Dynamic Conflict-Free Transmission Scheduling for Sensor Network Queries

Octav Chipara, Chenyang Lu, *Member*, *IEEE*,
John A. Stankovic, *Fellow*, *IEEE*, and Catalin-Gruia Roman, *Member*, *IEEE*

Abstract—With the emergence of high data rate sensor network applications, there is an increasing demand for high-performance query services. To meet this challenge, we propose Dynamic Conflict-free Query Scheduling (DCQS), a novel scheduling technique for queries in wireless sensor networks. In contrast to earlier TDMA protocols designed for general-purpose workloads, DCQS is specifically designed for query services in wireless sensor networks. DCQS has several unique features. First, it optimizes the query performance through conflict-free transmission scheduling based on the temporal properties of queries in wireless sensor networks. Second, it can adapt to workload changes without explicitly reconstructing the transmission schedule. Furthermore, DCQS also provides predictable performance in terms of the maximum achievable query rate. We provide an analytical capacity bound for DCQS that enables DCQS to handle overload through rate control. NS2 simulations demonstrate that DCQS significantly outperforms a representative TDMA protocol (DRAND) and 802.11b in terms of query latency and throughput.

Index Terms—Query scheduling, TDMA, sensor networks.

1 Introduction

Early research on wireless sensor networks (WSNs) has focused on low data rate applications, such as habitat monitoring [1]. In contrast, recent years have seen the emergence of high data rate applications, such as real-time structural health monitoring [2] and preventive equipment maintenance [3]. For instance, a structural health monitoring system may need to sample the acceleration of each sensor at rates as high as 500 Hz, resulting in high network load when a large number of sensors are deployed for fine-grained monitoring. Moreover, the system may have highly variable workload in response to environmental changes. For example, an earthquake may trigger a large number of new queries in order to assess any potential damage to the structure. Therefore, a key challenge is to provide a high-throughput query service that can collect data from large networks and adapt to workload changes.

To meet this challenge, we propose *Dynamic Conflict-free Query Scheduling* (DCQS), an integrated framework for transmission scheduling designed to meet the communication needs of high data rate applications. A data collection application may express its collection interests as queries over subsets of nodes which may involve data aggregation [4]. Instances of these queries are executed *periodically* to collect data at the base station. The use of routing trees in executing query instances introduces *precedence constraints* among packet transmissions. For example, when data aggregation is used, a node must wait for its children's data reports before computing an aggregated data report and relaying it to its parent. Intuitively, integrating application layer information (the periodicity of queries) and routing layer information (the precedence constraints) into the transmission scheduling process may lead to significant performance improvements. By incorporating this cross-layer information into the scheduling process, DCQS provides not only better performance than traditional transmission scheduling techniques designed for general workloads and networks, but also has the following salient features: 1) DCQS can adapt its transmission schedule in response to the addition/removal of queries and changes in query rates without having to recompute its transmission schedule. 2) DCQS dynamically determines the transmissions to be executed in each slot and, as a result, it may adapt to workload changes more effectively than traditional TDMA protocols with fixed transmissions schedules. 3) DCQS has low runtime overhead and limited memory requirements making it suitable for resource-constrained devices.

The remainder of the paper is organized as follows: Section 2 describes the query and network models we adopt. Section 3 details the design and analysis of DCQS. Section 4 describes how DCQS handles dynamic networks and workloads. Section 5 provides simulation results using NS2. DCQS is compared with existing transmission scheduling approaches in Section 6. Section 7 concludes the paper.

2 System Models

In the following, we describe the query and networks models that DCQS builds upon.

- *O. Chipara is with the Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, Mail Code 0404, La Jolla, CA 92093-0404. E-mail: ochipara@gmail.com.*
- *C. Lu and C.-G. Roman are with the Department of Computer Science and Engineering, Washington University in St. Louis, Bryan Hall, Campus Box 1045, One Brookings Drive, St. Louis, MO 63130. E-mail: lu@cse.wustl.edu, roman@wustl.edu.*
- *J.A. Stankovic is with the Department of Computer Science, School of Engineering and Applied Science, University of Virginia, 151 Engineer's Way, PO Box 400740, Charlottesville, VA 22904-4740. E-mail: stankovic@cs.virginia.edu.*

Manuscript received 1 Oct. 2008; revised 8 Oct. 2009; accepted 12 July 2010; published online 25 Oct. 2010.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-2008-10-0391. Digital Object Identifier no. 10.1109/TMC.2010.209.

2.1 Query Model

DCQS assumes a common query model in which source nodes produce data reports periodically. This model fits many applications that gather data from the environment at user-specified rates. Such applications generally rely on existing query services [5]. A query is characterized by the following parameters: a set of sources that respond to a query, the query period P_q, and the start time of the query ϕ_q, and an optional function for in-network aggregation [4]. *Query instances* are released periodically to gather data from the WSN. We use $I_{q,k}$ to denote the kth instance of query q. The query instance $I_{q,k}$ is released at time $R_{q,k} = \phi_q + k \cdot P_q$ which we call the release time of $I_{q,k}$. For each query instance, a node i needs $W_q[i]$ slots to transmit its (aggregated) data report to its parent. DCQS can support queries with in-network data aggregation, such as average and histogram [4], as well as more common forms of aggregation such as packet merging [6] and data compression [7]. While DCQS can optimize the performance of queries with aggregation, it also supports queries that do not perform aggregation.

A query service works as follows: a user issues a query to a sensor network through a base station, which disseminates the query description to all nodes. To facilitate data aggregation, each nonleaf node waits to receive the data reports from its children, produces a new data report by aggregating its data with the children's data reports, and then sends it to its parent. We assume that there is a single routing tree that spans all nodes and it is used to execute all query instances. This assumption is consistent with the approach adopted by existing query services [4]. During the lifetime of the application, the user may issue new queries, delete queries, or change the period of existing queries. DCQS is designed to support such workload dynamics efficiently.

2.2 Network Model

DCQS works by scheduling *conflict-free* transmissions in a time slot. To determine whether two transmissions are in conflict, we introduce the Interference-Communication (IC) graph. The IC graph, $\mathrm{IC}(E, V)$, has all nodes as vertices and has two types of directed edges: *communication* and *interference* edges. A communication edge $\overrightarrow{ab}$ indicates that the packets transmitted by a may be received by b. A subset of the communication edges forms the routing tree that is used for data collection. An interference edge $\overrightarrow{ab}$ indicates that a's transmission interferes with any transmission intended for b even though a's transmission may not be correctly received by b. An example of an IC graph is shown in Fig. 2.

The IC graph is used to determine if two transmissions may be scheduled concurrently. We say that two transmissions, $\overrightarrow{ab}$ and $\overrightarrow{cd}$ are *conflict-free* ($\overrightarrow{ab} \parallel \overrightarrow{cd}$) and may be scheduled concurrently if 1) a, b, c, and d are distinct and 2) $\overrightarrow{ad}$ and $\overrightarrow{cb}$ are not edges in E. Referring to Fig. 2, the transmissions $\overrightarrow{ea}$ and $\overrightarrow{fb}$ conflict due to the interference edge $\overrightarrow{eb}$. In contrast, transmissions $\overrightarrow{ne}$ and $\overrightarrow{po}$ are conflict-free, since edges $\overrightarrow{no}$ and $\overrightarrow{pe}$ are not part of the graph.

RID, a realistic method for constructing IC graphs based on Receive Signal Strength (RSS) measurements, is proposed in [8]. To gather RSS measurements, nodes transmit sequences of two packets. The first packet is broadcast at maximum power and is used to identify the sender and prepare the potential interfering nodes to measure the RSS during the subsequent packet transmission. The second packet is transmitted at the default power. Based on the collected RSS values, interference edges are added to the IC graph as follows: Consider a node p which receives packets from one of its children c. Node p knows c's RSS as well as the RSS of all other senders which may interfere with c's transmission. RID generates all sets of interferes $I(p)$ such that $|I(p)| \leq N_r$, where N_r is a bound on the number of senders that may be active in a time slot. Given the transmission $\overrightarrow{cp}$, RID computes Signal to Noise Plus Interference Ratio (SNIR) for each $I(p)$. If the SNIR for the set $I(p)$ is below a threshold, then incoming edges from the nodes in $I(p)$ to p are added to the graph. RID's communication cost is linear in the number of nodes.

The IC graph is based on the SNIR model. Empirical studies validating the accuracy of the SNIR model on 802.15.4 [9], [10], [8] and 802.11 [11] radios have already been performed. Moreover, MAC protocols which take advantage of the SNIR model have already been proposed and their performance validated empirically [12]. These previous studies on real hardware indicate that the IC graph is a realistic assumption. The IC graph was studied in [9], which presented a realistic approach for constructing IC graphs based on the SNIR model and RSS measurements. It should also be noted that the IC graph model adopted by our algorithm is more realistic than models often adopted by earlier scheduling algorithms (e.g., unit disk models and ignore interference).

Interference is inherently probabilistic and time-variant. It is important to note that the SNIR threshold controls the temporal stability of the IC graph. A conservative SNIR threshold would lead to a more stable IC graph at the cost of reduced throughput. We recognize that even when using conservative SNIR threshold, packets may be still corrupted as a result of intermittent interference. We address these issues through retransmissions and multipath routing (see Section 4).

While the IC graph is built conservatively to improve temporal stability, over time the interference relations may change significantly. We may detect changes in IC graph by monitoring the reliability of data collection over time. If the reliability falls below a user-set threshold then the IC graph is rebuilt. The IC graph also needs to be updated when nodes are added or removed.

3 Protocol Design

DCQS separates the problem of conflict-free query scheduling in two parts. First, we consider the problem of scheduling each query instance in isolation when all network resources are dedicated to its execution. To this end, DCQS constructs *plans* for executing each query instance. A plan is a sequence of *steps*, each comprised of a set of *conflict-free* packet transmissions. DCQS executes a plan sequentially by performing the transmissions assigned to each step. Upon the completion of a plan execution, the data reports from all sources involved in the query would have been delivered to the base station.

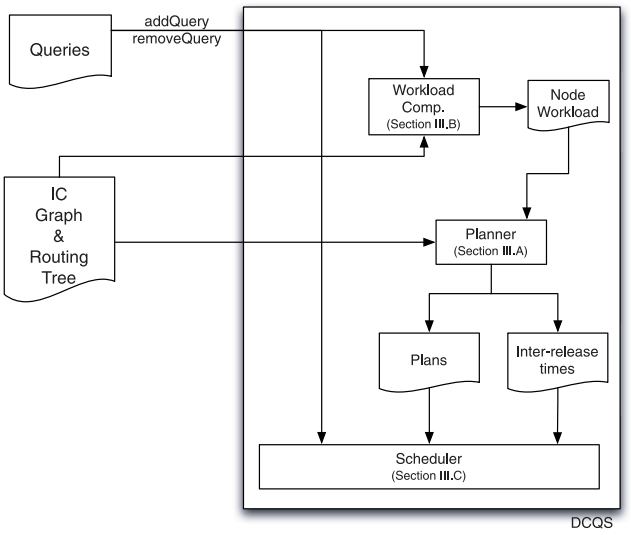


Fig. 1. DCQS uses cross-layer information in making scheduling decisions. It has two key components: a planner and a scheduler.

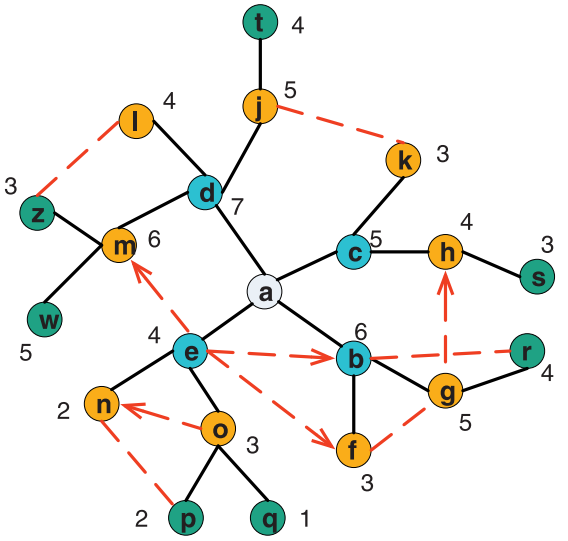


Fig. 2. IC graph: The solid and dotted lines represent communication and interference edges. The edges without arrows are bidirectional. The shown numbers are the steps in which each node transmits under a plan for an instance with a workload demand of one slot per node.

Next, we consider the problem of executing a set of queries submitted by the user. DCQS could accomplish this by executing instances one at a time as they are released according to their constructed plans. If this were the case, a *single* query instance would be executed at any time even though some instances may be executed concurrently due to spatial reuse. To improve throughput, DCQS dynamically determines which steps in the plans of the released instances may be executed without conflict and executes them concurrently in a *slot*. A *slot* is a period of time sufficient for transmitting a packet and acknowledging it. Note that, unlike traditional TDMA protocols, DCQS does not maintain an explicit schedule but rather determines the schedule at runtime based on the temporal properties of queries and their plans.

This approach has several intrinsic advantages:

1. DCQS separates the costly process of constructing plans from the dynamic transmission scheduling performed in each slot.
2. To reduce the overhead, DCQS reuses previously constructed plans for queries whenever possible. We will show that many queries may be executed according to the same plan.
3. The DCQS schedule executes query instances based on their temporal properties. Thus, DCQS can handle changes in query rates and the addition/removal of queries efficiently.
4. Rate control may be performed to prevent overload.

To facilitate efficient query scheduling, DCQS shares information across the traditional protocol stack boundaries (see Fig. 1). DCQS has two main components: a planner and a scheduler. The planner is responsible for constructing plans. The planner uses the IC graph and the following query information exposed by the application: the set of sources and the number of packets each node involved in a query has to transmit. The scheduler runs on every node and makes scheduling decisions at runtime based on the start time and period of queries as exposed by the application and the plans constructed by the planner.

DCQS works as follows:

1. When a new query is submitted, DCQS identifies a plan for its execution. As discussed in Section 3.1, it is often the case that many queries can be executed using the same plan. When no plan may be reused, the planner constructs a plan for executing the query.
2. Next, the base station performs rate control to ensure that the total query rate remain within the maximum query rate under DCQS. If necessary, the rates of the queries are decreased proportionally to not exceed the maximum query rate.
3. The phase, period, and aggregation function of the query are disseminated to all nodes.
4. At runtime, the scheduler executes all query instances.

We will start by presenting a centralized version of the planner. Next, we describe the local scheduler. The section concludes with the description of the distributed planner.

3.1 The Centralized Planner

In this section, we will present the centralized planner.

Definitions. A *plan* is an ordered sequence of *steps* that executes a *query instance*. Fig. 2 shows an example of a plan consisting of steps 1-7. By executing this plan, an instance of a query that has all nodes as sources and workload demand of one slot per node would deliver all data from sources to the base station.

A plan has the following properties: 1) In each step, conflict-free transmissions are assigned. 2) When the query involves aggregation, the plan must respect the precedence constraints introduced by aggregation: a node is assigned to transmit in a later step than any of its children. Note that DCQS does not impose any constraint on the order in which a node's children transmit. 3) Each node is assigned in sufficient steps to meet its workload demand. We use $T_q[s]$ to denote the set of transmissions assigned to step s in the plan of query q and L_q to denote the length of q's plan.

Reversed Plan	a	d	b	c	e	m	j	g	h	o	Actual Plan
1	d										7
2	b	m									6
3	c	j	g			w					5
4	e	l		h			t	r			4
5			f	k	o	z			s		3
6					n					p	2
7										q	1

Fig. 3. Constructed plan for IC graph in Fig. 2 with a workload demand of one slot per node. The first and last columns are the step indices in the reverse and actual plans. The top row denotes the intended receivers. The other entries denote the senders in each step.

An example of a plan with seven steps is shown in Fig. 3. In each step, multiple conflict-free transmissions are assigned. For example, nodes n and p may transmit in step 2 since their transmissions do not conflict ($\overrightarrow{ne} \parallel \overrightarrow{po}$). The precedence constraints introduced by aggregation are also respected: nodes p and q transmit in earlier steps than their parent o.

We opt for a node to wait for data from its children even for queries that do not involve aggregation because this results in transmission schedules that have *long contiguous* periods of activity/inactivity: the node transitions from a sleep state to the active state just-in-time to receive the data from its children and transitions back to sleep after it completes collecting data from its children and relaying it to its parent. Such schedules are efficient because they reduce the wasted energy in transitions between sleep and active states.

Since a node waits to receive the data reports from its children (to support data aggregation and improve energy efficiency), the query latency may be reduced by assigning the transmissions of a node with a larger depth in the routing tree to an earlier step of the plan. This reduces query latency because it reduces the time a node waits to receive the data reports from all of its children.

The pseudocode of the centralized planner is shown in Fig. 4. The centralized planner works in two stages. In the first stage, the planner constructs a *reversed* plan (V_q) in which a node's transmission is assigned to an *earlier* step than its children. In the second stage, it constructs the actual plan (T_q) by reversing the order of the steps to enforce the precedence constraints. We will be using the notation $V_q[i]$ and $T_q[i]$ to refer to the set of transmissions assigned in ith step of the reverse and actual plans, respectively. The planner maintains two sets of nodes during the construction of V_q: *completed* and *eligible*. A node n is a member of the *completed* set if the planner has already assigned n to transmit in sufficient steps such that its workload demand is met. The *eligible* set contains nodes whose parents are in the *completed* set. Initially, the *completed* set contains the root of the routing tree and the *eligible* set contains its children. The planner considers the eligible nodes in order of their *priority* and assigns steps in which they transmit to their parents. The priority of a node depends on its depth, number of children, and ID. Nodes with smaller depth have a higher priority. Among the nodes with the same depth, the ones with more children have higher priority. Node IDs are used to break ties. After the planner assigns steps for n to transmit to its parent, it moves n from the *eligible* to the *completed* set, and adds n's children to the *eligible* set. The

centralized-planner:
1: $completed = \{root\}$; $eligible = children(root)$;
2: while ($completed \neq V_q$)
3: Let n be the highest-priority node in $eligible$
4: invoke **assign-steps**(n)
5: $completed = completed \cup \{n\}$
6: $eligible = eligible \cup$ children(n)
7: reverse plan: $T_q[s] = V_q[L_q - s]$
assign-steps(n):
9: Let p be n's parent and assigned.
10: Let $V_q[s]$ be the last step in which a transmission $\overrightarrow{np}$ is assigned
11: $n.minStep = s + 1$; $n.assignedSteps = 0$
12: while ($n.assignedSteps < W_q[n]$)
13: if $\overrightarrow{np}$ does not conflict with any transmission $\overrightarrow{ab} \in V_q[n.minStep]$
14: $V_q[n.minStep] = V_q[n.minStep] \cup \{\overrightarrow{np}\}$
15: $n.assignedSteps = n.assignedSteps + 1$
16: else$n.minStep = n.minStep + 1$

Fig. 4. The centralized planner.

first stage is completed when the *completed* set contains all the nodes. In the second stage, the planner reverses the order of the steps in the reversed plan.

Let us consider how the scheduler assigns n's transmissions to its parent p in the reversed plan. The planner associates with each node two pieces of information: $n.minStep$ and $n.assignedSteps$. The value of $n.minStep$ is the step number in which the planner attempts to assign n's transmission to p, while the value of $n.assignedSteps$ is the number of steps in which n is assigned to transmit. Since nodes with smaller depth have a higher priority, p's transmissions to its parent has already been assigned to enough steps. Let s be the last step in the reversed plan V_q in which p transmits to its parent. In the reversed plan, the earliest step in which n may transmit its own data report to p is $n.minStep = s + 1$. This means that, in the actual plan, p must transmit its data report to its parent at least one step before the parent transmits its data report such that the precedence constraints introduced by data aggregation are respected. To determine if the transmission $\overrightarrow{np}$ may be assigned to $V_q[n.minStep]$ without conflict, n must verify that all transmission pairs that involve $\overrightarrow{np}$ and any transmission already assigned to $V_q[n.minStep]$ are conflict-free. The planner assigns node n to transmit in multiple steps until its workload demand $W_q[n]$ is met.

Fig. 2 shows an example topology and the plan constructed by the centralized planner. All nodes have a workload demand of one slot. The constructed plan is shown in Fig. 3. Initially, the children of the root a are eligible. The planner starts by scheduling d since it has the highest priority among the eligible nodes. The planner assigns $\overrightarrow{da}$ to step 1 since $V_q[1] = \emptyset$. Next, b becomes the highest priority eligible node. The first step in which $\overrightarrow{ba}$ may be assigned is step 1. However, since $\overrightarrow{ba} \not\parallel \overrightarrow{da}$, $\overrightarrow{ba}$ cannot be assigned to that step. We assign $\overrightarrow{ba}$ to step 2 since $V_q[2] = \emptyset$. Similarly, $\overrightarrow{ca}$ and $\overrightarrow{ea}$ are assigned to steps 3 and 4, respectively. When the planner completes assigning e's transmission to its parent ($\overrightarrow{ea}$), m becomes the highest priority eligible node. Since $\overrightarrow{da}$ is assigned to step 1, the first step to which $\overrightarrow{md}$ may be assigned is 2. Since in $V_q[2]$ only $\overrightarrow{ba}$ is assigned and $\overrightarrow{md} \parallel \overrightarrow{ba}$, we assign $\overrightarrow{md}$ to step 2. A more interesting case occurs when f becomes the highest priority eligible node. The earliest step to which

$\overrightarrow{fb}$ may be assigned is 3, since the transmission of its parent's transmission $\overrightarrow{ba}$ is assigned to step 2. The planner first attempts to assign $\overrightarrow{fb}$ to steps 3 and 4, but fails. $\overrightarrow{fb}$ cannot be assigned to step 3 due to $\overrightarrow{gb}$. $\overrightarrow{fb}$ cannot be assigned to step 4 because $\overrightarrow{ea} \not\parallel \overrightarrow{fb}$ due to $\overrightarrow{eb}$. Since no transmission is currently assigned to $V_q[5]$, $\overrightarrow{fb}$ is assigned to it. The planner continues to produce the reverse plan shown in the table. Next, the planner reverses the order in which the steps are executed. Accordingly, the last step in the reversed plan ($V_q[7]$) is the first step in the plan ($T_q[1]$), the second to last step in the reversed plan ($V_q[6]$) is the second step in the plan ($T_q[2]$), and so on.

It is worth noting that a plan may have uneven workload demands across nodes. For example, when data are collected from nodes without data aggregation, the nodes closer to the root may require multiple slots to transmit the data of their descendants. In this case, the planner would assign these nodes to transmit in multiple steps as to meet their workload demand.

3.2 Plan Sharing

The plan of a query q depends on the IC graph, the set of sources, and the aggregation function. Queries instances executed at different times may need different plans if the IC graph changes due to dynamic channel conditions. As discussed in Section 4.3, DCQS can construct plans that are robust against certain changes in the IC graph. This allows instances executed at different times to be executed according to the same plan. Moreover, note that queries with the same aggregation function and sources but with different period or start time can be executed according to the same plan. DCQS amortizes the overhead of computing query plans by executing multiple queries according to the same plan. This is often possible since queries with different temporal properties may be executed according to the same plan. We say that two queries belong to the same *query class* if they may be executed according to the same plan.

Even queries with different aggregation functions may be executed according to the same plan. Let $W_q[i]$ be an *upper bound* on the number of steps node i needs to transmit the aggregated data report to its parent for an instance of query q. If the planner constructs a plan for a query q, the same plan can be reused to execute a query q' if $W_q[i] \geq W_{q'}[i]$ for all nodes i. Examples of queries that share the same plan are the queries for the maximum temperature and the average humidity in a building. For both queries, a node transmits one data report in a single step (i.e., $W_{max}[i] = W_{avg}[i] = 1$ for all nodes i) if the slot size is sufficiently large to hold two values. For the max query, the outgoing packet includes the maximum value of the data reports from itself and its children. For the average query, the packet includes the sum of the values and the number of data sources that contributed to the sum.

The precision to which the workload demand can be computed depends on the nature of the aggregation function. Three cases are worth highlighting. First, when statistical functions such as min, max, average, or histograms are computed over a set of sensors, the workload on each node remains constant (see previous examples). Second, when sensors have a fixed rate, the load of each node may be computed easily based on its location in the routing tree by summing the workload of the descendant nodes and that of the node and dividing it by the size of a packet. For these two common uses, the workload of each node may be computed precisely. Finally, when sensors have variable rates, we advocate for constructing plans for the maximum data rate produced by each source. While this results in some internal fragmentation it keeps the number of plans to a minimum.

3.3 The Scheduler

In this section, we first describe how to construct a global conflict-free schedule. We then present an efficient local scheduling algorithm. For clarity, in this section, we assume that all queries are executed according to the same plan, i.e., they belong to the same query class. We extend our solution to handle multiple query classes in the next section.

Definitions. Each instance is executed according to the plan of its query class. We use $E_{q,k}[s]$ to denote the set of transmissions assigned to step s of $I_{q,k}$'s plan. We say that two steps of instances $I_{q,k}$ and $I_{q',k'}$ are *conflict-free* $E_{q,k}[s] \parallel E_{q',k'}[s']$ if all pairs of transmissions in $T_c[s] \cup T_{c'}[s']$ are conflict-free. We also use the notation $E_{q,k}[s] \not\parallel E_{q',k'}[s']$ to denote that the two steps conflict with each other. The scheduler executes steps such that: 1) All steps executed in a slot are conflict-free. 2) The relative order in which the steps of an instance are executed is preserved: if step $E_{q,k}[s]$ is executed in time slot i, step $E_{q,k}[s']$ is executed in slot i' and $s > s'$ then $i > i'$. This ensures that the precedence constraints required by aggregation are enforced.

The brute-force approach. Let us consider a brute-force algorithm to dynamically determine what steps may be executed in a slot. We say step $E_{q,k}[s]$ is *ready* if $E_{q,k}[s-1]$ has been executed. The first step $E_{q,k}[1]$ is ready when the instance $I_{q,k}$ is released at time $R_{q,k}$. Intuitively, the brute-force approach schedules in each slot multiple *conflict-free* and *ready* steps. Priority is given to executing steps in the transmissions plans of instances with earlier release times. To determine what steps may be executed in a slot, we need to know if any two steps in the plan conflict. To facilitate this, we construct a conflict table of size $L_q \times L_q$ that stores the conflicts between any pairs of steps in the plan of the query class. Fig. 5a shows the conflict table of the plan shown in Fig. 2. Fig. 5b shows the transmission schedule constructed using the brute-force approach under saturation conditions.

The brute-force approach constructs the schedule as follows: Initially, $E_{q,1}[1]$ is the only step ready and it is executed in slot 1. In slot 2, the steps $E_{q,1}[2]$ and $E_{q,2}[1]$ are ready. However, the earliest slot when $E_{q,2}[1]$ may be executed is slot 4 since according to the conflict table $E_{q,2}[1] \not\parallel E_{q,1}[1..3]$. So, in slot 4, we schedule $E_{q,1}[4]$ and $E_{q,2}[1]$. A more interesting case occurs when scheduling the steps in slot 6. In slot 6, $E_{q,1}[6]$ is executed since it has the earliest release time. $E_{q,2}[3]$ cannot be executed in slot 6 since $E_{q,2}[3] \not\parallel E_{q,1}[6]$. However, $E_{q,3}[1]$ is ready and its execution does not conflict with $E_{q,1}[6]$. Therefore, it is also executed in slot 6. The process continues to construct the schedule presented in Fig. 5b.

The brute-force approach is impractical due to its computation and storage costs. The computation time for determining what steps to schedule in a slot is quadratic in

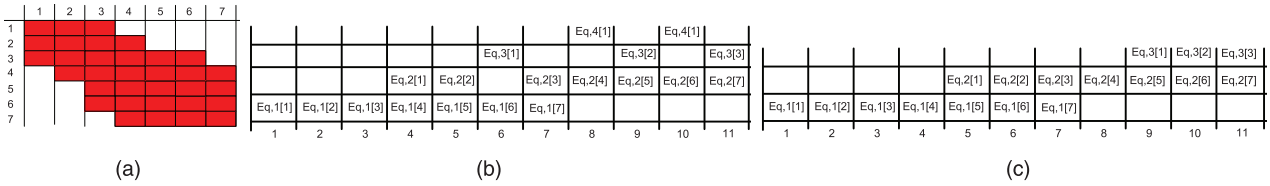


Fig. 5. The conflict table captures the transmission conflicts between pairs of steps from the plan shown in Fig. 3. A conflict is represented by the red rectangle. Based on the conflict table you can construct schedules either by brute-force or through the DCQS approach. (a) Conflict table. (b) Brute-force approach. (c) DCQS approach.

the number of ready steps in all instances that have been released. The memory cost for storing the conflict table is quadratic in the length of the plan. To alleviate these problems, we may trade some of the throughput in favor of reduced computational and storage costs. To this end, we impose the additional constraint that the execution of an instance cannot be *preempted*. The execution of a query instance is not preempted if, once its first step is executed, the subsequent steps of its plan are executed *without gaps* in the following slots. In Fig. 5b, the schedule constructed by the brute-force approach does not meet this constraint since the execution of $I_{q,2}$ is preempted in slot 6.

Minimum interrelease time. We define the *minimum interrelease time*, Δ, as the minimum number of slots the execution of $I_{q,k}$ must be delayed after another instance $I_{q',k'}$ starts executing such that the execution of $I_{q,k}$ and $I_{q',k'}$ are conflict-free. In other words, any two instances are conflict-free as long as their interrelease time is at least Δ.

Consider the execution of two consecutive instances $I_{q',k'}$ and $I_{q,k}$ (from one or two queries). If the interrelease time between $I_{q,k}$ and $I_{q',k'}$ is δ and the execution of instances cannot be preempted, then the steps $E_{q,k}[1]$ and $E_{q',k'}[\delta+1]$ are executed in the same slot of the transmission schedule. Hence, δ must be selected to ensure that $E_{q,k}[1] \parallel E_{q',k'}[\delta+1]$. However, the execution of $I_{q,k}$ may start in any slot after δ steps in the plan of $I_{q',k'}$ are executed. Therefore, we must guarantee that $E_{q,k}[1]$ does not conflict with $E_{q',k'}[\delta+1]$ *and* any of the subsequent slot executions, i.e., $E_{q,k}[1] \parallel E_{q',k'}[\delta+i+1]$ for all $i \in [0, L_c-\delta-1]$, where L_c if the length of the plan of query class c. The minimum interrelease time, Δ, is the smallest number such that the execution of any step $E_{q,k}[s]$ does not conflict with $E_{q',k'}[s+\delta+i+1]$, where $s \leq L_c$ and $i \in [0, L_c-s-\delta-1]$. Thus, the minimum interrelease time is

$$\Delta = \min_{\delta \in [1, L_c]} (E_{q,k}[s] \parallel E_{q',k'}[s+\delta+i+1]) \quad \forall i \in [0, L_c-s-\delta-1], s \leq L_c. \tag{1}$$

The minimum interrelease time is a measure of the degree of parallelism that is achieved in query execution. In the worst case, when $\Delta = L_c$ a single instance is executed at a time.

The scheduler. Each node employs a *local* scheduler that schedules the transmissions of all instances. The state of the scheduler includes: the start time and period of all queries, the plan's length, and the minimum interrelease time. Note that if all nodes have a consistent view of these parameters, they will construct independently the same schedule. The scheduler also knows the steps in which the host node transmits or receives. However, the scheduler does not need to know the specific steps in which any other nodes transmit or receive.

The scheduler has two FIFO queues: a *run* and a *release* queue. The *release* queue contains all instances released but not being executed. The *run* contains the instances to be executed in slot s. Although the *run* queue may contain multiple instances, a node is involved in transmitting/receiving for at most one instance (otherwise, it would be involved in two conflicting operations). A node n determines if it transmits/receives in s by checking if it is assigned to transmit/receive in any of the steps to be executed in s. If a node does not transmit/receive in s, it turns off its radio during s.

The scheduler enforces a minimum interrelease time of at least Δ between the start time of any two instances by starting an instance in two cases: 1) when no instances are executed (i.e., $\text{run} = \emptyset$) and 2) when the step distance between the head of the *release* queue (i.e., the instance with the earliest release time) and the tail of the *run* queue (i.e., the last instance that started) is larger then Δ. When an instance starts, it is moved from the *release* queue to the *run* queue.

The scheduler is simple and efficient. Thus, it is feasible to run it on resource-constrained devices. When a new instance is released, the scheduler inserts it in the *release* queue. In each slot, DCQS determines what instances should start. This operation takes $O(1)$, since it involves comparing the step distance between the instances at the head of *release* queue and tail of *run* queue with the minimum step distance. To determine if a node should send, receive, or sleep, DCQS iterates through the instances in the *run* queue. This requires $O(|run|)$ time if each node maintains a bit vector indicating whether it transmits, receives, or sleeps in each step of a plan. Thus, the complexity of the operations performed in a slot is $O(|run|)$. Due to the efficiency of the scheduler, a node may construct the transmission schedule dynamically at runtime. Moreover, the memory cost of the algorithm is also lower than the brute-force approach. The scheduler maintains only the minimum interrelease time instead of a table of conflicts as in the brute-force approach.

Fig. 5c presents the schedule constructed when the minimum interrelease time Δ is four slots. The constructed schedule has slightly lower throughput than the one constructed using the brute-force approach. This is due to the fact that DCQS does not preempt instances once they are executing. This illustrates our decision to trade off throughput to reduce the memory and processing costs. In spite of this, our results show that DCQS still achieves significantly higher throughput than existing solutions (see Section 5).

Analysis. In the following, we prove three properties of the DCQS scheduler. First, we prove that the scheduler never schedules conflicting transmissions in the same slot. Second, we analyze the network capacity in terms of query completion rate under DCQS. This result is important because it enables us to prevent the network workload to exceed DCQS's capacity (as described in Section 4.2). Finally, we characterize the energy consumption of a node.

Theorem 1. *The scheduler executes conflict-free transmissions in all slots.*

Proof. Consider the scheduler constructing a schedule for the following sequence of instances $I_{q_1,k_1}, I_{q_2,k_2}, I_{q_3,k_3} \ldots$. We note that the scheduler ensures that the pairs I_{q_1,k_1}, I_{q_2,k_2}, and I_{q_2,k_2}, I_{q_3,k_3} are conflict-free, but it does not directly ensure that I_{q_1,k_1} and I_{q_3,k_3} are conflict-free. In general, we must prove that I_{q_i,k_i} does not result in any conflict when its schedule overlaps with any instance I_{q_j,k_j} where $j > i$.

Let s_i and s_j be the steps in the plans of I_{q_i,k_i} and I_{q_j,k_j} that the scheduler assigns in the same slot. Since the scheduler enforces a minimum interrelease time of Δ between consecutive instances then $s_j - s_i \geq (j-i) \cdot \Delta \geq \Delta$. Thus, the scheduler executes conflict-free transmission in any slot. □

Property 1. *The maximum query rate of DCQS is $\frac{1}{\Delta \cdot slotSize}$ where $slotSize$ is the size of a slot in seconds.*

This is the case since an instance can be released every Δ slots for a maximum query completion rate of $\frac{1}{\Delta \cdot slotSize}$. Therefore, DCQS does not exceed its capacity if:

$$\sum_q \frac{\Delta}{P_q/slotSize} \leq 1, \tag{2}$$

where, P_q is the period of query q.

3.4 The Multiple Query Class Scheduler

We now extend our scheduler to support multiple query classes. To this end, we must refine the definition of minimum interrelease time to accommodate the case when instances are executed according to different plans. We define the *minimum interrelease of query classes c and c'* $\Delta(c, c')$ as the minimum number of slots an instance of class c' must wait after an instance of class c started such that there are no conflicts. Note that Δ is not commutative.

Given the minimum interrelease times between any ordered pairs of query classes, the scheduler needs to control the interrelease times of two consecutive query instances based on their query classes. We note that the storage cost of the multiple class scheduler is quadratic in the number of query classes, since it must store the minimum interrelease time of each pair of query classes. However, as discussed in Section 3.2, the number of query classes in a WSN is usually small.

When all queries belong to a single query class, the scheduler only needs to check if the step difference between the instance at the head of the *release* queue and the instance at the tail of the *run* queue exceeds the minimum interrelease time to guarantee conflict-free transmissions. However, in the case of multiple query classes, to guaranteee that *all* minimum interrelease times are enforced, the scheduler should check if the step difference between the instance at the head of *release* queue and *all* instances in *run* queue exceeds the minimum interrelease times between their respective query classes. An efficient mechanism for doing this is for the scheduler to keep track of the slot when the last instance of each query class started. To enforce *all* minimum interrelease times it suffices for the time when the last instance of each class started to exceed the minimum interrelease time between that class and the class of the instance at the head of the *release* queue. Thus, the number of comparisons necessary to enforce the minimum interrelease time equals the number of query classes. As a consequence, the scheduler handles multiple classes without increasing its computational complexity significantly since the number of classes is a constant (i.e., it does not depend on the number of instances either in *release* or in *run* queues).

Equation 2 allows us to compute DCQS's maximum query throughput for a single query class. It is easy to see that a conservative bound on the maximum query rate for multiple classes is at least $\frac{1}{\Delta_{max} \cdot slotSize}$, where Δ_{max} is the maximum minimum interrelease time for all pairs of query classes. However, this approach significantly underestimates the maximum query rate supported by DCQS particularly when the values of Δ differ significantly. To reduce the pessimism of the bound, we derive a *sufficient* condition for ensuring that all queries may be scheduled without exceeding the network capacity.

Theorem 2. *Given a set of queries classes C, DCQS may schedule all queries without exceeding the network capacity if:*

$$\sum_q \frac{\max_{c' \in C} \Delta(cls(q), c')}{P_q/slotSize} \leq 1, \tag{3}$$

where $cls(q)$ is q's class and P_q is q's period.

Proof. Consider a query instance $I_{q,k}$ of class $c = cls(q)$. Any query instance $I_{q',k'}$ of class c' may start after $I_{q,k}$ completes $\Delta(c, c')$ steps in its execution. As such, in the worst case, $I_{q,k}$ will delay the execution of any query instance $I_{q',k'}$ for at most $\max_{c' \in C} \Delta(c, c')$. In other words, $I_{q,k}$ prevents other query instances from being executed for at most $\max_{c' \in C} \Delta(c, c')$. Hence, the network utilization of q, i.e., fraction of time the network executes q alone is: $\frac{\max_{c' \in C} \Delta(c,c')}{P_q/slotSize}$. Thus, DCQS does not exceed its capacity if there is sufficient time to execute all queries, i.e., the total utilization of all queries does not exceed 1:

$$\sum_q \frac{\max_{c' \in C} \Delta(cls(q), c')}{P_q/slotSize} \leq 1.$$

□

While (3) is still a conservative bound on query capacity, as shown in our simulation study, it is close to the actual achievable throughput and hence is suitable for rate control.

It is worth highlighting that DCQS supports the case when queries involve overlapping node subsets. In this case, DCQS would construct a plan for each query and compute the minimum interrelease time between the two plans. Nodes shared by multiple queries will have higher

transmission demand including transmissions for all queries in which they are involved. It is important to note that DCQS does not construct an optimal schedule for executing these queries, but rather constructs a plans for executing each query in isolation. While this may result in suboptimal schedule, the construction of independent plans that may be executed concurrently by enforcing the minimum step distance facilitates the development of schedulers with small runtime overhead.

3.5 Distributed Planner

In this section we present a distributed planner which uses only neighborhood information in constructing plans. Specifically, a node knows only its adjacent communication and interference edges. We say that a node is in n's *one-hop neighborhood* if there is a communication or interference edge between it and n. The two-hop neighborhood of node n includes n's one-hop neighbors and their one-hop neighbors. After running the decentralized planner, a node knows its *local plan* which contains the steps of its two-hop neighbors.

To construct a local plan, a node communicates only with its one-hop neighbors. However, some of the neighbors may lie outside the node's communication range. A routing algorithm or limited flooding may be used to communicate with these nodes over multiple hops.

A node n constructs a plan in three stages: plan formulation, plan dissemination, and plan reversal. The formulation stage starts when a node n becomes the highest priority eligible node in its one-hop neighborhood. When this occurs, n broadcasts a *Plan Request* to gather information about transmissions which have already been assigned. To construct a conflict-free plan, n must know the steps in which its two-hop neighbors with higher priorities were assigned. Upon receiving the *Plan Request* from n, each one-hop neighbor checks if there is a node in its own one-hop neighborhood that has a higher priority than n. If no such node exists, the receiver responds with a *Plan Feedback* packet containing its local plan. Otherwise, the node does not reply. After a time-out, node n will retransmit the *Plan Request* to get any missing *Plan Feedback* from its one-hop neighbors. Since all *Plan Feedback* are destined for n, to reduce the probability of packet collisions, nodes randomize their transmissions in a small window. Once n receives the *Plan Feedback*, it has sufficient information to assign its transmissions to its parent using the same method as the centralized planner. In the second stage, n disseminates its local plan to its one-hop neighbors via a *Plan Send*. Upon receiving a *Plan Send*, a node updates its plan and acknowledges its action via a *Plan Commit*.

To ensure that DCQS constructs a conflict-free schedule, neighboring nodes must have consistent plans. We note that the distributed planner achieves this objective through retransmission when needed. If a *Plan Feedback* from a neighbor is lost, n assumes that a higher priority node has not yet been scheduled and retransmits the *Plan Request* until it has received *Plan Feedback* from each neighbor or reached the maximum number of retransmissions. Similarly, during the plan dissemination stage, node n retransmits the plan until all its neighbors acknowledge the correct reception of its *Plan Send* via a *Plan Commit*.

Finally, the planner reverses the plan. To do this, a node must know the length of the plan. We take advantage of the routing tree and data aggregation to compute the length of the plan as follows: A node computes the length of its local transmission plan length based on the maximum step number in which a transmission/reception is assigned. The node aggregates its local length of the plan with that of its children by taking the maximum. The result is sent to its parent. At the base station, the plan length may be computed. The root then uses the routing tree to disseminate this value. Upon receiving the plan length, a node reverses its plans.

It is important to note that DCQS relies on nodes having consistent state for proper execution of plans. We ensure that this is the case by bounding the time for each phase of the plan construction. If the planning process fails DCQS will abort the construction of the plan if the plan construction does not succeed within the allotted time. The node that failed to respond during plan construction is considered disconnected and removed from the IC graph. At this point, the process of plan construction is restarted. There is no point at which DCQS will start using a plan, before all nodes are synchronized with respect to the plan they are using.

Distributed computation of Δ. We now enhance the distributed planner to compute the minimum interrelease times. The key to computing the minimum interrelease time distributedly lies in the observation that a node may compute its local value for the minimum interrelease time based on its local plan and its local knowledge of the IC graph according to equation (1). The minimum interrelease time of the global plan is the maximum of the minimum interrelease times of the local plans. This suggests that, similar to the length of the plan, the global minimum interrelease time can be computed using in-network aggregation. In fact, the two may be computed concurrently. Once the aggregation process is complete, the root can compute the length, and minimum interrelease time of the plan and then disseminate them to all nodes.

4 Handling Dynamics

4.1 Dynamic Workload

DCQS can efficiently adapt to changes in the workload including arrival, deletion, and rate change of queries. Consider the case where a user issues a new query. The query service disseminates the query parameters to all nodes. Next, DCQS checks if a plan for the class of the issued query was constructed. If no plan was constructed, DCQS uses the decentralized planner to compute a new plan and the minimum interrelease times. DCQS isolates the execution of current queries from the setup of new queries by periodically reserving slots for protocol maintenance. During these slots, the planner computes the plan and minimum interrelease times. Once they are computed, the scheduler has sufficient information to construct a conflict-free schedule which accounts for execution of the new query.

If a query from the same class was previously issued, a plan for that class has already been constructed. As previously mentioned, queries from the same class share the same plan and minimum interrelease time. Since

usually there are only a small number of query classes, it is likely that DCQS already computed the plan and minimum interrelease times of a class. In this case, DCQS can handle the new query without any additional overhead. Similarly, DCQS can also handle query deletions and rate changes of existing queries without any overhead.

4.2 Preventing Overload

A key advantage of DCQS is that it has a known capacity bound in terms of the maximum query completion rate. Using inequality (3), we can easily detect overload conditions which obviates the need for complex congestion control mechanisms. When the user issues a query, DCQS uses inequality (3) to determine if the capacity is exceeded. If the capacity is exceeded, we consider the following two options. First, the user may be notified that the query will not be executed because the network capacity would be exceeded. Second, DCQS may reduce the rates of existing queries to allow the new query to be executed. For example, a simple rate control policy is to reduce the rates of all queries proportionally. This rate control policy is used in our simulations. As discussed in the previous section, DCQS may modify the period of a query without recomputing the plan or minimum interrelease times. Therefore, the only overhead is to disseminate the updated rates.

4.3 Robustness against Network Changes

We now describe how DCQS handles topology changes due to node or link failures. For DCQS to detect topology changes, we increase the slot size to allow a parent to acknowledge the correct reception of a data report from its child. A child can detect the failure of its parent or their link if it does not receive ACKs from its parent for several consecutive transmissions. A parent detects a child failure if it does not receive any data reports from that child for a number of query periods.

For all nodes to maintain a consistent schedule, DCQS must ensure the following: 1) the *two-hop neighbors* of a node have a consistent view of its local transmission plan, which dictates when the node transmits and receives data reports; 2) *all nodes* have consistent information about the length of the plans and the minimum interrelease times. In response to topology changes, the routing tree must be adjusted. Consider the case when a node n detects that its parent p failed and, as a result, it must select a new parent p'. This entails the planner assigning a step in the plan for $\overrightarrow{np'}$, while the step in which the transmission $\overrightarrow{np}$ is scheduled must be reclaimed. If $\overrightarrow{np'}$ can be assigned to the step in which $\overrightarrow{np}$ was scheduled or a different step *without conflicts* then DCQS only needs to update the local transmission plan. This involves node n disseminating its updated plan to its two-hop neighbors. If this is not possible, then DCQS recomputes a new plan. We note that the computation of the new plan affects only nodes with lower priority than n. If during the computation of the plan either the minimum interrelease times or the plan lengths are modified, this information must be disseminated to all nodes in the network. Consider the case when a child c of node n failed. In this case, the step in which c is assigned should be reclaimed. To reduce the overhead, DCQS reclaims such slots only when other topology changes occur.

To reduce the cost of handling topology changes, we now describe an approach to constructing robust plans that can tolerate some topology changes. To handle this, we change the mechanism used to adapt the routing tree in response to link or node failure. We allow a node to change its parent in the routing tree as long as the new parent is selected from a predefined *set of potential parents*. Our goal is to construct plans that are insensitive to a node changing its parent under the constraint that the new parent is in the set of potential parents. To this end, we introduce the concept of *virtual transmissions*. Although node n actually transmits to a single potential parent, we construct the plan and compute the minimum interrelease times as if n transmits to *all* potential parents. We trade off some of the throughput in favor of better tolerating topology changes. This is similar to other TDMA algorithms designed to handle topology changes [13], [14].

Wireless links are known to have variable quality as environmental conditions change. During the computation of workload demands for each node, the user must allocate sufficient time slots for potential packet retransmission to ensure reliability. DCQS already provides some robustness against change in link quality by having multiple parents among which a node may switch. However, DCQS may also account for variations in link quality through a different mechanism. DCQS can accommodate such changes by increasing the workload of a link based on its quality. For example, link layer commonly computes the expected number of transmissions (ETX) necessary for correctly delivering a packet. DCQS could allocate a node to transmit up to ETX times a packet to ensure reliable delivery. We note that to ensure robustness, a parent is forced to transmit at the scheduled time even if it did not receive all data reports from its children.

4.4 Supporting Other Traffic

DCQS is optimized for improving the performance of periodic queries. However, other types of traffic may also exist (e.g., data dissemination and aperiodic queries). The simplest solution for handling these transmissions is to periodically dedicate slots for their transmission. Transmissions during these slots are done using typical CSMA/CA techniques. This approach reserves a portion of the bandwidth for other types of traffic. Moreover, it is straightforward to account for these additional slots in our analysis.

4.5 Time Synchronization

DCQS requires that all nodes within the interference range be time synchronized. The sensor network community has proposed several efficient time synchronization protocols [15]. For example, FTSP has an average time synchronization error of $1.4\ \mu s$ per hop with minimal communication overhead. To account for time synchronization errors, DCQS employs guard time intervals. Accordingly, in the beginning of a slot, a short guard time interval is observed by all nodes to account for clock drift. As shown in earlier publications, the guard time intervals alleviate the need for fine-grained clock synchronization [16].

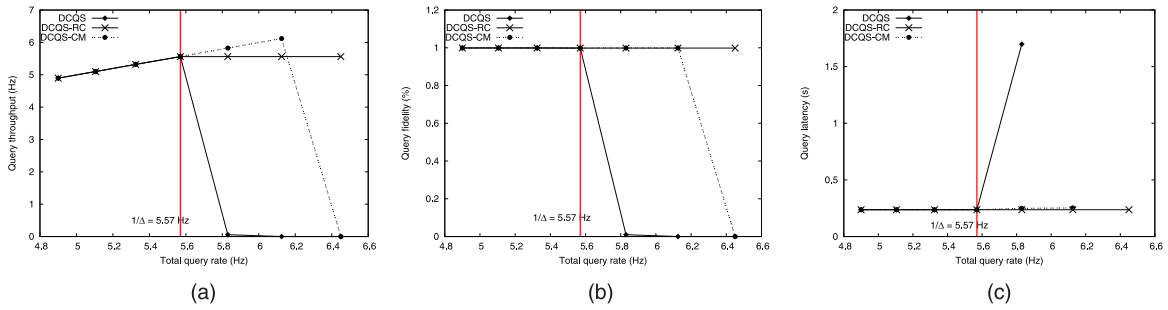


Fig. 6. Validation of capacity bound for single query. (a) Query throughput. (b) Query fidelity. (c) Query latency.

5 Performance Evaluation

We implemented the distributed version of DCQS in NS2. We used a two-ray propagation model at the physical layer. Since DCQS is targeted at *high* data rate applications such as structural health monitoring we configured our simulator according to the 802.11b settings. An overview of those deployments can be found in [17]. In our simulations, the bandwidth is 2 Mbps and the communication range is 125 m. We set the slot size to 8.16 ms which is sufficient to transmit 2 KB of data. A simulation run takes 200 s. Unless otherwise mentioned, each data point is the average of five runs and the 90 percent confidence intervals are plotted.

In the beginning of the simulation, the routing tree is constructed. The node closest to the center of the topology is selected as the base station. The base station initiates the construction of the routing tree by flooding setup requests. A node may receive multiple setup requests. The node selects as its parent the node that has the highest RSS among those with smaller depth than itself. The IC graph is constructed according to the method described in Section 2.2.

For performance comparison, we ran two baselines: 802.11b[18] and DRAND[19]. 802.11b is a representative CSMA/CA-based protocol, while DRAND is state-of-the-art TDMA protocol. Unlike DCQS, DRAND does not account for the interference relationships among nodes. Hence, the schedule it constructs may still result in collisions. To avoid this problem, we modified DRAND to treat the interference edges in the IC graph as communication edges. We augmented DRAND with a sleep-scheduling policy: a node remains awake if DRAND schedules itself or one of its children to transmit; otherwise, the node is turned off.

We evaluate the performance of three versions of DCQS: DCQS, DCQS-RC, and DCQS-CM. DCQS-RC and DCQS denote DCQS running with or without rate control. In DCQS, the planner uses the minimum interrelease time to decide when different steps may be executed concurrently. As discussed in Section 3.3, by knowing the entire conflict matrix one may make better scheduling decisions. DCQS-CM uses the conflict matrix rather than the minimum interrelease time to make scheduling decisions. We will use DCQS-CM to evaluate the degree of pessimism introduced by the minimum interrelease time. Note that DCQS-CM uses significantly more memory and has longer runtime overhead than DCQS. As a result, DCQS-CM may not be suitable for resource-constraint devices.

We focus on the following metrics: query throughput, query fidelity, and query latency. The query throughput is the number of queries instances completed per second. During the simulations, data reports may be dropped preventing some sources from contributing to the query result. We define the query fidelity to be the ratio of the number of sources that contributed to a query result received by the base station and the number of sources.

5.1 Single Query Class

The first set of experiments assumes that all queries belong to the same class. Under this setup, we will

1. validate the analytical results on DCQS's capacity,
2. compare the performance of DCQS against the baselines,
3. compare the scalability of DCQS to that of DRAND when the network size is varied,
4. evaluate DCQS's overhead in relation to the network size, and
5. evaluate the impact of the virtual transmissions on DCQS's performance.

5.1.1 Tightness of Capacity Bound

The first experiment is designed to validate our capacity result for a single query and show the effectiveness of the rate control policy. In this experiment, a single query is executed in the network size $675\ \mathrm{m} \times 675\ \mathrm{m}$. The topology is divided into grids of $75\ \mathrm{m} \times 75\ \mathrm{m}$. In each grid, a node is placed at random. We present results from a single run since DCQS constructs plans with different Δ values for different topologies. Under these settings, DCQS constructed a plan with $\Delta = 22$ slots. According to Theorem 1, the maximum query rate that DCQS may support is $\frac{1}{22 \cdot 8.16\ ms} = 5.57$ Hz. The vertical lines in Fig. 6 indicates the DCQS's capacity. To validate the capacity bound the query rate is varied from 4.9 to 6.45 Hz.

Fig. 6a shows the query throughput as the query rate is increased. We observe that the increase in query rate is matched by an increase in the query rate until DCQS's capacity (5.57 Hz) is reached. When workload exceeds the DCQS's capacity, the performance of DCQS degrades drastically. As discussed in Section 4.2, rate control may be used to avoid triggering the capacity bottlenecks. Fig. 6a shows that DCQS-RC maintains its good performance even when the offered load exceeds the DCQS's capacity. Since DCQS-CM uses the conflict matrix, it may execute more concurrent steps than DCQS. As a result, DCQS-CM may support a query rate as high as 6.125 Hz, which is a 9.9 percent

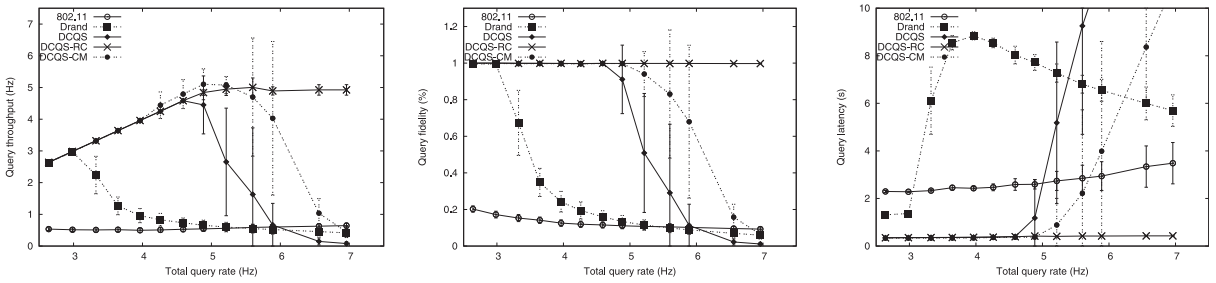


Fig. 7. Performance comparison when executing four queries and their base rate is varied. (a) Query throughput. (b) Query fidelity. (c) Query latency.

improvement of DCQS. However, this improvement comes at the cost of increased memory and processing overheads.

DCQS achieves 100 percent query fidelity below its network capacity as shown in Fig. 6b. This result shows that the schedule constructed by DCQS is conflict-free, validating the correctness of our algorithms. Moreover, DCQS-RC, which uses the rate control policy, avoids the drop in query fidelity under overload conditions. A similar pattern may be observed in terms of delay as shown in Fig. 6c. DCQS, DCQS-RC, and DCQS-CM have similar latencies up to the their capacity. When the capacity is exceeded and rate control is not used, the query latency increases sharply as shown by DCQS and DCQS-CM curves.

5.1.2 Multiple Queries

This set of experiments compares the performance of DCQS to that of the baselines under different workloads. The workload is generated by running four queries with different rates. The ratio of the rates of the four queries $Q_1 : Q_2 : Q_3 : Q_4$ is 8:4:2:1. We refer to Q_1's as the base-rate. We vary the workload by changing the base rate. The start time of the queries is spread evenly in an interval of 81.6 ms. The topology setup is identical to the previous experiment.

Fig. 7a shows the query throughput as the base rate is varied. A common trend may be observed: the protocols match the increase in the total query rate up to their respective capacity and then their performance plummets. The lowest throughput is obtained by 802.11 protocol. The reason for this outcome is that the capacity of 802.11 is exceeded in all tested settings. This is because contention-based protocols perform poorly under heavy workloads. DRAND outperforms 802.11 delivering all data up to a total query rate of 2.98 Hz. All of the DCQS variants outperform DRAND. DCQS-RC achieves a maximum query rate of 5 HZ which is 67 percent higher than DRAND. This result is attributed to the fact that DRAND assigns slots to nodes fairly. Fair allocation is unsuitable for queries in WSNs in which nodes have nonuniform workloads: for example, a node with more children needs to receive more messages per each query instance. As in the previous experiment, DCQS-RC maintains its good performance even under overload conditions. This shows that our rate control policy works not only in the single query case, but also for multiqueries. DCQS-CM delivers all data reports up to a total query rate of 5.21 Hz. This is an increase of 4 percent over DCQS-RC and 74 percent over DRAND. This result highlights that the reduction in throughput introduced by the additional constraint of not preempting queries is small.

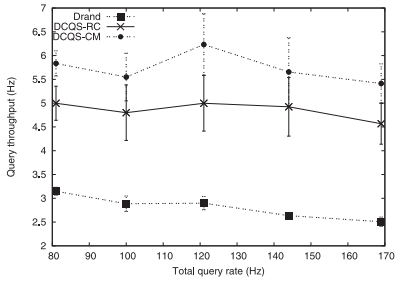
Fig. 7b shows the query fidelity of the protocols. As expected, 802.11 has poor query fidelity, whereas the TDMA protocols perform much better. DRAND achieves high query fidelity up to its capacity of 2.98 Hz after which it plummets due to queuing overflow. In contrast, DCQS-RC maintains 100 percent fidelity for all tested query rates. Fig. 7c shows the query latency of all protocols. Even for low query rates, DCQS has significantly outperforms DRAND. For example, when the query rate is 2.64 Hz, DRAND has a query latency of 1.31 s. In contrast, DCQS has a latency of 0.35 s which is 73 percent lower. DRAND has a higher query latency because at each hop a node may need to wait for the duration of an entire frame before it may transmit a packet to the parent. In contrast, DCQS accounts for the precedence constraints introduced by data aggregation when constructing the plans. This results in a significant reduction in the query latency.

This set of experiments indicates that DCQS significantly outperforms both 802.11 and DRAND in the considered metrics. Two factors contribute DCQS's high performance. First, the planner constructs plans based on a heuristic that accounts for the precedence constraints introduced by data aggregation. This is effective in reducing message latency. Second, the scheduler overlaps the execution of multiple instances to increase query throughput.

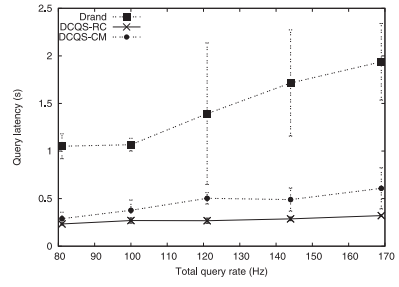
5.1.3 Impact of Topology Size

In the following, we evaluate the scalability of the TDMA protocols. To this end, we constructed topologies with an increasing number of nodes by increasing the deployment area and keeping the node density constant. All topologies are squares with edges of sizes 675, 750, 825, and 900 m. Each area is divided into grids of $75\ \mathrm{m} \times 75\ \mathrm{m}$ and a node is placed at random in each grid.

Fig. 8a shows the query throughput achieved by DCQS and DRAND for each topology. The capacity of DCQS-RC was computed theoretically and then verified experimentally. To determine capacity of DRAND or DCQS-CM, we increased the query rate until the query fidelity dropped below 90 percent. This is reasonable since the DRAND and DCQS-CM drop packets only if a node's queue fills up. When the topology has 81 nodes, DCQS outperforms DRAND by 58 percent. When the topology contains 169 nodes, the performance gap between the two protocols increases, DCQS outperforming DRAND by 115 percent.



(a)



(b)

Fig. 8. Performance comparison for topologies of different sizes. (a) Query throughput. (b) Query latency.

The increasing performance gains of DCQS are the result of being able to pipeline the execution of queries as the networks become larger. While the difference between DCQS and DCQS-CM is between 16-24 percent without a clear trend, it is worth noting that 90 percent confidence intervals overlap indicating that the performance may not be statistically significant.

Fig. 8b shows the query latency at the maximum query rate supported by each protocol. The query latency of all protocols DCQS increases with the topology size. However, DCQS's rate of increase is significantly lower than DRAND's. The key to understanding this result is that the one-hop delay of DRAND is significantly larger than that of DCQS. The one-hop delay corresponds to the slope of the shown curves. When using DRAND, a node often needs to wait for the entire length of a frame before it may transmit its packet. In contrast, DCQS has low one-hop delays. Two factors contribute to this. First, DCQS organizes its transmissions to account for the precedence constraints introduced by data aggregation. Second, DCQS executes multiple query instances concurrently. As such, the time until the query instance starts being executed is reduced. DCQS-CM has a slightly higher query latency than DCQS-RC. This can be justified by the fact that DCQS-CM buffers more queries for execution than DCQS-RC which reduces the query rates to avoid triggering the capacity bottlenecks.

5.1.4 Communication Costs

To evaluate the overhead of DCQS, we have collected statistics regarding the different types of packets transmitted by DCQS. We distinguish the following categories. The tree construction category includes all packets exchanged during the construction of the routing tree. The ICG construction category includes all the packets exchanged for constructing the IC graph. The planer category includes all the overhead associated with constructing plans. Finally, the minimum interrelease category shows the overhead of computing the minimum interrelease time. The overhead in packets for the topologies considered in the previous experiment are shown in Fig. 9.

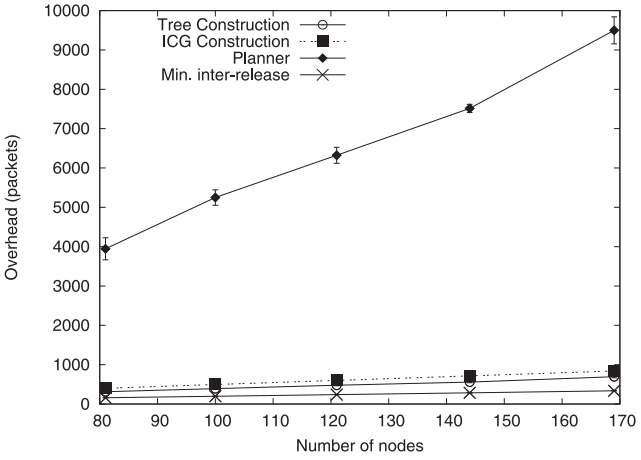


Fig. 9. Communication costs for different size networks.

Fig. 9 indicates that the cost of constructing plans dominates the DCQS overhead. This highlights the importance of the plan sharing strategies. The planning overhead is linear in the number of nodes in the network. Note that we did not make any particular effort in optimizing the performance of the planner. In fact, the planner does not take advantage of the broadcast nature of the wireless medium using only unicast packets to disseminate the plans. Therefore, we expect that the scheduling cost of construct plans may be reduced.

5.1.5 Virtual Transmissions

To evaluate the impact of the virtual transmissions approach presented in Section 4.3, we measured the performance of DCQS when the number of parents is varied from one to three. The results shown in Fig. 10 are obtained from five randomly generated topologies with the same parameters as in the previous experiment. We first computed DCQS-RC's maximum query rate and then verified it empirically. In addition, we measured DCQS-RC's latency at the maximum query rate.

Fig. 10 indicates that the DCQS's maximum query rate degrades as the number of potential parents is increased. The increase to having two or three potential parents leads in a throughput reduction of 9.8 and 12.6 percent, respectively. However, DCQS improved throughput over traditional TDMA protocols by at least 58 percent. Therefore, even if DCQS would construct plans in which multiple potential parents, there still are significant performance gains. The use of virtual transmissions also increases the query latency as shown in the same figure. The increase to two and three potential parents results in an increase in query latency of 7.6 and 9.6 percent, respectively. This is due to the fact that the addition of potential parents introduces new precedence constraints that force a node to wait longer before receiving all data reports from its children.

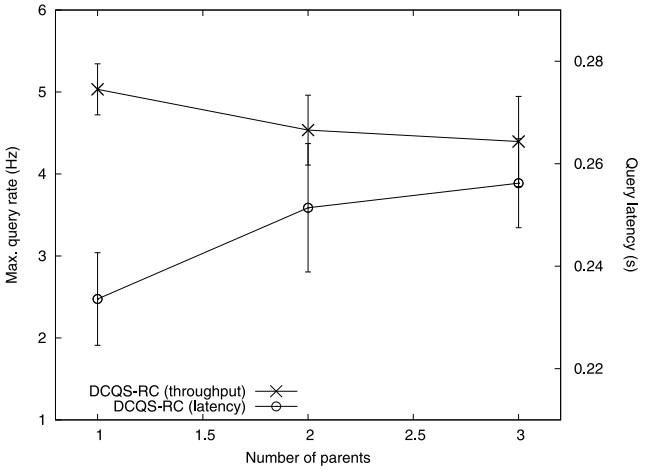


Fig. 10. Impact of virtual transmissions on DCQS.

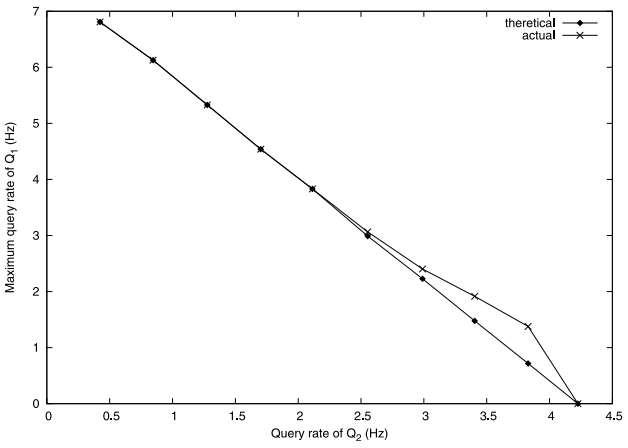


Fig. 11. Validation the capacity bound for multiple query classes.

5.2 Multiple Query Classes

The second set of experiments is used to evaluate DCQS with multiple query classes. We create different query classes by varying the sources of the queries. For each query class, we select at random a fraction of the leaf nodes as data sources. We note that if a node has as descendent a selected leaf node, then it also participates in that query class since it must forward the leaf's data. Similar to the previous experiments, data merging is performed as data are routed to the base station.

5.2.1 Tightness of Capacity Bound

This experiment uses the same network topology as the one presented in Section 5.1.1. However, different from the previous experiment, the workload includes two queries belonging to different query classes. The class of Q_1 involves 40 percent of the leaf nodes as sources, while the class of Q_2 involves all leaf nodes as sources. The obtained minimum interrelease times were $\Delta(c_1, c_1) = 16$ slots, $\Delta(c_1, c_2) = 14$ slots, $\Delta(c_2, c_1) = 29$ slots, and $\Delta(c_2, c_2) = 25$ slots. In Fig. 11, results from a single run are shown.

In Fig. 11, we fix the rate of query Q_2 and compare the theoretical capacity of Q_1 against its actual maximum query rate achieved in the simulations. The *theoretical* capacity of Q_1 is computed according to (3). To evaluate the tightness of the capacity bound, we increase the rate Q_1 until packets start being dropped. We refer to the maximum query rate under which no packet is dropped as the *actual* maximum query rate. As shown in Fig. 11, the theoretical maximum query rate never exceeds the actual maximum query rate. This result shows that the theoretical bound is a conservative bound of the actual query capacity. The theoretical maximum query rate remains the same when Q_2's rate is between 0.42 to 2.55 Hz as shown in Fig. 11. When Q_2's rate exceeds 2.55 Hz, the discrepancy between Q_1's theoretical and actual maximum query rates increases. The maximum difference is 0.66 Hz when Q_2's rate is 4.22 Hz. The slight increased pessimism may be explained as follows: First, both the utilization of Q_1 and Q_2 is overestimated. For example, Q_2's utilization is overestimated because to enforce minimum interrelease time between two instances of Q_2 the scheduler uses $\Delta(c_2, c_2) = 25$; to enforce the minimum interrelease time between an instance of Q_2 followed by an instance of Q_1, the scheduler uses $\Delta(c_2, c_1) = 29$; in contrast, our capacity analysis uses the maximum of the two values to represent Q_2's utilization. A similar argument holds for Q_1. Second, since the sum of utilization of Q_1 and Q_2 must be smaller than 1, overestimating the utilization of Q_2 leads to underestimating the fraction of the capacity that may be used by Q_1, and the pessimism increases as the rate of Q_2 increases. Similar results are obtained when we fix query rate of Q_1 and plot theoretical and actual maximum query rates of Q_2. The maximum difference between the Q_2's actual and predicted query rates is 0.51 Hz. This figure is omitted due to space limitations.

5.2.2 Varying Query Rates

This experiment compares the performance of DCQS to that of the baselines for multiple query classes. The workload comprises three queries with different rates, each belonging to a different class. Q_1, Q_2, and Q_3 include 100, 80, and 60 percent of the leaf nodes as sources, respectively. The ratio of the rates of the three queries Q_1:Q_2:Q_3 is 2.5:1.75:1. We vary the workload by changing the base rate.

Fig. 12a shows the query throughput when the total query rate is varied. As expected, all protocols match the increase in the total query rate up to their respective capacity bounds after which the query throughput degrades. 802.11b consistently has the lowest throughput. DRAND performs better than 802.11b achieving a maximum query throughput of 2.66 Hz. DCQS, DCQS-RC, and DCQS-CM outperform both baselines. DCQS-RC achieves a maximum query throughput of 4.48 Hz which represents a 62 percent improvement over DRAND. DCQS-RC avoids the performance degradation under overload conditions through rate control. DCQS-CM the highest query throughput of 5.23 Hz, which is an improvement of 17 percent over DCQS-RC. However, this comes at the cost of increased processing costs.

Fig. 12b shows the query fidelity as the total query rate is increased. As observed in the previous experiments, after a protocol exceeds its capacity, its query fidelity degrades drastically. DCQS-RC avoid the performance degradation caused by overload by using rate control. DCQS-RC achieves close to 100 percent fidelity in all experiments.

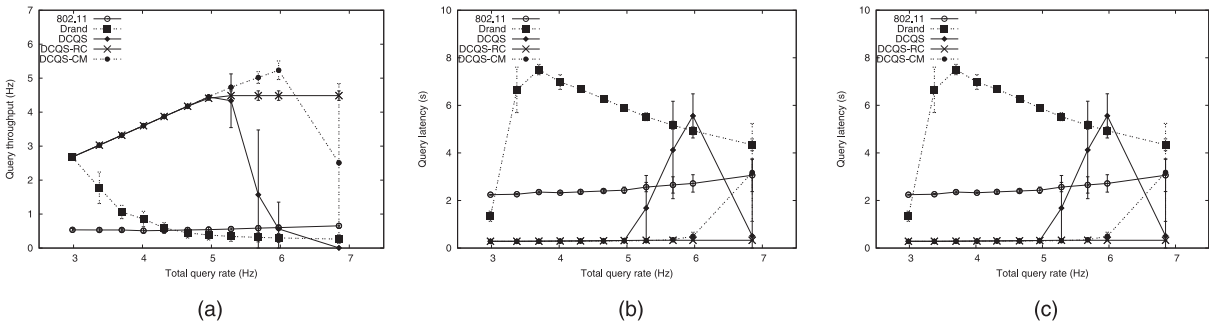


Fig. 12. Performance comparison when three queries belonging to different query classes. (a) Query throughput. (b) Query fidelity. (c) Query latency.

This validates that DCQS schedules conflict-free transmissions even in the case of multiple query classes.This shows that our algorithms work correctly even in the presence of multiple query classes with involving overlapping subsets of nodes. This shows that our algorithms work correctly even in the presence of multiple query classes with involving overlapping subsets of nodes.

Fig. 12c shows the average query latencies. When the total query rate is 2.98 Hz, DCQS achieves a query latency of 0.28 s compared to DRAND which had a query latency of 1.33 s. This is reduction of over 78 percent in query latency. The significant reduction in latency highlights the of taking into account the precedence constraints between packet transmissions.

6 Related Work

TDMA scheduling is attractive for high data rate sensor networks because it is energy-efficient and may provide higher throughput than CSMA/CA protocols under heavy load. Two types of TDMA scheduling problems have been investigated in the literature: node scheduling and link scheduling. In node scheduling, the scheduler assigns slots to nodes whereas, in link scheduling, the scheduler assigns slots to links through which pairs of nodes communicate. In contrast to earlier work, DCQS adopts a novel approach which we call *query scheduling*. Instead of assigning slots to each node or link, we assign slots to transmissions based on the specific communication patterns and temporal properties of queries in WSNs. This allows DCQS to achieve high throughput and low latency.

Early TDMA scheduling protocols were designed for static or uniform workloads [20], [21], [22], [23]. Such approaches are not suitable for dynamic applications with variable and nonuniform workloads. Several recent TDMA protocols can adapt to changes in workload. A common method to handle variable workloads is to have nodes periodically exchange traffic statistics and then adjust the TDMA schedule based on the observed workload [20], [24], [25]. However, exchanging traffic statistics frequently may introduce nonnegligible communication overhead. In contrast, DCQS can efficiently adapt to changes in workloads by exploiting explicit query information provided by the query service. Furthermore, it features a local scheduling algorithm that can accommodate changes in query rates and additions/deletions of queries without explicitly reconstructing the schedule.

To combine the benefits of CSMA and TDMA, several hybrid protocols have been proposed [16], [26]. For example, ZMAC [16] constructs a transmission schedule for each node but it allows nodes to contend for access to other nodes slots using channel polling. Similarly, Funneling-MAC [26], a hybrid protocol designed for data collection, constructs a TDMA schedule that involves only the nodes within a few hops of the sink (where high contention occurs) while the remainder of the nodes use a CSMA protocol. As result, the overhead of maintaining a multihop TDMA schedule is reduced. As an efficient TDMA scheduling algorithm, DCQS may be integrated with existing hybrid schemes [16] [26].

Recently, several theoretical bounds for wireless networks have been derived [27], [28], [29]. These bounds provided important insight on the fundamental limits of wireless networks. However, they cannot be directly applied in practice because they are derived based on ideal assumptions. In contrast, in this paper, we derive a tight bound on the maximum query rate achieved under DCQS. Such a bound is of practical importance since it can be used to prevent network overload.

7 Conclusions

This paper presents DCQS, a novel transmission scheduling technique specifically designed for query services in wireless sensor networks. DCQS features a planner and a scheduler. The planner reduces query latency by constructing transmission plans based on the precedence constraints in in-network aggregation. The scheduler improves throughput by overlapping the transmissions of multiple query instances concurrently while enforcing a conflict-free schedule. Our simulation results show that DCQS achieves query completion rates at least 62 percent higher than DRAND, and query latencies at least 73 percent lower than DRAND. Furthermore, we derive the theoretical capacity bounds that may be used to prevent network overhead through rate control.

Acknowledgments

This work is funded by the US National Science Foundation under ITR grants CCR-0325529 and CCR-0325197, and under NeTS-NOSS grant CNS-0627126.

References

[1] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," *Proc. First ACM Int'l Workshop Wireless Sensor Networks and Applications (WSNA)*, 2002.

[2] K. Chintalapudi, J. Paek, O. Gnawali, T. Fu, K. Dantu, J. Caffrey, R. Govindan, and E. Johnson, "Structural Damage Detection and Localization Using NETSHM," *Proc. Fifth Int'l Conf. Information Processing in Sensor Networks (IPSN),* 2006.

[3] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis, "Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea," *Proc. Third Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2005.

[4] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TAG: A Tiny Aggregation Service for Ad-Hoc Sensor Networks," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI),* 2002.

[5] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Systems,* vol. 30, no. 1, pp. 122-173, 2005.

[6] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A Unifying Link Abstraction for Wireless Sensor Networks," *Proc. Third Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2005.

[7] N. Kimura and S. Latifi, "A Survey on Data Compression in Wireless Sensor Networks," *Proc. Int'l Conf. Information Technology: Coding and Computing (ITCC),* 2005.

[8] G. Zhou, T. He, J.A. Stankovic, and T.F. Abdelzaher, "RID: Radio Interference Detection in Wireless Sensor Networks," *Proc. IEEE INFOCOM,* 2005.

[9] R. Maheshwari, S. Jain, and S.R. Das, "A Measurement Study of Interference Modeling and Scheduling in Low-Power Wireless Networks," *Proc. Sixth Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2008.

[10] D. Son, B. Krishnamachari, and J. Heidemann, "Experimental Study of Concurrent Transmission in Wireless Sensor Networks," *Proc. Fourth Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2006.

[11] R. Maheshwari, J. Cao, and S.R. Das, "Physical Interference Modeling for Transmission Scheduling on Commodity WiFi Hardware," *Proc. IEEE INFOCOM,* 2009.

[12] G.X. Mo Sha, G. Zhou, S. Liu, and X. Wang, "C-MAC: Model-Driven Concurrent Medium Access Control for Wireless Sensor Networks," *Proc. IEEE INFOCOM,* 2009.

[13] I. Chlamtac and A. Farago, "Making Transmission Schedules Immune to Topology Changes in Multi-Hop Packet Radio Networks," *IEEE/ACM Trans. Networking,* vol. 2, no. 1, pp. 23-29, Feb. 1994.

[14] J.-H. Ju and V.O.K. Li, "An Optimal Topology-transparent Scheduling Method in Multihop Packet Radio Networks," *IEEE/ACM Trans. Networking,* vol. 6, no. 3, pp. 298-306, June 1998.

[15] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, "The Flooding Time Synchronization Protocol," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2004.

[16] I. Rhee, A. Warrier, M. Aia, and J. Min, "Z-MAC: A Hybrid MAC for Wireless Sensor Networks," *Proc. Third Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2005.

[17] J.P. Lynch and K.J. Loh, "A Summary Review of Wireless Sensors and Sensor Networks for Structural Health Monitoring," *The Shock and Vibration Digest,* vol. 38, no. 2, pp. 91-128, 2006.

[18] *IEEE Standard 802.11, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,* IEEE, 1999.

[19] I. Rhee, A. Warrior, J. Min, and L. Xu, "DRAND: Distributed Randomized TDMA Scheduling for Wireless Ad Hoc Networks," *Proc. ACM MobiHoc,* 2006.

[20] I. Cidon and M. Sidi, "Distributed Assignment Algorithms for Multihop Packet Radio Networks," *IEEE Trans. Computers,* vol. 38, no. 10, pp. 1353-1361, Oct. 1989.

[21] A. Ephremides and T. Truong, "Scheduling Broadcasts in Multihop Radio Networks," *IEEE Trans. Comm.,* vol. 38, no. 4, pp. 456-460, Apr. 1990.

[22] R. Ramaswami and K.K. Parhi, "Distributed Scheduling of Broadcasts in a Radio Network," *Proc. IEEE INFOCOM,* 1989.

[23] E. Arikan, "Some Complexity Results about Packet Radio Networks," NASA STI/Recon Technical Report N, vol. 83, Mar. 1983.

[24] V. Rajendran, K. Obraczka, and J.J. Garcia-Luna-Aceves, "Energy-Efficient Collision-Free Medium Access Control for Wireless Sensor Networks," *Proc. First Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2003.

[25] L. Bao and J.J. Garcia-Luna-Aceves, "A New Approach to Channel Access Scheduling for Ad Hoc Networks," *Proc. ACM MobiCom,* 2001.

[26] G.-S. Ahn, S.G. Hong, E. Miluzzo, A.T. Campbell, and F. Cuomo, "Funneling-MAC: A Localized, Sink-Oriented MAC for Boosting Fidelity in Sensor Networks," *Proc. Fourth Int'l Conf. Embedded Networked Sensor Systems (SenSys),* 2006.

[27] P. Gupta and P.R. Kumar, "The Capacity of Wireless Networks," *IEEE Trans. Information Theory,* vol. 46, no. 2, pp. 388-404, Mar. 2000.

[28] V.S.A. Kumar, M.V. Marathe, S. Parthasarathy, and A. Srinivasan, "Algorithmic Aspects of Capacity in Wireless Networks," *Proc. ACM SIGMETRICS,* 2005.

[29] T.F. Abdelzaher, S. Prabh, and R. Kiran, "On Real-Time Capacity Limits of Multihop Wireless Sensor Networks," *Proc. 25th IEEE Int'l Real-Time Systems Symp. (RTSS),* 2004.

Octav Chipara received the PhD degree from Washington University in St. Louis. He is a postdoctoral fellow at the University of California, San Diego. His research interests include real-time embedded systems, wireless sensor networks, and cyber-physical applications. His current research involves developing systems, protocols, and deployment tools for supporting wireless technologies for medical applications.

Chenyang Lu received the MS degree from the Institute of Software, Chinese Academy of Sciences, in 1997 and the PhD degree from the University of Virginia in 2001. He is an associate professor of computer science and engineering at Washington University in St. Louis. He is a coauthor of more than 100 publications and received the US National Science Foundation CAREER Award in 2005. His research interests include real-time embedded systems, wireless sensor networks, and cyber-physical systems. He is a member of the IEEE and the IEEE Computer Society.

John A. Stankovic received the PhD degree from Brown University. He is the BP America professor in the Computer Science Department at the University of Virginia. He won the IEEE Real-Time Systems Technical Committee's award for Outstanding Technical Contributions and Leadership. He is ranked among the top 250 highly cited authors in computer science by Thomson Scientific Institute. He is a fellow of the IEEE and the IEEE Computer Society.

Catalin-Gruia Roman is the Harold B. and Adelaide G. Welge professor of computer science in the Department of Computer Science and Engineering at Washington University in St. Louis. He has an established research career with numerous published papers in a multiplicity of computer science areas including mobile computing, formal design methods, formal languages, biomedical simulation, computer graphics, and distributed databases. His current research involves the study of formal models, algorithms, design methods, and middleware for mobile computing and sensor networks. He is a member of the IEEE.