $\mathcal{O}(|E||V|^2)$. The second heuristic *ACSPIPE_2* is executed in $\mathcal{O}(|E|+|V|)$ because a BFS is enough to define which layer each vertex belongs to. The last heuristic *ACSPIPE_3* needs a run-time complexity of no more than $\mathcal{O}(|V|)$ to get a random vertex. The vertex ordering, using weights from the heuristics, can be executed in $\mathcal{O}(|V|^2)$. The execution of *ACSPIPE_1* heuristic represents the worst case. Therefore, the run-time complexity of Line 2 is $\mathcal{O}(|E||V|^2)$.

The scheduling part is executed from Line 3 to Line 15. Line 3 shows that each vertex must be picked to be scheduled. Thus, it is executed $|V|$ times. Line 6 indicates that each vertex may be rescheduled n times, until there is no conflict with the previous scheduled vertices. The estimation of the magnitude of n is not trivial. However, we know that $n \leq |V|$, because $|V|$ is the schedule length's upper bound, or the maximum number of time slots to be tested on each vertex. Line 9 obtains the conflicting arcs surrounding vertex v. It is possible to discover the conflicts inspecting v's first and second neighbors and verifying their scheduled time. Therefore, it requires at most $|V|^2$ steps. The reachability verification, executed in Line 10, can be done by BFS on $\mathcal{O}(|E| + |V|)$. Line 12 is a simple arc removal, which has a run-time complexity of $\mathcal{O}(|E|)$. The remaining lines of the scheduling part are executed with run-time complexity of $\mathcal{O}(1)$. The worst case run-time complexity of the scheduling part is $\mathcal{O}(|V|^4)$.

The algorithm last block defines the routing part. The slack time, calculated in Line 16 and Line 17, has run-time complexity of $\mathcal{O}(|E|)$. Line 19 and Line 22 are simple arc inversions, each one executed in $\mathcal{O}(|E|)$. Line 21 consists in a slightly modified BFS algorithm, which runs on $\mathcal{O}(|E| + |V|)$. The analysis of Line 20 involves the run-time complexity of Edmonds' algorithm, used to obtain the minimum cost arborescence. Tarjan described an implementation of Edmonds' algorithm in [21] that runs in $\mathcal{O}(|E| \log |V|)$. With a simple modification, the algorithm can run in $\mathcal{O}(|V|^2)$, which is more suitable for dense graphs. An implementation error is corrected by Camerini et al. in [22]. Gabow et al. [23] give an $\mathcal{O}(|V| \log |V| + |E|)$ implementation for optimum spanning arborescence. The authors of [23] note that is not possible to improve on the time complexity for any Edmonds' algorithm implementation because the algorithm can also be used to sort n numbers, and sorting n numbers requires $\mathcal{O}(n \log n)$ time. Since it always has to inspect every edge of the graph, we cannot expect to find a better run-time complexity for Edmonds' algorithm than $\mathcal{O}(|V| \log |V| + |E|)$. Therefore, we will assume the run-time complexity for this step to be $\mathcal{O}(|V| \log |V| + |E|)$.

Based on the analysis of the 3 parts of the *pipelined aggregation convergecast* algorithm, setup and vertex order

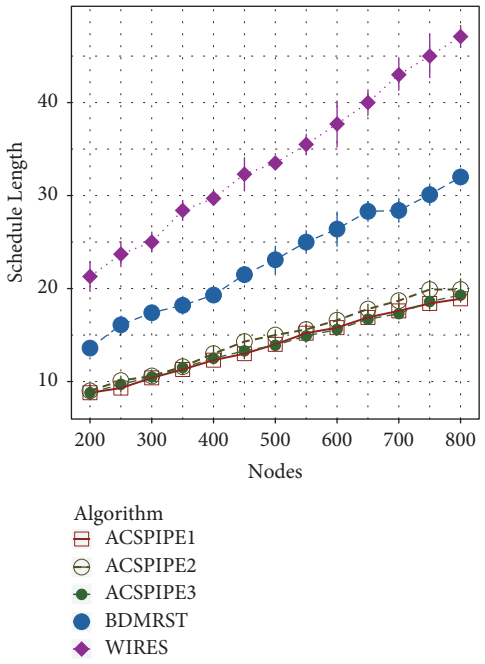


Figure 8: Schedule length.

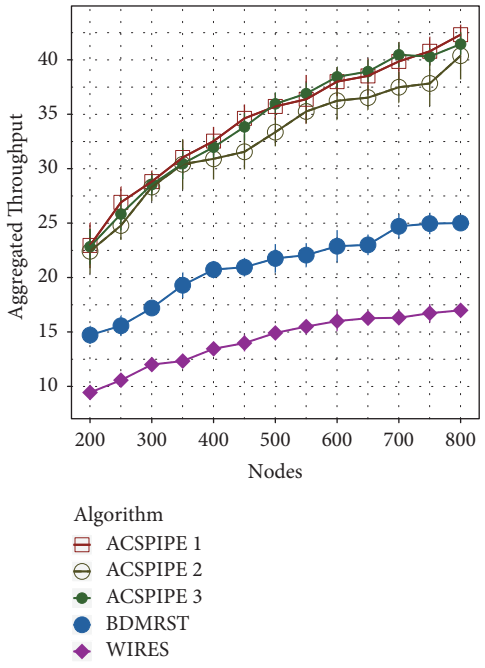


Figure 9: Aggregate throughput.

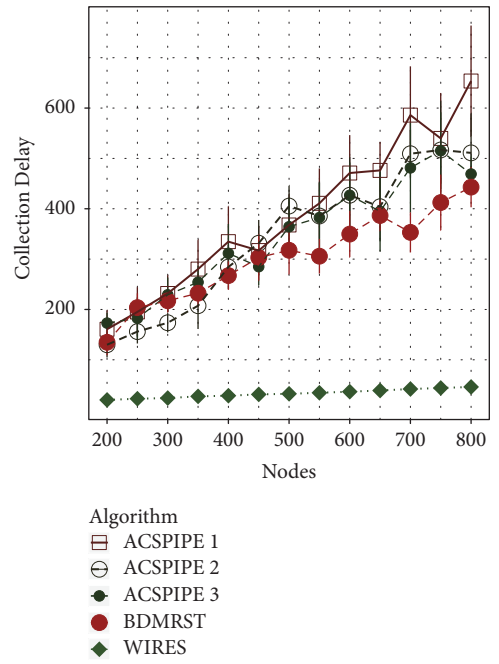


Figure 10: Snapshot collection delay.

definition (Lines 1-2), scheduling part (Lines 3-15), and routing part (Lines 16-22), the run-time complexity of our algorithm is $\mathcal{O}(|V|^4)$.

5. Experiments

We evaluated our algorithm performance by simulating sets of connected graphs generated by placing sensors in a square region of size 200 × 200. The sensor positions are uniformly randomly distributed over the area. The sink node position is also random. Each node has transmission range of $R = 25$, unless otherwise noted. We used sets of nodes ranging from 200 to 800 nodes. Each point in the figures represents the average value for a set of ten graphs with the same number of sensors. The error bars represent 95% confidence intervals. As mentioned in Section 3, the three components of the objective function are given equal weights; i.e., $c_\Delta = c_\delta = c_b = 1$. We implemented the three varieties of vertex selection described in Section 4: *ACSPIPE_1*, *ACSPIPE_2*, and *ACSPIPE_3*. However, judging by the results, the exact choice of *ACSPIPE* variety proved to be of minor significance. We compare *ACSPIPE* against *WIRES* [9] and *BDMRST* [10]. *WIRES* represents a solution using aggregation convergecast without snapshot pipelining, where all the precedence constraints are satisfied within a single schedule cycle, and it is designed using the traditional two-phase approach. *BDMRST* uses pipelining but preselects an aggregation tree with some characteristics. As *BDMRST* was designed as a multichannel scheduler, we restrict the *BDMRST* runs to single channel.

Figure 8 shows the schedule length produced by each algorithm, while Figure 9 presents the aggregate throughput. The collection delay is depicted in Figure 10. The aggregation tree produced by each algorithm is characterized in Figure 11 by means of the maximum node in-degree. Figure 12 captures the tradeoff between throughput and delay for each algorithm. Similarly, Figure 13 shows the relation between the schedule length and the collection delay. The points on both graphs are average values of sets with the same number of nodes.

5.1. Discussion. It has been repeatedly observed that, in general, scheduling data transmissions on wireless networks

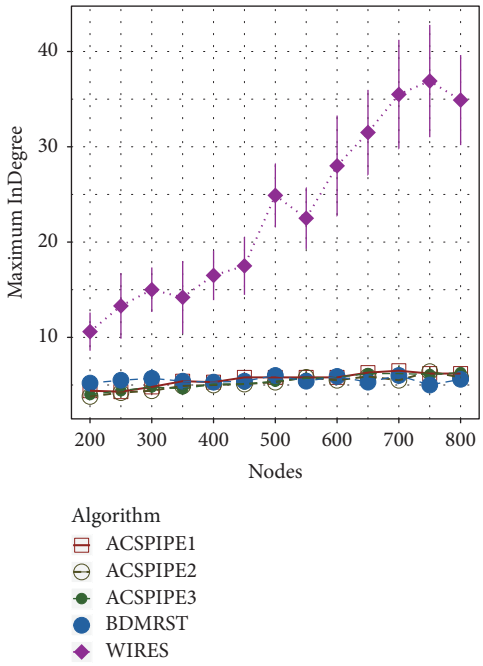


Figure 11: Maximum in-degree.

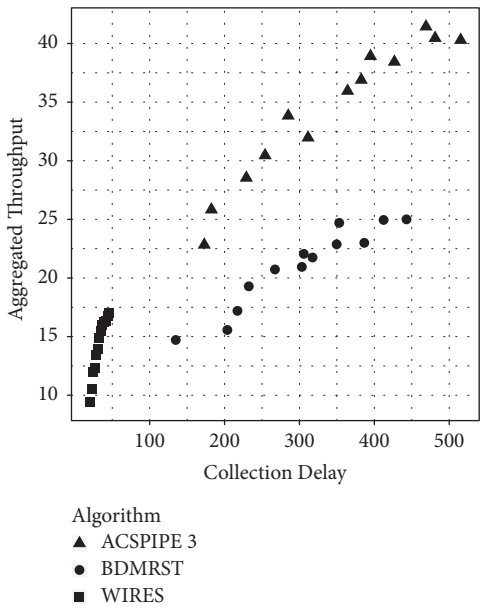


Figure 12: Throughput versus delay.

reveals tradeoffs between throughput and delay [24, 25]. The situation is no different with aggregation convergecast. The impact of trying to minimize latency is clearly seen in Figure 10, where *WIRES* exhibits the lowest data collection latency because of the linkage between delay and throughput (one is the reciprocal of the other). This linkage results in limited aggregate throughput, as shown in Figure 9. The removal of emphasis from latency by means of allowing the precedence constraints to be satisfied over multiple schedule cycles expands the solution space. *ACSPIPE* and

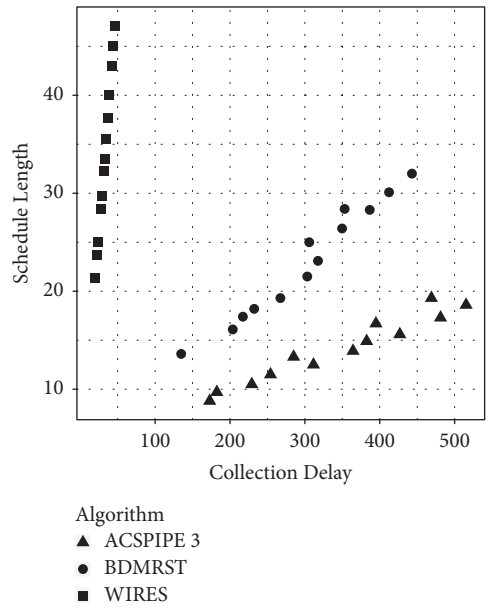


Figure 13: Schedule length versus delay.

BDMRST can explore a larger solution space and new tradeoff possibilities. The aggregate throughput improves, but the collection delay is increased.

Networks with large number of nodes may have an additional limitation to achieve better aggregate throughput. If the transmission range is fixed, the throughput is limited by two factors: (a) for smaller and dense areas, the interference is the dominant factor restricting the schedule length reduction, and, (b) for wider and sparse areas, with uniform node dispersion, the number of hops from each node to the sink is the dominant factor restricting the schedule length reduction. In both cases, the number of time slots necessary tends to be large, and the overall throughput decreases. In both cases, the use of pipelining can be beneficial (despite the precedence constraints) because of the parallelism it brings to data collection.

Apart from precedence constraints, the preselection of an aggregation tree with some predefined characteristics can also limit the performance. *BDMRST*'s bounded-degree and minimum radius restrict the solution space of possible aggregation trees. The effect of bounded-degree is observed on Figure 11. The end result is a solution superior to *WIRES* in terms of aggregate throughput but inferior to *ACSPIPE*. Besides, *BDMRST*'s collection latency is not significantly better than *ACSPIPE*'s (Figure 10). It is fair to say that the intention of *BDMRST*'s authors was to apply it to a multichannel environment, with enough frequencies to eliminate secondary conflicts. Therefore, the only obstacle to improve schedule length is the tree radius (in which *BDMRST* shows better results than *ACSPIPE*). If there are not enough frequencies to eliminate all secondary conflicts, *BDMRST*'s performance is undercut, because it restricts the potential aggregate throughput, without substantial reduction of collection latency.

An interesting behavior to notice among *ACSPIPE* varieties is that the second heuristic (giving priority to vertices

further away from the sink) presents slightly different results. After 350 nodes, the aggregation throughput (Figure 9) is inferior than the that of the remaining two. This fact is also reflected in the tree radius (Figure 16), where the tree depth is smaller. Even with a shorter tree, the collection delay is not much better. The observation suggests that the longer the tree radius, the smaller the pipelining schedule. Another aspect shown by the experiments in Figure 13 is the coupling of throughput and delay ($\Delta = l$) in schemes like *WIRES* creating a linear relationship between the two.

We also observed that *ACSPIPE* presents a natural decrease of aggregate throughput when the transmission radius increases (Figure 14). A larger transmission range increases interference, consequently more time slots are necessary to overcome the contention, and the collection delay will increase as a result (Figure 15).

5.2. Optimal Solution for Small Networks. We compared the results produced by our algorithm against the optimal solution in small networks of ten nodes. The optimal solution of each sample was obtained using a modified version of the *Constraint Satisfaction Model* for aggregation convergecast described in [26]. We modified constraints 5 and 7 of that model (precedence constraint) and introduced a constraint that connects topology, schedule, and snapshots in a single constraint. The new constraint is derived from the formulation described in Section 3.2. It is implemented as a logical constraint of the model. Namely, (4) and (5) depict the new logical constraint to be added to those described in [26]. The results, under $c_\Delta = c_\delta = c_b = 1$, are shown in Table 5. We ran the algorithms on ten sample instances of 10 nodes, using a varying number of links across instances in order to capture the impact of increased density and connectivity.

$$(PP[u] = v) \wedge (SH[u] > SH[v]) \Longrightarrow WV[u] = WV[v] + 1 \quad (4)$$

$$(PP[u] = v) \wedge (SH[u] < SH[v]) \Longrightarrow WV[u] = WV[v] \quad (5)$$

5.3. Energy Consumption. In this section we wish to determine which algorithmic solution produces a logical topology and schedule that reduces the energy use. Even though data aggregation scheme is *per se* an energy saving scheme, the number of transmissions and receptions per node continues to play an important role in determining its energy consumption. That is, we assume that the main source of energy drain is packet transmissions and receptions. The existence of a schedule allows a node to completely turn off its transceiver when not scheduled to transmit or receive. The periodic nature of the schedules allows us to look at the energy behavior of each node, v_i, during a single schedule cycle and extrapolate from this its long-term behavior. In a single aggregation convergecast cycle, a node receives transmissions from q_i children and transmits once their aggregation result. If the transmissions and reception energy are, respectively, E_{tx} and E_{rx}, then, in a single cycle, node v_i expends $q_i * E_{rx} + E_{tx}$ units of energy. If we assume $E_{tx} \approx E_{rx} = E_{op}$ then the energy expended becomes $(q_i+1)E_{op}$. (q_i+1) is the degree of node v_i in the logical topology of the aggregation convergecast tree; i.e., $(q_i + 1) = \deg(v_i)$. The simplification does not hurt generality for comparison purposes across algorithms. One can keep track of $\deg(v_i)$, to account for children (receive cost) separately from the (single) transmission cost. However, this would mean the introduction of one more parameter, the relative transmit versus receive cost, $r_e = E_{tx}/E_{rx}$, which is of little impact given that many transceivers have r_e around 1.0 (typically between 0.8 and 1.5). For example, the Texas Instrument CC2530 has $r_e = 1.35$ (transmit versus

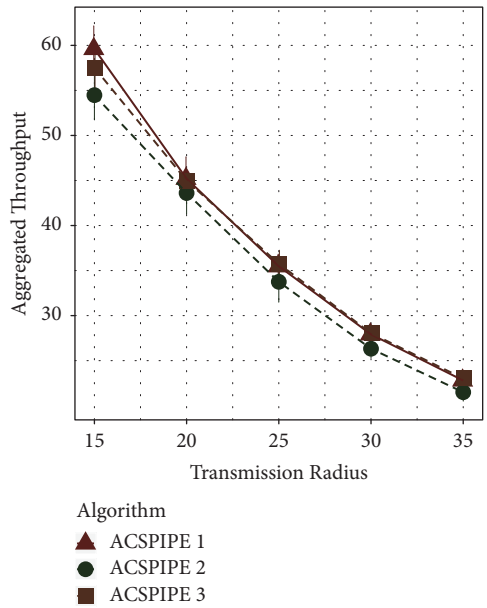


Figure 14: 500 nodes: throughput versus TX range.

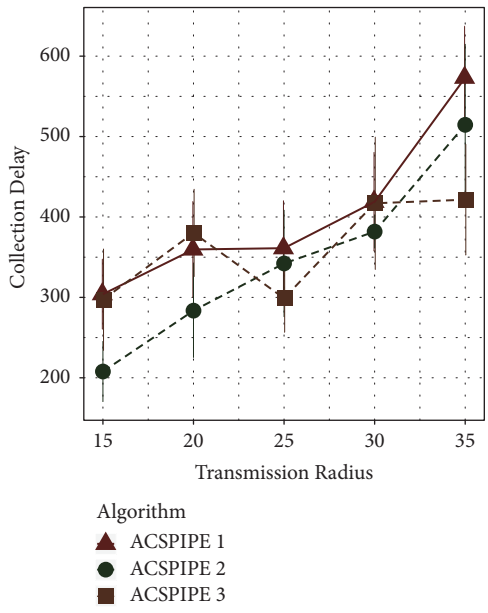


Figure 15: 500 nodes: delay versus TX range.

Table 5: Comparison against optimal solution.

Sample	Links	Optimal			ACSPIPE_1			ACSPIPE_2			ACSPIPE_3		
		δ	Δ	$\max(b_{v_j})$	δ	Δ	$\max(b_{v_j})$	δ	Δ	$\max(b_{v_j})$	δ	Δ	$\max(b_{v_j})$
S0	31	5	7	10	8	8	20	8	8	23	8	8	23
S1	26	5	6	9	7	13	19	7	7	17	7	7	17
S2	24	4	6	9	6	7	19	6	11	19	6	11	18
S3	31	5	7	10	7	7	24	7	13	25	7	13	23
S4	27	6	7	8	6	11	19	7	7	21	6	11	24
S5	35	6	7	8	7	7	20	7	7	20	7	13	22
S6	21	5	6	9	6	16	27	5	21	30	5	9	18
S7	21	4	6	9	7	19	33	7	13	20	7	19	26
S8	16	4	6	9	5	13	28	5	13	22	5	9	16
S9	20	6	7	9	8	8	21	8	8	24	8	8	23
Avg	25.2	5.0	6.5	9.0	6.7	10.9	23.0	6.7	9.8	22.1	6.6	10.8	19.0

receive current of 39.6 mA versus 29.6 mA). If we observe the behavior of the node over a time interval T, and given the schedule length is l, then the energy consumption of each node can be expressed as

$$E(v_i) = E_{op} * \deg(v_i) * \frac{T}{l} \quad (6)$$

For comparison across algorithms, the observation interval is the same, T, and the energy for each operation, E_{op}, is the same; hence we can normalize all the $E(v_i)$ values by $E_{op} * T$, leading to a simplified $E_n(v_i) = \deg(v_i)/l$. Essentially, $\deg(v_i)/l$ expresses the rate of energy depletion at node v_i. As it is common in energy depletion studies, the most stringent criterion is the time until the first node fails. Assuming all nodes initially start with the same energy budget, the first node that fails is the one with the highest depletion rate, i.e., $\max\{E_n(v_i)\}$, over v_i's. On a particular aggregation tree, the highest depletion rate is determined by the node with the highest degree. The interpretation of this model is straightforward: logical topologies with nodes having smaller degrees use fewer energy units per cycle for transmission and reception, because there will be fewer receptions to be executed by the transceiver. However, a topology with low-degree nodes may end up spending more energy than a high-degree topology during the same amount of time T if it is used more frequently (smaller l). As we would have intuitively expected, higher throughput comes at the cost of higher energy cost.

Figure 17 presents the relationship between the two factors that influence node's energy consumption, the maximum node degree $\deg(v_i)$, and the schedule length l. The lines on each algorithm group is a linear regression of the average values, and the shaded area represents the confidence interval of 95%. Figure 18 shows the energy consumption rate for the maximum degree nodes for each algorithm, for different node densities. The number next to each point is the average throughput achieved by the algorithm on the particular node density.

Given the results in the previous section, we use one representative of the ACSPIPE family of algorithms, since the performance differences between them are minor. *BDMRST* exhibits a constant maximum node degree (as its design

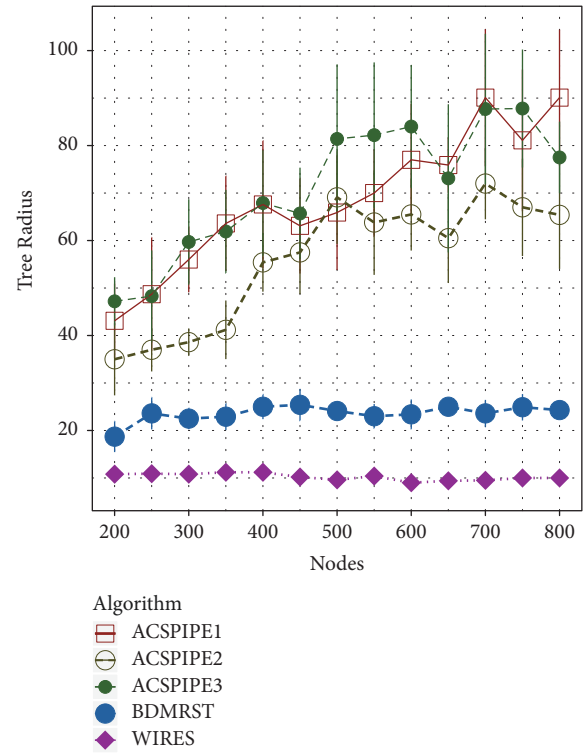


Figure 16: Tree radius.

stipulates) and the lowest energy consumption rate among all protocols. In spite of that, the throughput is lower than *ACSPIPE* as shown earlier. *ACSPIPE* closely follows the topological characteristics of BDMRST, that is, resulting in small maximum degree. This is a welcomed characteristic, but it is not a preference explicitly encoded in the algorithm, as is the case in *BDMRST*. Rather, it emerges as a side-effect of the construction process. The *ACSPIPE* energy consumption rate is higher than *BDMRST* because it is able to achieve higher throughput. Both protocols (*BDMRST* and *ACSPIPE*) decrease their energy depletion rate as the node density increases. *WIRES* demonstrates completely different

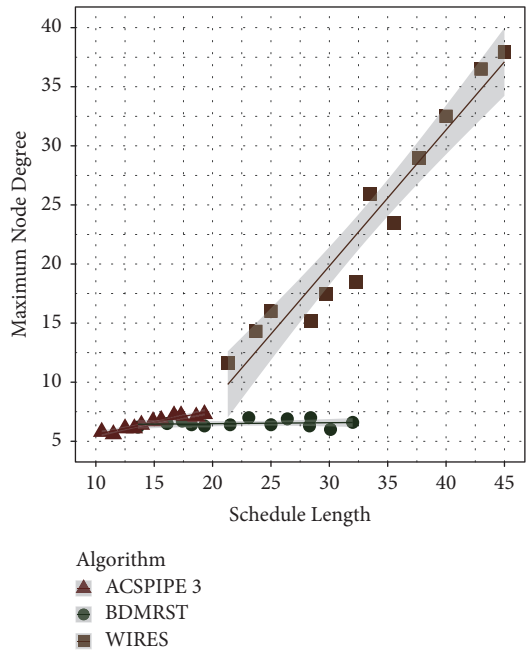


Figure 17: Relationship between schedule and maximum node degree.

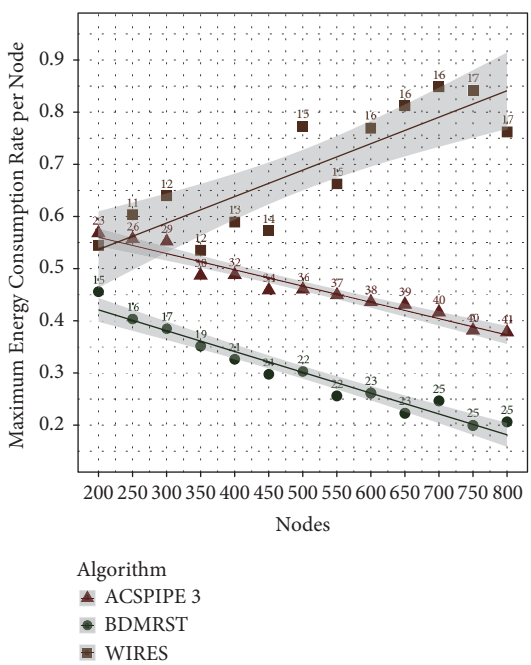


Figure 18: Energy consumption rate according to node density.

dynamics. In *WIRES*, the selection of a logical topology that exclusively addresses the influence of precedence constraints on the solution produces a different relation between node degree and schedule length. The result is seen on Figure 17, which points to an increased energy consumption rate as node density increases. This is because, for *WIRES*, the yielded logical topology (an SPT) results in larger degree in denser graphs. The preference of *WIRES* to minimizing precedence constraints may also lead it to longer schedule lengths [26].

Overall, pipelining impacts the rate of energy consumption of the aggregation convergecast. *ACSPIPE* results in a low maximum node degree and hence low depletion rate. Even though the energy consumption rate might be higher in absolute terms than in *BDMRST*, this is due to the higher throughput achieved, which is only possible through increased node activity per unit of time.

6. Conclusion

We propose a snapshot pipelining scheme applicable to the aggregation convergecast problem. By expanding the satisfaction of precedence constraints from one single cycle to multiple cycles and allowing multiple data collections to proceed concurrently, we are able to attain higher throughput which is necessary for certain classes of sensor network applications. In this paper we address primarily the theoretical and algorithmic facets of such a design. Practical implementation issues are for future work but a first step to evaluating the energy consumption behavior of our scheme has been carried out in Section 5.3.

Our strategy to algorithmically deriving pipelined schedules hinges on a different way to account for interference during the schedule construction phase. Specifically, we account for the possibility that it restricts the paths available from nodes to the sink but without committing to a particular spanning (aggregation) tree. This is in contrast to using interference on a given aggregation tree to limit the time slots when a node is allowed to transmit. Essentially, the only limitation for a node to be allowed to transmit in a time slot is to ensure that it does not preclude the existence of a directed path connecting some other node to the sink. We call this restriction *reachability constraint*. Consequently, we convert the aggregation convergecast formulation, from taking into account *precedence constraint*, to taking into account *reachability constraints*. We also include the notion of concurrently propagating snapshots by labeling each node transmission with the order index of a specific snapshot it forwards. We complete the model by decoupling delay and throughput and including a backlog metric.

A new family of heuristics (*ACSPIPE*) that uses snapshot pipelining is proposed and is based on preserving reachability during schedule construction. We compare our scheme with two other existing algorithms, one from the traditional two-phase (routing first, scheduling second) variety and another that uses pipelining albeit on a predefined aggregation tree. Even though the proposed algorithm is not providing the optimal throughput, our approach is able to present solutions with high throughput improving over the existing literature, but at the cost of latency. Our work is another expression of the well-known tradeoff of throughput versus latency. It has been recognized that, for fixed random networks, higher throughput can only be obtained at the cost of increasing delay [27]. The novelty of our contribution is in demonstrating how to structure the solution space for aggregation convergecast scheduling and how to model the problem, such that snapshots are incorporated, throughput

is decoupled from delay, and buffering occurring in the network is accounted for in the selected topology.

Data Availability

No data were used to support this study.

Disclosure

This paper is a significantly expanded version based on our previous manuscript presented at [1].

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] E. De Souza and I. Nikolaidis, "On the application of pipelining in aggregation convergecast scheduling," in *Proceedings of the 2013 IEEE 14th International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM 2013*, pp. 1–9, June 2013.

[2] X. Chen, X. Hu, and J. Zhu, "Minimum data aggregation time problem in wireless sensor networks," in *Mobile Ad-hoc and Sensor Networks*, vol. 3794 of *Lecture Notes in Computer Science*, pp. 133–142, Springer, Berlin, Germany, 2005.

[3] X. Chen, X. Hu, and J. Zhu, "Data Gathering Schedule for Minimal Aggregation Time in Wireless Sensor Networks," *International Journal of Distributed Sensor Networks*, vol. 5, no. 4, pp. 321–337, 2009.

[4] Q. Bramas and S. Tixeuil, "The complexity of data aggregation in static and dynamic wireless sensor networks," *Information and Computation*, vol. 255, no. part 3, pp. 369–383, 2017.

[5] B. Yu, J. Li, and Y. Li, "Distributed data aggregation scheduling in wireless sensor networks," in *Proceedings of the 28th Conference on Computer Communications (INFOCOM '09)*, pp. 2159–2167, Rio de Janeiro, Brazil, April 2009.

[6] S. C.-H. Huang, P.-J. Wan, C. T. Vu, Y. Li, and F. Yao, "Nearly constant approximation for data aggregation scheduling in wireless sensor networks," in *Proceedings of the 26th IEEE International Conference on Computer Communications (IEEE INFOCOM '07)*, pp. 366–372, IEEE, May 2007.

[7] X.-Y. Li, X. Xu, S. Wang et al., "Efficient data aggregation in multi-hop wireless sensor networks under physical interference model," in *Proceedings of the IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, 2009. MASS '09*, pp. 353–362, October 2009.

[8] Ö. D. Incei and B. Krishnamachari, "Enhancing the data collection rate of tree-based aggregation in wireless sensor networks," in *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '08)*, pp. 569–577, IEEE, San Francisco, Calif, USA, June 2008.

[9] B. Malhotra, I. Nikolaidis, and M. A. Nascimento, "Aggregation convergecast scheduling in wireless sensor networks," *Wireless Networks*, vol. 17, no. 2, pp. 319–335, 2011.

[10] A. Ghosh, Ö. D. Incel, V. S. Anil Kumar, and B. Krishnamachari, "Multichannel scheduling and spanning trees: Throughput-delay tradeoff for fast data collection in sensor networks," *IEEE/ACM Transactions on Networking*, vol. 19, no. 6, pp. 1731–1744, 2011.

[11] S. Upadhyayula and S. K. S. Gupta, "Spanning tree based algorithms for low latency and energy efficient data aggregation enhanced convergecast (DAC) in wireless sensor networks," *Ad Hoc Networks*, vol. 5, no. 5, pp. 626–648, 2007.

[12] M. K. An, N. X. Lam, D. T. Huynh, and T. N. Nguyen, "Minimum latency data aggregation in the physical interference model," *Computer Communications*, vol. 35, no. 18, pp. 2175–2186, 2012.

[13] G. Wang, Q. Hua, and Y. Wang, "Minimum Latency Aggregation Scheduling for Arbitrary Tree Topologies under the SINR Model," in *Ad-hoc, Mobile, and Wireless Networks*, vol. 7363 of *Lecture Notes in Computer Science*, pp. 139–152, Springer Berlin Heidelberg, Berlin, Germany, 2012.

[14] S. Ji, R. Beyah, and Z. Cai, "Snapshot/continuous data collection capacity for large-scale probabilistic wireless sensor networks," in *Proceedings of the IEEE Conference on Computer Communications, INFOCOM 2012*, pp. 1035–1043, March 2012.

[15] H. Choi, J. Wang, and E. A. Hughes, "Scheduling for information gathering on sensor network," *Wireless Networks*, vol. 15, no. 1, pp. 127–140, 2009.

[16] J. Grönkvist, "Assignment methods for spatial reuse tdma," in *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing, MobiHoc 00*, pp. 119–124, IEEE Press, Piscataway, NJ, USA, 2000.

[17] G. Laporte, "Generalized subtour elimination constraints and connectivity constraints," *Journal of the Operational Research Society*, vol. 37, no. 5, pp. 509–514, 1986.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[19] J. Bang-Jensen and G. Z. Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer, London, UK, 2009.

[20] J. Edmonds, "Optimum branchings," *Journal of Research of the National Bureau of Standards*, vol. 71B, pp. 233–240, 1967.

[21] R. E. Tarjan, "Finding optimum branchings," *Networks. An International Journal*, vol. 7, no. 1, pp. 25–35, 1977.

[22] P. M. Camerini, L. Fratta, and F. Maffioli, "A note on finding optimum branchings," *Networks. An International Journal*, vol. 9, no. 4, pp. 309–312, 1979.

[23] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica*, vol. 6, no. 2, pp. 109–122, 1986.

[24] M. J. Neely and E. Modiano, "Capacity and delay trade-offs for ad hoc mobile networks," *IEEE Transactions on Information Theory*, vol. 51, no. 6, pp. 1917–1937, 2005.

[25] S. Toumpis and A. J. Goldsmith, "Large wireless networks under fading, mobility, and delay constraints," in *Proceedings of the INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, March 2004.

[26] E. De Souza and I. Nikolaidis, "Modeling aggregation convergecast scheduling using constraints," in *Proceedings of the 14th ACM International Conference on Modeling, Analysis, and Simulation of Wireless and Mobile Systems, MSWiM'11*, pp. 231–234, New York, NY, USA, November 2011.

[27] A. E. Gamal, J. Mammen, B. Prabhakar, and D. Shah, "Throughput-delay trade-off in wireless networks," in *Proceedings of the INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 1, March 2004.