Real-Time Query Scheduling for Wireless Sensor Networks

Octav Chipara, Chenyang Lu, *Member, IEEE,* Gruia-Catalin Roman, *Member, IEEE*

***Abstract*— Recent years have seen the emergence of wireless cyber-physical systems that must support real-time queries of physical environments through wireless sensor networks. This paper proposes Real-Time Query Scheduling (RTQS), a novel approach to conflict-free transmission scheduling for real-time queries in wireless sensor networks. First, we show that there is an inherent trade-off between latency and real-time capacity in query scheduling. We then present three new real-time schedulers. The non-preemptive query scheduler supports high real-time capacity but cannot provide low response times to high priority queries due to priority inversions. The preemptive query scheduler eliminates priority inversions at the cost of reduced capacity. The slack stealing query scheduler combines the benefits of the preemptive and non-preemptive schedulers to improve the capacity while meeting query deadlines. We provide schedulability analysis for each scheduler. The analysis and advantages of our approach are validated through NS2 simulations.**

***Index Terms*— Query scheduling, schedulability analysis, sensor networks**

I. Introduction

Recent years have seen the emergence of wireless cyber-physical systems that must support real-time data collection at high data rates over wireless sensor networks (WSNs). Representative examples include emergency response [1], structural health monitoring [2], [3], and process measurement and control [4]. Such systems pose significant challenges. First, the system must handle multiple traffic classes with different deadlines. For example, during an earthquake, acceleration data from sensors mounted on a building must be delivered to the base station in a timely manner to detect any structural damage. Such traffic should have higher priority than temperature data collected for climate control. Thus, a WSN protocol should provide *effective prioritization* between different traffic classes while meeting their respective deadlines. Second, the system must support *high throughput* since sensors may generate a high workload. For example, structural health monitoring requires numerous accelerometers to be sampled at high rates generating high network loads [2]. Furthermore, since the system must deliver data to base stations within their deadlines, it is important for the system to achieve *predictable and bounded end-to-end latencies*.

Many cyber-physical systems use query services to periodically collect data from sensors to a base station over multi-hop wireless networks. In this paper, we propose ***R**eal-**T**ime **Q**uery **S**cheduling* (RTQS), a transmission scheduling approach for real-time queries in WSNs. RTQS includes three real-time query schedulers that exploit the unique characteristics of WSN queries including many-to-one communication and periodic sampling. We provide upper bounds on the response times of real-time queries for each scheduler. A unique aspect of our approach is that it *bridges the gap between wireless sensor networks and schedulability analysis techniques* which have traditionally been applied to real-time processor scheduling.

This paper makes four contributions: First, we show that real-time query scheduling has an inherent tradeoff between latency and real-time capacity. Second, we developed three schedulers: (1) The *nonpreemptive* query scheduler achieves high capacity but cannot provide low response times to high priority queries due to priority inversions. (2) The *preemptive* query scheduler eliminates priority inversions at the cost of reduced capacity. (3) The *slack stealing* scheduler combines the advantages of preemptive and non-preemptive schedulers to improve real-time capacity while meeting query deadlines. Third, we provide a schedulability analysis for each scheduler that enables predictable real-time query services through online admission and rate control. Finally, we demonstrate demonstrate the advantages of RTQS over existing contention-based and TDMA-based protocols in terms of both real-time performance through simulations.

The paper is organized as follows. Section II reviews the related work. Section III describes the query and network models. Section IV and Section V detail the design and analysis of RTQS. Section VII provides simulation results. Section VIII concludes the paper.

II. Related Work

Real-time communication protocols adopt contention-based or TDMA-based approaches. Contention-based protocols support real-time communication through probabilistic differentiation. This is usually achieved by adapting the contention window and/or the initial back-off of the CSMA/CA mechanism. Overviews of these approaches are presented in [5] and [6]. Rate and admission control [7], [8], [9] may be used with contention-based protocols to handle congestion. However, these approaches are unsuitable for high data rate and real-time application because (1) they provide highly variable communication latencies due to their random back-off mechanisms and (2) achieve low throughput under heavy load to excessive channel contention.

In multi-hop networks, real-time communication may also be achieved through real-time routing [10], [11]. For example, the SPEED protocol [10] builds on geographic routing to deliver packets at an uniform velocity in a multi-hop network. SPEED was extended to support multiple delivery speeds through multi-path routing [11] . These protocols usually build upon contention based MAC protocols and, as a result, inherit their drawbacks.

TDMA protocols can achieve predictable communication latencies and higher throughput than contention-based protocols under heavy load. Several real-time TDMA protocols were designed

O. Chipara is with the Department of Computer Science, University of Iowa

C. Lu is with the Computer Science and Engineering, Washington University in St. Louis

G-C. Roman is with the Department of Computer Science, University of New Mexico

Digital Object Indentifier 10.1109/TC.2012.172

Symbol	Description
l, m, and h	Indices for low/medium/high priority queries
P_l	Period of query l
D_l	Deadline of query l
R_l	Response time of query l
$\Gamma_l[i]$	Workload of query l on node i
ϕ_l	Start time of query l
u, v	Instance number of a query
$I_{l,u}$	The u^{th} instance of query l
$r_{l,u}$	Release time of instance $I_{l,u}$
L_l	Length of plan of query l
$T_l[i]$	Transmissions in step i in the plan of query l
T_l and V_l	The actual and reversed plans of query l
$I_{l,u}.i$	Step number i in the plan of instance (l, u)
$\overrightarrow{ab}$	Directed edge from a to b

Fig. 1. Table of symbols

for single-hop networks. The IEEE 802.15.4 standard specifies Guaranteed Time Slots (GTS) for achieving predictable delays in single hop networks. A flexible slot reservation mechanism was proposed [12] where slots are allocated based on delay or bandwidth requirements.

Several protocols aim at supporting real-time communication in multi-hop networks. Two papers proposed real-time transmission scheduling for robots [13], [14]. Both protocols assume that at least one robot has complete knowledge of the robots' positions and/or network topology. While these protocols may work well for small teams of robots, they are not suitable for queries in multi-hop WSNs. Implicit EDF [15] provides prioritization in a single-hop cell. The protocol supports multi-hop communication by assigning different frequencies to cells with potential conflicts. However, the protocol does not provide prioritization for transmitting packets across cells. In contrast, RTQS provides prioritization in multi-hop networks without requiring multiple frequencies.

The adoption of the WirelessHART standard has renewed the interest in real-time communication. A number of scheduling protocols have been proposed for effectively scheduling packet transmissions under the WirelessHART model [16], [17], [18]. These solutions adopt a centralized approach to the construction of transmission schedules and do not support spatial reuse. As a result, the scalability of such approaches is limited. In contrast, RTQS overcomes these limitations by supporting spatial reuse and provides decentralized query schedulers.

In earlier work we proposed DCQS [19], a transmission scheduling technique for WSN queries. In contrast to traditional TDMA protocols designed to support general workloads, DCQS is specifically designed to exploit specific communication patterns and temporal properties of queries in WSNs. This allows DCQS to achieve high throughput and low latency. However, DCQS does not support query prioritization or real-time communication, which is the focus of this paper.

III. System Models

In this section, we characterize the query services for which RTQS is designed and describe the network model. As we introduce notations, we will be summarized them in Fig. 1.

A. Query Model

RTQS supports queries in which source nodes produce data reports periodically. This model fits many cyber-physical systems that gather data from the environment at user specified rates. A query l is characterized by the following parameters: a set of sources, the start time ϕ_l, the period P_l, the deadline D_l, a static priority, and an in-network aggregation function [20]. The primary focus of RTQS is to support the efficient execution of real-time queries that involve data aggregation. This design is motivated by the common use of data aggregation functions such as packet merging [21], data compression [22], or statistical functions (e.g., average, histogram [20]) in WSNs. When the user does not specify an aggregation function, RTQS will use packet merging as the default aggregation function to reduce communication workloads and energy consumption.

A new *query instance* is released in the beginning of each period to gather data from the WSN. We use $I_{l,u}$ to refer to the u^{th} instance of query l whose release time is $r_{l,u} = \phi_l + u \cdot P_l$. For brevity, in the remainder of the paper we will refer to a query instance simply as an *instance*. The priority of an instance is the priority of its query. If two instances have the same priority, the instance with the earliest release time has higher priority. For each query a node i needs $\Gamma_l[i]$ slots to transmit its (aggregated) data report to its parent.

The performance of a query service is characterized by two parameters: throughput and real-time capacity. The throughput of a query service is $\sum_l \frac{1}{P_l}$. The real-time capacity of a query service is the maximum throughput for which the query service does not drop packets and meets the deadlines of all the queries. Henceforth, we refer to real-time capacity simply as the capacity.

A query service works as follows: a user issues a query to a sensor network through a base station, which disseminates the query parameters to all nodes [20]. The query service maintains a *routing tree* rooted at the base station. To facilitate data aggregation each non-leaf 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. During the lifetime of the application the user may issue new queries, delete queries, or change the period or priority of existing queries. RTQS is designed to support such dynamics efficiently.

B. Network Model

RTQS models a WSN as an Interference-Communication (IC) graph. The IC graph, IC(E,V), has all nodes as vertices (V) and has two types of directed edges (E): *communication* and *interference* edges. A *communication edge* $\overrightarrow{ab}$ indicates that a packet transmitted by a may be received by b. A subset of the communication edges forms the routing tree used for data aggregation. 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. The IC graph is used to determine if two transmissions can be scheduled concurrently. We say that two transmissions, $\overrightarrow{ab}$ and $\overrightarrow{cd}$ are *conflict-free* ($\overrightarrow{ab} \parallel \overrightarrow{cd}$) and can be scheduled concurrently if (1) a, b, c, and d are distinct and (2) $\overrightarrow{ad}$ and $\overrightarrow{cb}$ are not communication/interference edges in E.

The IC graph accounts for link asymmetry and irregular communication and interference ranges observed in WSN [23]. The IC graph is computed and stored in a distributed fashion: a node *only* knows its incoming/outgoing communication and interference edges. The RID protocol proposed by Zhou et al. is a practical solution for constructing the IC graphs [23].

IV. Real-time Query Scheduling

RTQS provides predictable and differentiated query latencies through prioritized conflict-free transmission scheduling. Our

approach relies on two components: a *planner* and a *scheduler*. The planner constructs a *plan* for executing all the instances of a query. A plan is an ordered sequence of *steps*, each comprised of a set of conflict-free transmissions. RTQS employs the same distributed algorithm as DCQS to construct plans. The scheduler that runs on every node and determines the time slot in which each step of a plan is executed. To improve the capacity, the scheduler may execute steps from multiple instances in the same slot as long as they do not conflict.

RTQS works as follows: (1) When a query is submitted, RTQS identifies a plan for its execution. RTQS usually reuses a previously constructed plan for the new query as multiple queries may be executed using the same plan (see Section IV-A). When no plan may be reused, the planner constructs a new one using the distributed planner. (2) RTQS determines if a query meets its deadline using our schedulability analysis. The schedulability analysis is performed on the base station using information collected during the distributed plan construction. If the query is schedulable, the parameters of the query are disseminated; otherwise, the query is rejected. Note that plans are cached even when the schedulability analysis fails. (3) At run-time the scheduler running on each node executes all admitted queries in a distributed fashion.

In contrast to DCQS, the key contribution of RTQS is the design and analysis of three *real-time* query schedulers. Each scheduler achieves a different tradeoff between query latency and capacity. *Nonpreemptive Query Scheduler* (NQS) supports a high capacity but at the cost of priority inversion, while *Preemptive Query Scheduler* (PQS) eliminates priority inversions at the cost of lower capacity. *Slack-stealing Query Scheduler* (SQS) combines the benefits of NQS and PQS by improving capacity while meeting all deadlines.

A. Constructing plans

Plan Properties: A plan has two properties: (1) Each node is assigned sufficient steps to transmit its entire data report. We use $T_l[i]$ to denote the set of transmissions assigned to step i ($0 \leq i < L_l$) in the plan of query l, where L_l is the length of the plan. (2) The plan also must respect the precedence constraints introduced by aggregation: a node is assigned to transmit in a later step than any of its children.

We remind the reader that RTQS will use packet merging as its default data aggregation function. This design decision is a trade-off between query latency and energy consumption. The enforcement of the precedence constraints introduced by packet merging may increase query latency (i.e., length of the plan) as shorter plans may be constructed when these constraints are not enforced. However, the use of packet merging has two key advantages: (1) it reduces the communication workload and (2) the constructed transmission schedules 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.

Distributed Planner: Since a node waits to receive the data reports from its children, the planner may reduce the query latency by assigning the transmissions of a node with a larger depth in the routing tree to an earlier step of the plan. This strategy reduces the query latency because it reduces the time a node waits for the data reports from all its children. More sophisticated heuristics may be developed. For example, we may account for the case when the routing tree is unbalanced to construct shorter plans. A difficulty associated with this approach is that the protocol would have to keep track of the size of the subtrees as the topology changes. In contrast, our goal is to develop a heuristic that performs well in realistic scenarios (see Section VII) and allows for a simple distributed implementation on resource constrained WSN nodes rather than focusing on the construction of optimal schedules.

A detailed description of the planner may be found in [19]; here, we overview the distributed planner and analyze its communication complexity. The planner works in two stages. In the first stage the planner constructs a *reversed* plan (V) 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) by reversing the order of the steps to enforce the precedence constraints.

The planner associates with each node a priority given by the triple *(depth, child count, ID)* with the root having the highest priority. A node n waits until it is the highest priority and unscheduled node within its one-hop neighborhood. When this occurs, n collects the local plans of its two hop neighbors that have higher priority. Using this information, n determines the steps in which its children will transmit: if n transmits (to its parent) in step t, each of its children will be assigned in the first step after t that does not conflict with the transmissions of the higher priority nodes. A child is assigned in sufficient steps to meet its workload demand. The first stage completes with node n disseminating its local plan to its two hop neighbors.

The second stage involves the reversal of the plan which requires nodes to know the length of the plan. This is accomplished in two steps. By definition, a plan's length is the maximum step number in which a node transmits. This value may be computed at the base station using standard data aggregation with $\max$ as the aggregation function. Next, the plan's length is disseminate to all nodes by taking advantage of the already established routing tree. Once the value of the plan's length is received by a node, the reversal of the plan is a local operation. Upon the completion of the distributed planning process, each node in the network stores the time step in which it and its children transmit as well as the length of the plan. Note that the base station does not have a global view of the network but rather the plans are stored in a distributed manner.

Fig. 2 shows an IC graph and the actual and reversed plans constructed by the planner. The solid lines indicate the communication edges in the routing tree while the dashed lines indicate interference edges. Node a is the base-station. The plan in Fig. 2 is constructed assuming that the data report generated by a node can be transmitted in a single step for each instance. The constructed plan meets the two constraints previously specified: (1) The planner assigns conflict-free transmissions in each step. For example, transmissions $\overrightarrow{ne}$ and $\overrightarrow{po}$ are assigned to step $T_l[1]$ since they do not conflict with each other. (2) The precedence constraints introduced by aggregation are respected. For example, nodes p and q are assigned in earlier steps than their parent o.

The message complexity of the distributed planner is $2(N_2 + N)$, where N_2 is the size of the two-hop neighbors and N is the number of nodes (i.e., $N = |E|$). The first stage requires a node to collect the local plans of its two-hop neighbors and then disseminate its local plan to the same nodes. This contributes a

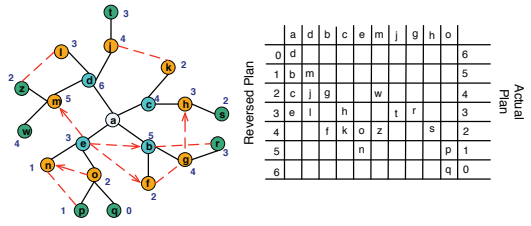


Fig. 2. IC Graph and associated plan

total of $2 \cdot N_2$ messages. The second stage requires each node to transmit two packets, one during the computation of the plan length and the other during its dissemination. This contributes an additional $2 \cdot N$ messages.

The overhead of the distributed planner was evaluated through simulations in [19]. Simulations results indicate that the cost of plan construction dominates the overall message overhead exceeding the cost of constructing the routing tree or the IC graph. More importantly, the simulations show the distributed planner scales well: message complexity increases linearly with the number of nodes when the node density remains constant. This is consistent with the above analysis.

Plan Reuse: Due to the significant cost associated with constructing plans, we must minimize the number of constructed plans. We observed that the plan of a query l depends on the IC graph, the set of source nodes, and the aggregation function. Query instances executed at different times may need different plans if the IC graph changes. However, to handle dynamics in channel conditions, RTQS can construct plans that are robust against certain variations in the IC graph (as discussed in [19]). 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 periods, start times, or priority can be executed according to the same plan. Furthermore, even queries with different aggregation functions may be executed according to the same plan. Let $\Gamma_l[i]$ be the number of slots node i needs to transmit its data report to its parent for an instance of query l. If the planner constructs a plan for a query l, the same plan can be reused to execute a query h if $\Gamma_l[i] = \Gamma_h[i]$ 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., $\Gamma_{max}[i] = \Gamma_{avg}[i] = 1$ for all nodes i) if the slot size is sufficiently large to transmit a packet with 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. We say that two queries belong to the same *query class* if they may be executed according to the same plan. Since queries with different temporal properties and aggregation functions may share a same plan, a WSN may only need to support a small number of query classes. This allows RTQS to amortize the cost of constructing a query plan over many queries and effectively reduces the overhead.

B. Overview of Scheduling Algorithms

RTQS supports both preemptive and nonpreemptive query scheduling. NQS controls only the start of an instance: once an instance *starts* executing, the scheduler cannot preempt it. In contrast, PQS will *preempt* an instance to allow a higher priority instance to execute when the two cannot be executed concurrently. The slack stealing scheduler determines *dynamically whether to preempt* instances. It avoids preemption to improve capacity when it can still meet all deadlines, and performs preemption when needed to meet deadlines.

The scheduler executes a query instance according to the plan of its query. The scheduler improves the capacity by overlapping the transmissions of multiple instances (belonging to one or more queries) such that: (1) All steps executed in a slot are conflict-free. Two steps of instances $I_{l,u}$ and $I_{h,v}$ are *conflict free* ($I_{l,u}.i \parallel I_{h,v}.j$) if all pairs of transmissions in $T_l[I_{l,u}.i] \cup T_h[I_{h,v}.j]$ are conflict free. (2) The steps of a plan are executed in order: if step $I_{l,u}.i$ is executed in slot s_i, step $I_{h,v}.j$ is executed in slot $s_j < s_i$ then $I_{h,v}.j < I_{l,u}.i$. This ensures that the precedence constraints required by aggregation are preserved. The local scheduler running on a node maintains a record of the start time, period, and priority of admitted queries. Plans are stored distributedly: each node knows when it transmits/receives packets and the plan's length.

The following is a brute-force approach for constructing a preemptive scheduler: in every slot s, a brute-force scheduler would consider the released instances in order of their priority and execute all steps that do not conflict in s. This solution has a high time complexity since each pair of steps must be checked for conflicts. We are interested in schedulers that have low time complexity as they will determine the steps to be in executed in a slot dynamically. To reduce the time complexity of the scheduler we introduced the concept of *minimum step distance* in [19] (originally named the minimum inter-release time). Let $I_{l,u}.i$ and $I_{h,v}.j$ be two steps in the plans of instances $I_{l,u}$ and $I_{h,v}$. We define the *step distance* between $I_{l,u}.i$ and $I_{h,v}.j$ as $|I_{l,u}.i - I_{h,v}.j|$. The *minimum step distance* $\Delta(l,h)$ is the smallest step distance between $I_{l,u}$ and $I_{h,v}$ such that two steps $I_{l,u}.i$ and $I_{h,v}.j$ may be executed concurrently without conflict:

$$|I_{l,u}.i - I_{h,v}.j| \geq \Delta(l,h) \Rightarrow I_{l,u}.i \parallel I_{h,v}.j$$
$$\forall I_{l,u}.i < L_l, I_{h,v}.j < L_h$$

L_l and L_h are the plan lengths of queries l and h. Thus, to ensure that no conflicting transmissions are executed in a slot, it suffices to enforce a minimum step distance between any two steps.

The minimum step distance captures the degree of parallelism that may be achieved due to spatial reuse in a multi-hop WSN. Consider the case when $L = L_q = L_h$. In the worst case, when $\Delta(l,h) = L$, a single instance is executed in the network at a time. If $\Delta(l,h) = L/2$, then two instances can be executed in the network at the same time. A distributed algorithm for computing $\Delta(l,h)$ is presented in [19]. The minimum step distance $\Delta(l,h)$ depends on the IC graph and the plans of l and h. The number of minimum step distances that a scheduler stores is quadratic in the number of plans. Two pairs of queries (l,h) and (m,n) have the same minimum step distance if (l,m) and (h,n) have the same plan. Therefore, since a small number of plans is sufficient to meet the communication requirements of an application, then the number of step distances that must be stored is also small.

For clarity, we first present the scheduling algorithms assuming that all queries are executed according to a single plan of length L in this section. In this case, the scheduler maintains a single minimum step distance Δ. We extend our algorithms to handle queries with different plans in the next section.

C. Nonpreemptive Query Scheduler (NQS)

To efficiently enforce the minimum step distance for NQS, we take advantage of the fact that once an instance is started, it cannot be preempted. As such, the earliest time at which an instance $I_{l,u}$ may start (i.e., execute step $I_{l,u}.i = 0$) is after the previous instance $I_{h,v}$ completes step $I_{h,v}.j = \Delta - 1$ (since $|\Delta - 0| \geq \Delta$). Since the execution of $I_{l,u}$ and $I_{h,v}$ cannot be preempted, if we enforce the minimum step distance between the start of the two instances then their concurrent execution is conflict-free for their remaining steps since steps $I_{l,u}.i = x$ and $I_{h,v}.j = x + \Delta$ are executed in the same slot and $|(x + \Delta) - x| \geq \Delta$. Therefore, to guarantee that a nonpreemptive scheduler executes conflict-free transmissions in each slot, it suffices to enforce a minimum step distance of Δ between the start times of any two instances.

NQS maintains two queues: a *run* queue and a *release* queue. The *release* queue is a priority queue containing all instances that have been released but are not being executed. The *run* queue is a FIFO queue and 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 transmissions). A node n determines if it transmits/receives in slot s by checking if it is assigned to transmit/receive in any of the steps to be executed in slot s.

NQS enforces a minimum step distance of at least Δ between the start times of any two instances by starting an instance in two cases: (1) when there is no instance being executed (i.e., *run*=$\emptyset$) and (2) when the step distance between the head of the *release* queue (i.e., the highest priority instance that has been released) and the tail of the *run* queue (i.e., the last instance that started) exceeds Δ. An instance is moved from the *release* to the *run* queue when it starts.

Consider the example in Fig. 5(a) where queries Q_h, Q_m and Q_l are executed according to the shown workload parameters. The queries are executed using the same plan of length $L = 15$ and minimum step distance $\Delta = 8$. We assign higher priority to queries with tighter deadlines. The upward arrows indicate the release time of an instance. I_l (in the example we drop the instance count since it is always zero) is released and starts in slot 0 since no other instance is executing (*run*=$\emptyset$). The first instances of Q_m and Q_h are released in slots 2 and 6, respectively. However, neither may start until slot 8 when I_l completes 8 steps resulting in priority inversions. I_h then starts at slot 8 since it is the highest priority instance in *release*. Similarly, in slot 16, NQS starts I_m after I_h completes $\Delta = 8$ steps. NQS continues to construct the schedule in figure.

When a new instance is released, NQS inserts it in the *release* queue. This takes $O(\log |release|)$ time since *release* is a priority queue keyed by the priority of instances. In each slot, NQS determines what instances should start. This operation takes constant time, 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 transmit or receive, NQS iterates through the *run* queue. This requires $O(|run|)$ time if a node maintains a bit vector indicating whether it transmits or receives in each step of a plan. Thus, the complexity of the operations performed in a slot is $O(|run|)$.

```
event: new instance I_{l,u} is released
    if (run=∅) then start(I_{l,u})
    else release = release ∪ {I_{l,u}}
event: start of new slot s
    if (release ≠ ∅)
      let I_{l,u} be the highest priority instance in release
      if (Last_{q',k'}.i ≥ Δ) then
        start(I_{l,u})
        Last_{q',k'} = I_{l,u}
    for each I_{l,u} ∈ run execute-step(I_{l,u})
start(I_{l,u}):
    run = run ∪ {I_{l,u}}
execute-step(I_{l,u}):
    determine if node should send/recv in I_{l,u}.i
    I_{l,u}.i = I_{l,u}.i + 1
    if I_{l,u}.i = L_q then run = run \ {I_{l,u}}
```

Fig. 3. NQS pseudocode

D. Preemptive Query Scheduler (PQS)

A drawback of NQS is that it introduces priority inversions. To eliminate priority inversions, we devised PQS which preempts the instances that conflict with the execution of a higher priority instance. A key feature of PQS is a new and efficient mechanism for enforcing the minimum step distance that supports preemption. To enforce the minimum step distance, PQS maintains L_q *mayConflict* sets. Each *mayConflict*$[x]$ set contains the instances which are in the *run* queue and conflict with *any* instance executing step x in its plan:

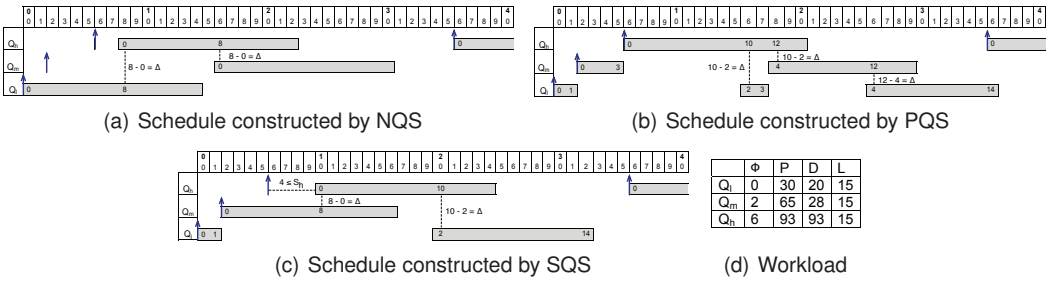
$$\textit{mayConflict}[x] = \{I_{h,v} | I_{h,v} \in \textit{run and } |x - I_{h,v}.i| < \Delta\}$$

PQS (see Fig. 4) maintains a *run* and a *release* queue keyed by the priority of instances. When a new instance is released, it is added to the *release* queue.

PQS starts/resumes an instance $I_{l,u}$ ($I_{l,u} \in$ *release*) in two cases: (1) If the next step $I_{l,u}.i$ may be executed concurrently with all instances in the *run* queue without conflict, PQS starts/resumes it. To determine if this is the case, it suffices for PQS to check if *mayConflict*$[I_{l,u}.i]$ is empty. When an instance is started or resumed, it is moved from the *release* to the *run*. The membership of $I_{l,u}$ in the *mayConflict* sets is updated to reflect that $I_{l,u}$ is executed in the current slot: $I_{l,u}$ is added to all *mayConflict*$[x]$ sets such that $|I_{l,u}.i - x| < \Delta$ since the execution of any of those steps would conflict with the execution of step $I_{l,u}.i$. (2) $I_{l,u}$ is also started/resumed if it has higher priority than all the instances in *mayConflict*$[I_{l,u}.i]$ since otherwise there will be a priority inversion. For $I_{l,u}$ to be executed without conflict, all instances in *mayConflict*$[I_{l,u}.i]$ must be preempted. When an instance is preempted, it is moved from the *run* to the *release* and it is removed from all *mayConflict* sets. As in case (1), $I_{l,u}$ is added to all *mayConflict*$[x]$ sets such that $|I_{l,u}.i - x| < \Delta$.

After an instance executes a step, its membership in the *mayConflict* sets must also be updated. Since step $I_{l,u}.i$ is executed in slot s, in the next slot (when $I_{l,u}$ executes step $I_{l,u}.i + 1$) $I_{l,u}$ will not conflict with an instance executing step $I_{l,u}.i - \Delta$ but will conflict with an instance executing step $I_{l,u}.i + \Delta$. Accordingly, $I_{l,u}$ is removed from *mayConflict*$[I_{l,u}.i - \Delta]$ and added to *mayConflict*$[I_{l,u}.i + \Delta]$.

Fig. 5(b) shows the schedule of PQS. I_l starts in slot 0 since no other instances have been released (*mayConflict*$[0] = \emptyset$). I_m is released in slot 2. Since *mayConflict*$[0] = \{I_l\}$ and I_m has higher priority than I_l, PQS preempts I_l. Consequently, I_l is removed from the *run* queue and all *mayConflict* sets, and it is



	Φ	P	D	L
Q_l	0	30	20	15
Q_m	2	65	28	15
Q_h	6	93	93	15

Fig. 5. Schedule examples for NQS, PQS, and SQS (single-class case). Workload described in Fig. 5(d)

event: new instance $I_{l,u}$ is released
  release = *release* $\cup \{I_{l,u}\}$
event: start of new slot s
  for each $I_{l,u} \in$ *release*
    if (**may-resume**($I_{l,u}$) = true) then **resume**($I_{l,u}$)
  for each $I_{l,u} \in$ *run*
    execute-step($I_{l,u}$)
resume($I_{l,u}$):
  run = *run* $\cup \{I_{l,u}\}$; *release* = *release* $- \{I_{l,u}\}$
  add $I_{l,u}$ to all mayConflict[x] such that $|I_{l,u}.i - x| < \Delta$
preempt(S):
  run = *run* $- S$; *release* = *release* $\cup S$
  remove $I_{l,u}$ from all mayConflict
may-resume($I_{l,u}$):
  if (mayConflict[$I_{l,u}.i$] $= \emptyset$) then return true
  if ($I_{l,u}$ has higher priority all instances in mayConflict[$I_{l,u}.i$])
    preempt(mayConflict[$I_{l,u}.i$]); return true
  return false
execute-step($I_{l,u}$):
  determine if node should send/recv in $I_{l,u}.i$
  $I_{l,u}.i = I_{l,u}.i + 1$
  if $I_{l,u}.i = L$ then *run* = *run* $- \{I_{l,u}\}$
  mayConflict[$I_{l,u}.i - \Delta$]=mayConflict[$I_{l,u}.i - \Delta$] $- \{I_{l,u}\}$
  mayConflict[$I_{l,u}.i + \Delta$]=mayConflict[$I_{l,u}.i + \Delta$] $\cup \{I_{l,u}\}$

Fig. 4. PQS pseudocode

added to the *release* queue. I_m is added to *run* queue and to all *mayConflict*[x] sets where $0 \leq x < 8$. I_h is released in slot 6. Since *mayConflict*[0] $= \{I_m\}$ and I_h has higher priority then I_m, PQS preempts I_m and starts I_h. The *mayConflict* sets are updated accordingly. An interesting case occurs in slot 16, when I_h executes step 10. At this point, *mayConflict*[2] $= \emptyset$ since I_m was preempted and I_h completed 10 steps ($|10 - 2| \geq 8$). As a result, I_l may execute step 2 in its plan while I_h executes step 10 without conflict. I_h and I_l are executed concurrently until step 18 because their step distance exceeds the minimum step distance. In the beginning of slot 18, *mayConflict*[4]=$\{I_l\}$. Note that I_h is not a member of this set since $|12 - 4| \geq 8$. Since the step counter of I_m is 4 and I_m has higher priority than I_l, PQS preempts I_l and resumes I_m. PQS then updates the conflict sets by removing I_l from all of them and adding I_m to *mayConflict*[x] sets where $|x-4| < 8$. I_l resumes in slot 26 when *mayConflict*[4] becomes empty. The example shows that by eliminating priority inversion PQS achieves lower latencies for I_h and I_m than NQS. However, the capacity is lower because it allows less overlap in the execution of instances. This shows the tradeoff between latency and capacity in query scheduling. We will characterize this tradeoff analytically in the next section.

When an instance is released, it is added to the *release* queue which takes $O(\log |release|)$ time. In every slot, PQS iterates through the instances in *release* to determine if they may be resumed. If we organize the *mayConflict* sets as balanced trees keyed by instance priority, the time complexity of this operation is $O(|release| \cdot \log |run|)$. We note that the complexity of **resume** and **preempt** is at most L. Similar to NQS, $O(|run|)$ is necessary for a node to determine if it transmits, receives, or sleeps in a slot. Thus, the time complexity of operations performed per slot is $O(|release| \cdot \log |run| \cdot L + |run|)$.

E. Analysis of NQS and PQS

Next, we present the worst-case response analyses of PQS and NQS. The response time of a query is the maximum query latency of its instances. Our analysis can be used for admission and rate control at the base station when a query is submitted. We assume that the deadlines are shorter than the periods. For convenience we use the slot size as the time unit.

Analysis of NQS. Since NQS is non-preemptive, the response time R_l of query l is the sum of its plan's length L and the worst-case delay W_l that any instance experiences before it is started: $R_l = W_l + L$.

To compute W_l, we construct a recurrent equation similar to the response time analysis for processor scheduling [24]. Consider an instance I_l. Note that for clarity we drop the instance index from the instance notation in our analysis. Since NQS is a nonpreemptive scheduling algorithm, to compute the response time of a query l, we must compute the worst-case interference of higher priority instances and the maximum blocking time of l due to the nonpreemptive execution of lower priority instances. Our analysis is based on the following two properties.

Property 1: An instance is blocked for at most $\Delta - 1$ slots.

Proof: Consider the following two cases based on when an instance I_l is released. (1) If all executing lower priority instances have completed at least Δ steps, NQS starts I_l without blocking. (2) If a lower priority instance which did not complete Δ steps is executing, I_l is blocked. Note that there can be only one lower priority instance that blocks I_l, because the interval between the starting times of two consecutive instances must be at least Δ. Hence, there is only *one* executing instance that has not completed Δ steps when I_l is released. The longest blocking time occurs when the low priority instance has completed one step when I_l is released. In this case I_l is blocked for $\Delta - 1$ slots. ■

Property 2: A higher priority instance interferes with a lower priority instance for at most Δ slots.

Proof: NQS starts the highest priority instance when the last started instance has completed at least Δ steps. Therefore, every high priority instance delays the execution of a low priority instance by at most Δ slots. The worst-case interference occurs when the lower and higher priority instances are released simultaneously. ■

The number of instances of a higher priority query h that interfere with I_l is upper-bounded by $\lceil \frac{W_l}{P_h} \rceil$. Therefore, the worst-case delay that I_l experiences before it starts is:

$$W_l = (\Delta - 1) + \sum_{h \in hp(l)} \left\lceil \frac{W_l}{P_h} \right\rceil \cdot \Delta \qquad (1)$$

where $hp(l)$ is the set of queries with priority higher than or equal to l's priority except itself. W_l can be computed by solving (1) using a fixed point algorithm similar to that of the response time analysis [24].

Note that our analysis differs from the classical processor response time analysis in that multiple transmissions may occur concurrently without conflict in a WSN due to spatial reuse of the wireless channel. This is captured in our analysis in that a higher priority instance may delay a lower priority instance by at most Δ, which is usually smaller than the execution time of the instance (i.e., the plan's length L).

Analysis of PQS. A higher priority instance cannot be blocked by a lower priority instance under PQS. We observe that after an instance completes Δ steps, no newly released instance will interfere with its execution because their step distance would be at least Δ, allowing them to execute concurrently. Therefore, we split I_l into two parts: a preemptable part of length Δ and nonpreemptable part of length $L - \Delta$. Higher priority instances may interfere with I_l only during its preemptable part. Thus, the response time of a query l is the sum of response time of the preemptable part R'_l and the length of the nonpreemptable part: $R_l = L - \Delta + R'_l$.

A query h with higher priority than l interferes with l for at most $\lceil \frac{R'_l}{P_h} \rceil \cdot C_{max}(l,h)$ slots, where $C_{max}(l,h)$ is the worst-case interference of an instance of h on an instance of l. Thus, worst-case response time of the preemptable part of l is:

$$R'_l = \Delta + \sum_{h \in hp(l)} \left\lceil \frac{R'_l}{P_h} \right\rceil \cdot C_{max}(l,h) \quad (2)$$

After finding the worst-case interference, R'_l may be computed by solving (2) using a fixed point algorithm similar to the one used in the response time analysis [24]. Next, we determine the worst-case interference.

Theorem 1: An instance I_l is interfered by a higher priority instance I_h for at most $C_{max}(l,h) = \min(2\Delta, L)$ slots.

Due to space limitations, we refer the reader to [25] for the proof of Theorem 1. The generalization of this theorem to multiple classes case is included in this paper as Theorem 3 and its proof is included in Section V.

Capacity-Latency Tradeoff: It is important to note that prioritization comes at cost: preemption of queries results in reduced capacity. The reduction in capacity is attributed to the difference in the worst-case interference introduced by a high priority instances for the two schedulers. NQS does not use preemption and has a maximum interference (of a higher priority instance) equal to Δ, where $\Delta \leq L$. In contrast, PQS uses preemption and, as a result, the maximum interference increases to $\min(2\Delta,\ L)$, where $\min(2\Delta,\ L) \geq L$. The additional interference in the preemptive case results in a lower degree of concurrency and, hence, lower capacity. This shows the inherent trade-off between latency and capacity in RTQS.

F. Slack Stealing Query Scheduler (SQS)

SQS combines the benefits of NQS and PQS: it improves capacity while meeting all deadlines. SQS is based on the observation that preemption lowers capacity, and hence, it should be used only when necessary for meeting deadlines. We define the *slack* of a query l (S_l) to be the maximum number of slots that an instance of l allows a lower priority instance to execute before preempting it. SQS has two components: an admission algorithm and a scheduling algorithm. The admission algorithm runs on the base station to determine the slack and schedulability of each query when it is issued. The scheduling algorithm executes admitted queries based on their slacks.

SQS Scheduler: SQS may start an instance $I_{h,v}$ in any slot in the interval $[r_{h,v}, r_{h,v}+S_h]$, where S_h is the slack of query h and $r_{h,v}$ is the release time of the v^{th} instance of h. Intuitively, SQS can dynamically determine the best time within the interval to start $I_{h,v}$ such that $I_{h,v}$'s interference on lower priority instances is reduced. Since a lower priority instance $I_{l,u}$ is not interfered by $I_{h,v}$ if $I_{l,u}$ has completed at least Δ steps, SQS postpones the start of the higher priority instance $I_{h,v}$ if the lower priority instance $I_{l,u}$ has completed at least $\Delta - S_h$ steps. An advantage of the slack stealing approach is that it opportunistically avoids preemption and the related capacity reduction when allowed by query deadlines.

SQS requires a minor modification to PQS. Specifically, we change how the release of an instance $I_{h,v}$ is handled. If *mayConflict*[0] is empty, $I_{h,v}$ is released immediately. If SQS determines that all the instances in *mayConflict*[0] have completed at least $\Delta - S_h$ steps, SQS delays $I_{h,v}$ until the lower priority instances complete Δ steps in their plans (i.e., when *mayConflict*[0] becomes empty). All instances whose release is delayed are maintained in a *pending* queue. If $I_{h,v}$ does not have sufficient slack to allow the lower priority instances to complete Δ steps, then SQS (1) preempts all instances in *mayConflict*[0], (2) resumes the highest priority instance in the *release* or *pending* queues (which is not necessarily $I_{h,v}$), and (3) moves all instances from the *pending* queue to the *release* queue.

Fig. 5(c) shows the schedule under SQS with the example workload. Assume that the admission algorithm of SQS determined that Q_h and Q_m have slacks $S_h = 5$ and $S_m = 2$, respectively. I_l is released and starts its execution in slot 0. I_m is released in slot 2. SQS preempts I_l, because even if I_m would be postponed for $S_m = 2$ slots, I_l would not complete $\Delta = 8$ steps. I_h is released in slot 6. SQS decides to continue executing I_m because in $4 \leq S_h$ slots, I_m will complete executing $\Delta = 8$ steps, i.e., SQS avoids preempting I_m by allowing it to steal 4 slots from I_h. SQS uses preemption in slot 2 but not in slot 6. This highlights that SQS can adapt preemption decisions to improve capacity while meeting all deadlines.

Admission Algorithm. The admission algorithm determines the schedulability and slacks of queries. It considers queries in decreasing order of their priorities. For each query, it performs a binary search in $[0, \Delta]$ to find the maximum slack that allows the query to meet its deadline. Note that there is no benefit for a lower priority instance to steal more than Δ slots from a higher priority instance since they may be executed in parallel when their step distance is at least Δ. The admission algorithm tests whether the query can meet its deadline by computing its worst-case response time as a function of the slack. If the query is unschedulable with zero slack, it is rejected; otherwise, it is admitted.

To compute the worst-case response time of a query we split aN instance into two parts: a preemptable part and a nonpreemptable part. Under PQS, the preemptable part is Δ slots. In contrast, under SQS, an instance I_l may steal from a higher priority instance at least $m_l = \min_{x \in hp(l)} S_x$ steps. Thus, the length of the preemptable part is at most $\Delta - m_l$ slots under SQS; the length of the nonpreemptable part is therefore $L - (\Delta - m_l)$ slots. Hence,

the worst-case response time of query l with slack S_l is:

$$R_l(S_l) = L - (\Delta - m_l) + R'_l(S_l) \quad (3)$$

where R' is the worst-case response of time the preemptable part.

Theorem 2: Under SQS, an instance I_l may be interfered by a higher priority instance I_h for at most $C_{max} = \min(2\Delta - m_l, L)$ slots, where $m_l = \min_{x \in hp(l)} S_x$.

Proof: We initially assume $L > 2\Delta - m_l$. Similar to PQS the worst-case interference occurs when a higher priority instance is released during I_l's preemptable part. In this case, I_l either (1) steals slack from one or more higher priority instances or (2) does not steal slack from any higher priority instance.

(1) When I_l steals slack we consider the following two sub-cases depending on whether I_l successfully steals enough slack to complete Δ steps.

(1a) I_l completes Δ steps without being preempted. In this case I_h's interference on I_l is zero.

(1b) Otherwise, I_l is preempted after executing x steps by a higher priority instance I_m (not necessarily I_h). Next, we show that the execution of I_m does not affect I_h's interference on I_l. As a result, it would be sufficient to only consider the case when I_h itself preempts I_l. We note that I_m must have a higher priority than I_h since SQS always resumes the highest priority instance in *release* when an instance is preempted. I_h's interference on I_l is not affected by I_m if neither I_l nor I_h execute while I_m executes its preemptable part (i.e., the relative phasing of I_l and I_h remains the same). I_h cannot execute because it cannot start before I_m completes Δ steps (due to minimum step distance). Note that I_l cannot steal slack from I_m as I_l is in *release*. I_l cannot execute as I_h must be started before I_l resumes (since I_h's next step is 0, I_l's next step is $x > 0$, and hence the step distance between I_m and I_h is higher than that between I_m and I_l). Since, I_h cannot start before I_m completes Δ steps, I_l also cannot start before I_m completes Δ steps.

We now consider the case when I_h is the instance that preempts I_l. Similar to Theorem 1 we consider sub-cases depending on whether I_h is preempted. If I_h is not preempted, according to the proof of Theorem 1, I_h's interference on I_l is $C = \Delta + x$. However, unlike in PQS where $x < \Delta$, for SQS we have a tighter bound on x: $x < \Delta - m_l$. Hence, I_h's interference on I_l is $C_{max} = 2\Delta - m_l$. If I_h is preempted by a higher priority instance, let y be the number of steps I_h has completed before it is preempted. We note that $y < m_l$, since m_l is the smallest slack of any query whose priority is higher or equal to l. Similar to PQS, the worst-case interference in the two cases is: $C(x) = \Delta + \max(x, y)$. However, unlike PQS, we have tighter bounds on x and y: $x < m_l$ and $y < m_l$. Thus, the worst-case interference of I_h on I_l is $C_{max} = 2\Delta - m_l$.

(2) In this case I_l is preempted by I_h. This case is handled similarly to (1b).

Similar to PQS, when $L < 2\Delta - m_l$ the interference cost is reduced L. Therefore the worst-case interference of I_h on I_l is $\min(2\Delta - m_l, L)$. ■

To compute R'_l we must account for the jitter introduced by slack stealing, i.e., a higher priority instance I_h may delay its start by at most S_h. Accordingly, R' is:

$$R'_l(S_l) = (\Delta - m_l) + S_l + \sum_{h \in hp(l)} \left\lceil \frac{R'_l(S_l) + S_h}{P_h} \right\rceil \cdot C_{max}(l, h)$$

where, $\Delta - m_l$ is the maximum length (execution time) of the preemptable part, S_l is the maximum time interval when I_l may be blocked by a lower priority instance due to slack stealing, and $C_{max}(l, h) = \min(2\Delta - m_l, L)$ is the worst-case interference when slack stealing is used.

V. Handling Multiple Classes

So far the algorithms and their analyses were presented under the assumption that queries belong to the same class i.e., they are executed according to the same plan. In this section, we consider the case when there are multiple query classes. We define the *minimum step distance between two queries 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.

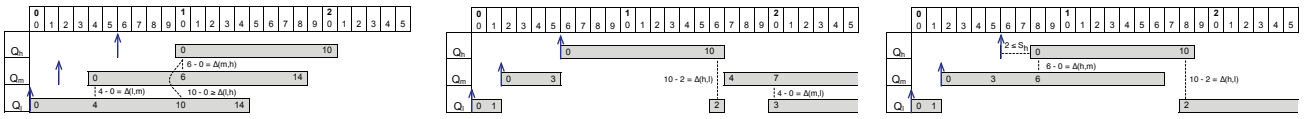
A. Multi-class NQS

When all queries belong to a single query class, NQS only needs to check if the step distance between the highest priority instance in the *release* queue and the instance at the tail of the *run* queue exceeds the minimum step distance to guarantee conflict-free transmissions. However, in the case of multiple classes, to guarantee that *all* minimum step distances are enforced, NQS must check whether the step distance between the highest priority instance in the *release* queue and *all* instances in *run* queue exceeds the minimum step distances between their respective query classes. NQS accomplishes this efficiently by keeping track of the slot when the last instance of each query class started. To enforce *all* minimum step distances it suffices to record the time when the last instance of each class started. Using this information, NQS ensures that any new instance that is released is started only when its step distances to all other instances currently being executed exceeds their respective minimum step distances.

To handle multiple classes, NQS stores the following additional information: (1) for each pair of query classes, NQS maintains the minimum step distances and (2) an integer per query class to keep track of when the last instance of each class started. The number of comparisons necessary to enforce the minimum step distances equals the number of query classes. Therefore, NQS handles multiple classes without increasing its computational complexity 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).

Fig. 6 provides examples of the schedulers executing a multi-class workload consisting of three queries: Q_h, Q_m, and Q_l. The three queries have the same phases, periods, and deadlines as in the single-class examples that we previously considered (see Fig. 5). However, consistent with the differences between the single-class and multi-class cases, query plans may have different lengths and the RTQS schedulers must enforce pair-wise minimum step-distances during query execution. The workload parameters are summarized in Fig. 6(d).

NQS constructs the schedule shown in Fig. 6(a) as follows. In slot 0, the first instance of Q_l is released and NQS starts its execution as no instances are currently being executed. The first instance of Q_m is released in slot 2, however, NQS will not execute it until slot 4 when $\Delta(l, m) = 4$. The first instance of Q_h is released in slot 6. Since NQS is nonpreemptive, it will execute Q_l and Q_m concurrently until slot 10 when the execution of first instance of Q_h may proceed without conflict as *both*



(a) Schedule constructed by NQS (b) Schedule constructed by PQS (c) Schedule constructed by SQS

	Φ	P	D	L
Q_l	0	30	20	15
Q_m	2	65	28	15
Q_h	6	93	93	11

Δ	Q_l	Q_m	Q_h
Q_l	8	4	8
Q_m	4	8	6
Q_h	8	6	8

(d) Workload

Fig. 6. Schedule examples for NQS, PQS, and SQS (multi-class case). Workload described in Fig. 6(d).

$\Delta(m,h) = 6 \leq 6$ and $\Delta(l,h) = 8 \leq 10$. Note that in contrast to the single-class case when NQS required a single comparison to determine when it is safe to start executing a new instance, in the multi-class case, NQS requires $O(c)$ comparisons to ensure that the pair-wise minimum step distances constraints are satisfied.

Schedulability Analysis: For clarity, in the schedulability analysis of NQS and subsequent multi-class schedulers we use interchangeably h and $cls(h)$ in $\Delta(cls(l), cls(h))$ to denote the minimum step distance between the query classes of l and h.

To extend the schedulability analysis of NQS to the multi-class case, we must determine the impact of having multiple classes on the blocking and interference terms (see Property 1 and Property 2, respectively). The maximum blocking time an instance of a query l suffers due to a lower priority instance of a query m is:

$$B_l = \max_{m \in lp(l)} \Delta(m,l) - 1 \tag{4}$$

Proving that the Equation 4 holds follows a similar argument as the proof of Property 1. The worst-case blocking of an instance of query l occurs when a lower priority instance of a query m for which $\Delta(m,l) = B_m + 1$ starts one step before l's instance is released.

The multi-class NQS scheduler starts an instance I_l after all instances I_h previously started complete $\Delta(h,l)$ steps. Thus, the worst-case interference of a higher priority instance I_h on I_l is at most $I_l(h) = \max_{m \in hp(l)} \Delta(h,m)$. Thus, the worst-case response time of a query l is $R_l = L_l + W_l$ where L_l is the length of plan of query class l and W_l is the worst-case delay an instance of l observes before it starts:

$$W_l = B_l + \sum_{h \in hp(l)} \left\lceil \frac{W_l}{P_h} \right\rceil \cdot I_l(h) \tag{5}$$

B. Multi-class PQS

To extend PQS to multiple classes, we need to extend the definition of the *mayConflict* sets. We define *mayConflict*$[x][c]$ to be the set that contains the instances which are in the *run* queue and conflict with *any* instance executing step x in the plan of class c:

mayConflict$[x][c]$=$\{I_{h,v} \in$ *run* $\mid x - I_{h,v}.i < \Delta(h,c)$ and h is started/resumed earlier than any instance of $c\}$

The functions used by PQS (*resume*, *may-resume*, and *execute-step*) need to be updated (see Figure 7). In updating PQS to handle multiple classes, attention must be paid to the *order* of arguments used in the minimum step distances since Δ is not commutative.

According to the definition of the *mayConflict* sets, the multi-class PQS scheduler determines the instances which may interfere

```
event: new instance I_{l,u} is released
    release = release ∪ {I_{l,u}}
event: start of new slot s
    for each I_{l,u} ∈ release
      if (may-resume(I_{l,u}) = true) then resume(I_{l,u})
    for each I_{l,u} ∈ run
      execute-step(I_{l,u})
resume(I_{l,u}):
    run = run ∪ {I_{l,u}}; release = release − {I_{l,u}}
    for each c ∈ C
      add I_{l,u} to all mayConflict[x][c] such that
        |I_{l,u}.i − x| < Δ(l,c)
preempt(S):
    run = run − S; release = release ∪ S
    remove I_{l,u} from all mayConflict sets
may-resume(I_{l,u}):
    if (mayConflict[I_{l,u}.i][cls(l)] = ∅) then return true
    if (I_{l,u} has higher prio. instances in mayConflict[I_{l,u}.i][cls(l)])
      preempt(mayConflict[I_{l,u}.i]); return true
    return false
execute-step(I_{l,u}):
    determine if node should send/recv in I_{l,u}.i
    I_{l,u}.i = I_{l,u}.i + 1
    if I_{l,u}.i = L(l) then run = run − {I_{l,u}}
    for each c ∈ C
      remove {I_{l,u}} from mayConflict[I_{l,u}.i − Δ(l,c)][c]
      add {I_{l,u}} to mayConflict[I_{l,u}.i + Δ(l,c)][c]
```

Fig. 7. Pseudocode of multi-class PQS schedule.

with an instance $I_{l,u}$ executing step $I_{l,u}.i$ by inspecting the set *mayConflict*$[I_{l,u}.i][cls(l)]$. Accordingly, PQS will start/resume $I_{l,u}$ if either (1) the *mayConflict*$[I_{l,u}.i][cls(l)]$ set is empty or (2) all its members have lower priority than $I_{l,u}$. These changes are reflected in the *may-resume* function.

When an instance is started/resumed, it must be added to the appropriate *mayConflict* set in the *resume* function. This entails adding $I_{l,u}$ to all sets *mayConflict*$[x][c]$ such that $|I_{l,u}.i - x| < \Delta(l,c)$ and c is a query class. When a step in a plan is executed, the membership in the *mayConflict* sets is updated in the *execute-step* function. For any given class c, when an instance $I_{l,u}$ completes executing a step, it is removed from *mayConflict*$[I_{l,u}.i - \Delta(l,c)][c]$ and added to *mayConflict*$[I_{l,u}.i + \Delta(l,c)][c]$.

The multi-class PQS scheduler maintains $\sum_{c \in C} L(c)$ *mayConflict* sets, where $L(c)$ is the length of the plan for class c. Since c is a constant, the time complexity of the multi-class PQS is same as in the single class case.

Fig. 6(b) shows the schedule constructed by PQS in the multi-class case. The schedule is constructed similar to single-class case with the important distinction that *mayConflict* sets are constructed to recognize the pairwise minimum step distance requirements. For example, in slot 16, step 10 of Q_h and step 2 of Q_l may be executed concurrently since their minimum step distance constraint is satisfied: $\Delta_{h,l} = 10 - 2 = 8 \geq 8$.

Schedulability Analysis: To extend the analysis to the multi-class case, we follow the same approach as in the single query class case. We start by observing that once a query executes more than $E_l = \max_{m \in hp(l)} \Delta(l,m)$, no other query instance may preempt its execution. We split the execution of a query in two parts, a preemptable part of length E_l and a non-preemptable part of length $L_l - E_l$. Thus, the response time of a query is the sum of the response time of the preemptable part R'_l and the length of the non-premptable part: $R_l = L_l - E_l + R'_l$

The response time of the preemptable part is:

$$R'_l = E_l + \sum_{h \in hp(l)} \left\lceil \frac{R'_l}{P_h} \right\rceil \cdot C_{max}(l,h) \tag{6}$$

Theorem 3: An instance I_l is interfered by a higher priority instance I_h for at most $C_{max}(l,h) = \min(\Delta(h,l) + \Delta(l,h), L_l)$.

Proof: We analyze I_h's interference on I_l in three cases:
(1) If I_h is released while I_l is executing its nonpreemptable part, the interference is zero.
(2) I_h is released no later than I_l, then I_h's interference on I_l is at most $\Delta(h,l)$, since I_l may start when I_h completes $\Delta(h,l)$ steps. Thus, $C(l,h) = \Delta(h,l)$.
(3) If I_h is released while I_l is executing its preemptable part, I_h preempts I_l. Let x be the number of steps I_l has completed, when I_h preempts it. We note that $x \leq \Delta(l,h)$ since I_l otherwise both I_l and I_h may be executed concurrently without conflict. There are three sub-cases to be considered: (a) I_h is not preempted by a higher priority instance, (b) I_h is preempted by a higher priority instance and I_l is not resumed before I_h, and (c) I_h is preempted by a higher priority instance and I_l is resumed before I_h.
(3a) If I_h is not preempted by any higher priority instance, then I_l will be resumed after I_h completes $\Delta(h,l)+x$ steps to enforce the minimum step distance between I_l and I_h. Thus, the interference is $C(l,h) = \Delta(h,l) + x \leq \Delta(h,l) + \Delta(l,h)$.
(3b) If I_h is preempted by I_m after it completes y steps and I_l cannot be resumed before I_h is. We know that $y < \Delta(h,l)$ since otherwise I_l and I_h may be executed concurrently. The earliest time when I_l may be resumed is when I_m completes at least $\Delta(m,l) + x$ steps and I_h completes $\Delta(m,h) + y$ steps. Depending on the relationship between x and y there are two possible interference patterns as shown in Figure 8. If $x \geq y$ (see case 3b-i), then the interference is $C(l,h) = y \leq \Delta(h,l)$, since $y \leq \Delta(h,l)$. If $x < y$ (see case 3b-ii), then the interference $C(l,h) = \Delta(h,l) + x \leq \Delta(h,l) + \Delta(l,h)$ since $x \leq \Delta(l,h)$.
(3c) If I_h is preempted by I_m, PQS may resume I_l before I_h since it resumes a lower priority instance as soon as it does not conflict with any higher priority instance. As such, the earliest time I_l may be resumed, is after I_m completes $\Delta(m,l)+x$ steps. I_l will be executed until step x' when I_h may be resumed without conflicting with I_m i.e., after I_m completes executing $\Delta(m,h)+y$ steps. We note that $x' < \Delta(l,h)$ since otherwise I_l and I_h may be executed concurrently without conflict. In this case, the interference of I_h is $C(l,h) = \Delta(h,l) + x' \leq \Delta(h,l) + \Delta(l,h)$.
From all the above cases, I_h's worst-case interference on I_l is $C_{max}(l,h) = \Delta(h,l)+\Delta(l,h)$. However, when $L(h) < \Delta(h,l)+\Delta(l,h)$, I_h finishes before I_l reaches $\Delta(h,l)+\Delta(l,h)$; in this case the interference is only $L(h)$. Thus, I_h's worst-case interference on I_l is $C_{max}(l,h) = \max(\Delta(h,l) + \Delta(l,h), L(h))$. ■

Due to space constrains we omit the proof of the theorem.

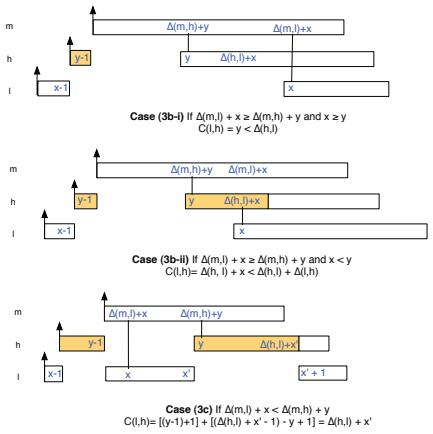


Fig. 8. Interference of I_h on I_l under multiclass PQS.

C. SQS Multi-class Scheduler

Similar to the single class case when we built the SQS scheduler by modifying the PQS scheduler, the multi-class SQS scheduler is built upon the multi-class version of the PQS scheduler. However, rather than allowing an instance $I_{l,u}$ to steal steal slack from a higher priority instance $I_{h,v}$ if $I_{l,u}$ completed at least $\Delta - S_h$ steps, in the multi-class case we allow $I_{l,u}$ to steal slack only if it completed at least $\Delta(l,h) - m_l$ steps, where $m_l = \min_{h \in hp(l)} S_h$. This ensures that if $I_{l,u}$ starts stealing slack, then it will always succeed. The extension of SQS to multiple classes does not increasing the time complexity.

The admission algorithm follows a similar approach to the single-class case. We divide the execution of a query l into a preemptable and a non-preemptable part. The length of the preemptable part is at most $E_l - m_l$, where $E_l = \max_{h \in hp(l)} \Delta(l,h)$ and $m_l = \min_{h \in hp(l)} S_h$. Accordingly, the length of the non-preemptable part is $L_l - (E_l - m_l)$. The worst-case interference of a higher priority instance on a given instance is:

Theorem 4: An instance I_l may be interfered by a higher priority instance I_h for at most $C_{max} = \min(\Delta(h,l) + \Delta(l,h) - m_l, L_l)$.

As we proved the worst-case interference for the single class SQS scheduler starting from the PQS interference result, it is straight forward to prove Theorem 4 starting from the results for the interference of multi-class PQS and recognizing that in the case of slack stealing there is a tighter bound on the variable x in Theorem 3 ($x < \Delta(l,h) - m_l$ rather than $x < \Delta(l,h)$).

To compute R'_l we must account for the jitter introduced by slack stealing, i.e., a higher priority instance I_h may delay its start by at most S_h. Accordingly, R' is:

$$R'_l(S_l) = (E_l - m_l) + S_l + \sum_{h \in hp(l)} \left\lceil \frac{R'_l(S_l) + S_h}{P_h} \right\rceil \cdot C_{max}(l,h)$$

where, $\Delta - m_l$ is the maximum length (execution time) of the preemptable part, S_l is the maximum time interval when I_l may be blocked by a lower priority instance due to slack stealing, and $C_{max}(l,h) = \min(\Delta(h,l) + \Delta(l,h) - m_l, L_l)$ is the worst-case interference when slack stealing is used.

An example of the schedule constructed by SQS is included in Fig. 6(c). In slot 6, an instance of the highest priority query is released while an instance of the medium priority query Q_m is executing. Unlike PQS, SQS does not preempt its execution but rather uses slack stealing to allow it to execute for an

additional two slots. In slot 8, the instance of Q_h can be executed concurrently with Q_m as their pairwise minimum step distance requirement $\Delta(h, m)$ is satisfied.

VI. Practical Considerations

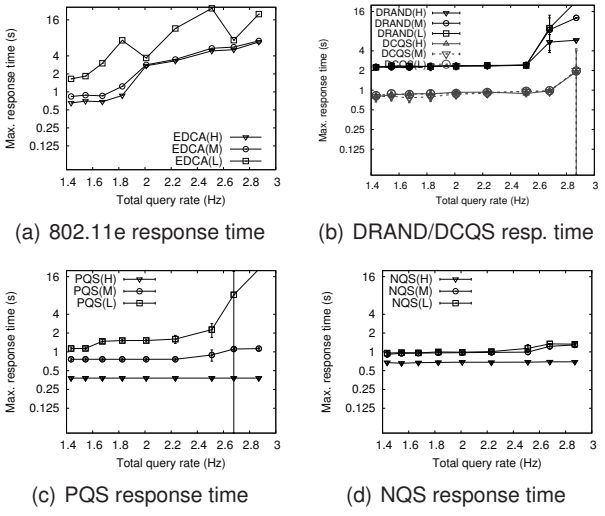
Topology Changes: Topology changes can negatively affect the performance of RTQS: due to changes in the routing tree, the plans would have to be consistently updated introducing a significant overhead. We propose two mechanisms to address this challenge: route diversity and retransmissions.

The routing tree algorithm may be adapted to allow for a child to have multiple parents, in effect providing route diversity. Accordingly, a node would be allowed 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 inter-release times as if n transmits to *all* potential parents. We trade-off some of the throughput in favor of better tolerating topology changes. This trade-off is similar to other TDMA algorithms designed to handle topology changes [26] [27]. Simulation results presented in [19] indicate that increasing the number of parents to two or three leads to throughput reductions of 9.8% and 12.6%, respectively. This suggests that route diversity is an effective mechanism for improving reliability without significantly lowering throughput.

Wireless links are known to have variable quality as environment change. During the computation of the workload demands for each node, the user must allocate sufficient time slots for potential packet transmissions to ensure reliability. RTQS already provides some robustness against changes in link quality by having multiple parents among which a node may switch. However, RTQS may also account for variations in link quality through a different mechanism. RTQS can accommodate such changes by increasing the workload of a link based on its quality. For example, the link layer commonly computes the expected number of transmissions (ETX) require to transmit packets successfully. RTQS could allocate a node to transmit up to ETX times a packet to ensure reliable delivery.

Time Synchronization: RTQS requires that all nodes within the interference range be time synchronized. The sensor network community has proposed several efficient time synchronization protocols [28]. For example, FTSP has an average time synchronization error of 1.4 μs per hop with minimal communication overhead. TDMA protocols handle time synchronization errors by using guard intervals on each slot. The guard intervals must exceed the size of synchronization error, which are on the order of $2\mu s$ for typical protocols such as FTSP [28]. This introduces a small constant overhead in each slot. In our simulation setup, our slot is 8.3 ms and, as a result, the addition of guard interval of $2\mu s$ would add minimal overhead.

Supporting Other Traffic: RTQS is optimized for improving the performance of periodic queries. However, other types of traffic may also exist (e.g., data dissemination, aperiodic queries). A 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 traffic types. It is straightforward to account for these additional slots in our analysis.



(a) 802.11e response time (b) DRAND/DCQS resp. time

(c) PQS response time (d) NQS response time

Fig. 9. Response time of baselines, PQS, and NQS

VII. Simulations

We implemented RTQS in NS2. Since we are interested in supporting *high data rate* applications such as structural health monitoring we configured our simulator according to the 802.11b settings having a bandwidth of 2Mbps. This is reasonable since several real-world structural health monitoring systems use 802.11b interfaces to meet their bandwidth requirements. An overview of these deployments may be found in [3]. At the physical layer a two-ray propagation model is used. We model interference according to the Signal-to-Interference-plus-Noise-Ratio (SINR) model, according to which a packet is received correctly if its reception strength divided by the sum of the reception strengths of all other concurrent packet transmissions is greater than a threshold (10 dbm in our simulations).

In the beginning of the simulation, the IC graph is constructed using the method described in [23]. 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 from different nodes. The node selects as its parent the node that has the best link quality indicator among those with smaller depth than itself. We determined the slot size as follows. We set the slot size to 8.3ms, which is large enough to transmit 2KB of data. In our simulations, all queries are executed according to the same plan as every node sends its data report in a slot.

For comparison we consider three baselines: 802.11e, DCQS [19] and DRAND [29]. We did not use 802.15.4 as a baseline, since the standard is designed for low data rate applications and hence is unsuitable for our target high data rate applications. 802.11e is a representative contention-based protocol that supports prioritization in wireless networks. In our simulations we use the Enhanced Distributed Channel Access (EDCA) function of 802.11e since it is designed for ad hoc networks. EDCA prioritizes packets using different values for the initial backoff, initial contention window, and maximum contention window of the CSMA/CA protocol. We configured these parameters according to their defaults in 802.11e. We used the 802.11e NS2 module from [30]. DRAND is a recently proposed TDMA protocol. DCQS is a query scheduling algorithm that constructs TDMA schedules

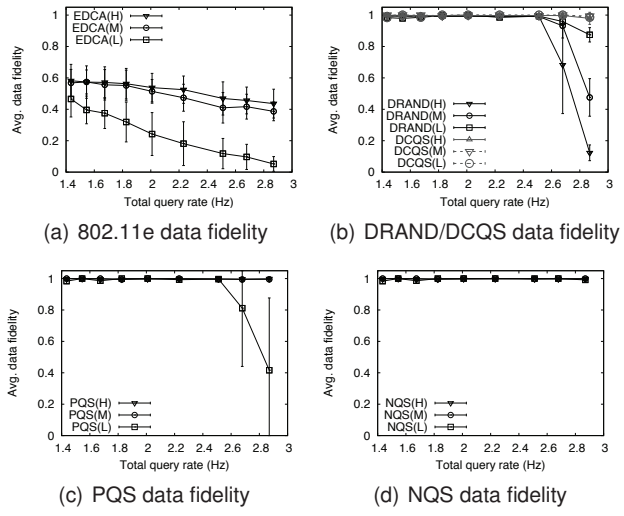


Fig. 10. Data fidelity of baselines, PQS, and NQS

to execute queries. However, neither DCQS nor DRAND support prioritization or real-time transmission scheduling.

We use *response time* and *data fidelity* to compare the performance of the protocols. The *response time* of a query instance is the time between its release time and completion time, i.e., when the base station receives the last data report for that instance. During the simulations, data reports may be dropped preventing some sources from contributing to the query result. The *data fidelity* of a query instance is the ratio of the number of sources that contributed to the aggregated data reports received by the base station and the total number of sources.

In the following we compare the performance of NQS and PQS with the baselines (Section VII-A) and evaluate the RTQS algorithms under different workloads and validate our response time analysis (Section VII-B). In all settings, we set the deadlines of the queries to be smaller or equal to the period as to match the conditions under for which we derived the schedulability analysis.

A. Comparison with Baselines

The presented results are the average of five runs on different topologies. The 90% confidence interval of each data point is also presented. In all experiments 100 nodes are deployed as follows: an area of 750m $\times$750m is divided into 75m $\times$ 75m grids in which a node is placed at random within each grid. We simulate three queries with high, medium and low priorities. The query priorities are determined based on their deadlines: the tighter the deadline, the higher the priority. The ratios of the query periods Q_H:Q_M:Q_L are 1.0:2.2:4.7. Deadlines are equal to periods.

Figs. 9 and 10 show the average response time and data fidelity of different protocols as the total query rate is increased from 1.43Hz to 2.87Hz. 802.11e EDCA provides prioritization between queries: when the total query rate is 1.43Hz, the average response times of Q_H and Q_L are 0.34s and 0.74s, respectively (see Fig. 9(a)). However, 802.11e EDCA has poor data fidelity for all queries (see Fig. 10(a)). The poor performance of 802.11e EDCA is due to high channel contention, which results in significant packet delays and packet drops. This shows the disadvantage of contention-based protocols for high data rate queries.

The TDMA protocols, DCQS and DRAND (see Figs. 9(b) and 10(b)), have significantly higher data fidelity than 802.11e EDCA. The data fidelity results indicate that DCQS provides a higher capacity than DRAND. Moreover, DCQS provides lower response time than DRAND (see Fig. 9(b)). DCQS performs better because it exploits the inter-node dependencies introduced by queries in WSNs. However, neither protocol provides query prioritization since all queries have similar response times.

In contrast to DCQS and DRAND, PQS provides query prioritization as seen in their response times. For instance, when the total query rate is 2.51Hz, PQS provides an average response time of 0.38s for Q_H, which is 75% lower than the average response time of 1.48s for Q_L (see Fig. 9(c)). PQS achieves the same capacity as DRAND, but lower than DCQS due to the high cost of preemption (see Section IV-E). PQS achieves close to 100% fidelity when the total query rate is lower than 2.51Hz (see Fig. 10(c)). For higher query rates, the fidelity drops because the offered load exceeds PQS's real-time capacity (the schedulability test failed at these rates). NQS also provides query prioritization (the y-axis has a log scale), but the differences in response times are smaller than in PQS due to the priority inversions of non-preemptive scheduling (see Fig. 9(d)). In contrast to PQS, NQS has almost 100% data fidelity for all queries when the total query rate is as high as 2.87Hz. Therefore, NQS achieves higher capacity than PQS. This comparison shows the tradeoff between latency and capacity predicted by our analysis.

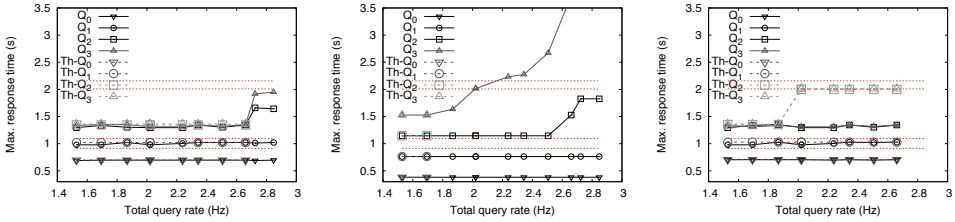
B. Comparison of RTQS Algorithms

In this subsection we compare the performance of all RTQS algorithms and validate their response time analysis. We consider four queries Q_0, Q_1, Q_2, and Q_3 in decreasing order of priority. The ratios of their periods Q_0:Q_1:Q_2:Q_3 is 1.0:1.2:2.2:3.2. In this experiment, we fix the rates of the queries and vary the deadline of the highest priority query.

To evaluate the RTQS algorithms under a broad range of workloads, we perform three experiments. In the first experiment, we fix the deadlines of the queries and vary their rates. In the second experiment, we fix the query rates and vary the deadline of the highest priority query. In the last experiment, we evaluate the performance of the RTQS algorithms for multiple classes.

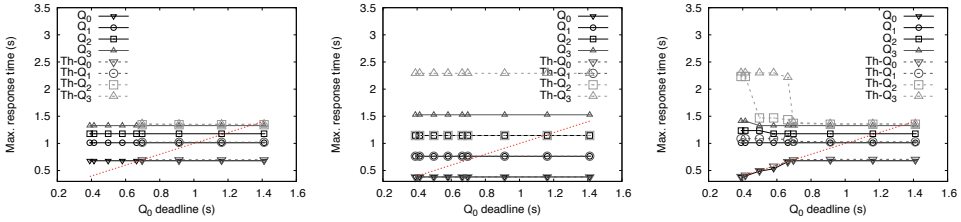
Experiment 1: Figs. 11(a) - 11(c) show the measured and the theoretical maximum response times of NQS, PQS, and SQS under different total query rates. The dotted horizontal lines indicate the query deadlines. NQS meets all deadlines when the total query rate is within 2.85Hz. In contrast, PQS supports a lower query rate since Q_3 misses its deadline when the total query rate is 2.23Hz. The long response time of Q_3 is due to the high preemption cost suffered by the low priority queries under PQS. This indicates that PQS is unsuitable for workloads in which the low priority queries have tight deadlines.

Similar to NQS, SQS can support a higher query rate than PQS without missing deadlines. In this experiment, the deadlines are lax and hence preemption is not necessary for meeting them. As such, SQS dynamically avoids preemption and the associated real-time capacity reduction. SQS achieves a slightly lower real-time capacity than NQS because it is limited by the conservative response time analysis. When the admission algorithm decides that the queries are unschedulable, it cannot find a slack assignment for the queries. Therefore we cannot run SQS at a rate beyond its theoretical bound. In contrast, we may increase the rate further under NQS, which achieves a higher real-time capacity than its theoretical bounds because its response time analysis is derived based on worst-case arrival patterns which do not always occur in our simulations.



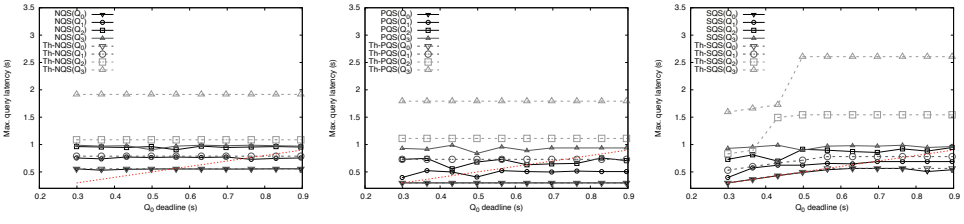
(a) NQS response time (b) PQS response time (c) SQS response time

Fig. 11. Response time of queries when workload is varied by changing rates. All queries belong to the same class.



(a) NQS response time (b) PQS response time (c) SQS response time

Fig. 12. Response time of queries when workload is varied by changing the deadline of Q_0.



(a) NQS response time (b) PQS response time (c) SQS response time

Fig. 13. Response time of queries when workload is varied by changing the deadline of Q_0.

Experiment 2. In this experiment we increase the deadline of the lowest priority query and vary the deadline of the highest priority query Q_0. This experiment evaluates the RTQS algorithms when the low priority queries have lax deadlines.

Figs. 12(a) - 12(c) show the maximum response times of NQS, PQS, and SQS, respectively. For clarity, only Q_0's deadline is plotted since in this experiment the other queries always meet their deadlines. PQS meets Q_0's deadline when it is 0.39s. In contrast, NQS meets its deadline only when Q_0's deadline is bigger then 0.69s. NQS misses Q_0's deadline when it is tight due to the priority inversion under non-preemptive scheduling. This indicates that NQS is unsuitable for high priority queries with tight deadlines. Interestingly, under SQS, the response time of Q_0 changes depending on its deadline (Fig. 12(c)). As the deadline becomes tighter, the response time of Q_0 also decreases and remains below the deadline. We also see an increase in the response times of the lower priority queries as Q_0's deadline is decreased. This is because as Q_0's deadline decreases the lower priority queries may steal less slack from Q_0. This shows that SQS adapts effectively based on query deadlines. Moreover, note that SQS provides smaller latencies for the lower priority instances than PQS. This is because SQS has a higher real-time capacity than PQS since it uses preemption only when it is necessary for meeting packet deadlines.

Experiment 3. In this experiment, we compare the performance of the RTQS algorithms in the presence of multiple 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 to the base-station. Similar to the previous experiments, data merging is performed as data is routed to the base-station. In this experiment there are two classes: c_0 includes 100% of the leaf nodes while c_1 includes 60% of the leaf nodes. The queries Q_0 and Q_2 belong to class c_0 while Q_1 and Q_3 to class c_1.

Figs. 13(a) - 13(c) show the maximum response times for NQS, PQS and SQS when the deadline is varied. In each graph we also plot the deadline of Q_0. Similar to the previous experiment, PQS schedules the workload for tighter deadlines of Q_0 than NQS. This is because in contrast to NQS, PQS does not introduce any priority inversions. From the Figs. 13(a) and 13(b) it is clear that neither algorithm changes its behavior as Q_0's deadline is varied. In contrast, SQS adapts its behavior to meet Q_0's deadline. In all experiments, the measured response times of all RTQS algorithms are lower than the analytical worst-case response times.

VIII. Conclusions

High data rate real-time queries are important to many wireless cyber-physical systems. This paper proposes RTQS, a novel transmission scheduling approach designed real-time queries in WSNs. RTQS bridges the gap between wireless sensor networks and schedulability analysis techniques which have traditionally been applied to real-time processor scheduling.

We first analyze the inherent tradeoff between throughput and prioritization under conflict-free query scheduling. We then present the design and schedulability analysis of three new real-time scheduling algorithms for prioritized transmission scheduling. NQS achieves high throughput at the cost of priority inversion, while PQS eliminates priority inversion at the cost of

query throughput. SQS combines the advantages of NQS and PQS to achieve high query throughput while meeting query deadlines. NS2 simulations demonstrate that both NQS and PQS achieve significantly better real-time performance than representative contention-based and TDMA protocols. Moreover, SQS can maintain desirable real-time performance by adapting to deadlines. Real-time query scheduling provides an promising approach to provide predictable real-time queries in WSNs.

REFERENCES

[1] R. Mangharam, A. Rowe, R. Rajkumar, and R. Suzuki, "Voice over sensor networks," in *Real-Time Systems Symposium*, 2006.
[2] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, "Health monitoring of civil infrastructures using wireless sensor networks," in *Information Processing in Sensor Networks*, 2007.
[3] 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, 2006.
[4] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, and M. Nixon, "Wirelesshart: Applying wireless technology in real-time industrial process control," in *Real-Time and Embedded Technology and Applications Symposium*, 2008.
[5] H. Zhu, M. Li, I. Chlamtac, and B. Prabhakaran, "A survey of quality of service in ieee 802.11 networks," in *IEEE Wireless Communications*, 2004.
[6] W. Pattara-Atikom, P. Krishnamurthy, and S. Banerjee, "Distributed mechanisms for quality of service in wireless lans," *IEEE Wireless Communications*, 2003.
[7] K. Karenos, V. Kalogeraki, and S. Krishnamurthy, "A rate control framework for supporting multiple classes of traffic in sensor networks," in *Real-Time Systems Symposium*, 2005.
[8] G.-S. Ahn, A. Campbell, A. Veres, and L.-H. Sun, "Supporting service differentiation for real-time and best-effort traffic in stateless wireless ad hoc networks (swan)," *IEEE Transactions on Mobile Computing*, vol. 1, no. 3, 2002.
[9] M. Barry, A. Campbell, and A. Veres, "Distributed control algorithms for service differentiation in wireless packet networks," in *INFOCOM*, 2001.
[10] T. He, J. Stankovic, C. Lu, and T. Abdelzaher, "Speed: a stateless protocol for real-time communication in sensor networks," in *International Conference on Distributed Computing Systems*, 2003.
[11] E. Felemban, C.-G. Lee, and E. Ekici, "Mmspeed: Multipath multi-speed protocol for qos guarantee of reliability and timeliness in wireless sensor networks," *IEEE Transactions on Mobile Computing*, 2006.
[12] A. Koubaa, M. Alves, and E. Tovar, "i-GAME: an implicit GTS allocation mechanism in IEEE 802.15.4 for time-sensitive wireless sensor networks," in *Euromicro Conference on Real-Time Systems*, 2006.
[13] T. Facchinetti, L. Almeida, G. Buttazzo, and C. Marchini, "Real-time resource reservation protocol for wireless mobile ad hoc networks," in *Real-Time Systems Symposium*, 2004.
[14] H. Li, P. Shenoy, and K. Ramamritham, "Scheduling messages with deadlines in multi-hop real-time sensor networks," in *Real-Time and Embedded Technology and Applications Symposium*, 2005.
[15] M. Caccamo, L. Y. Zhang, L. Sha, and G. Buttazzo, "An implicit prioritized access protocol for wireless sensor networks," in *RTSS*, 2002.
[16] A. Saifullah, Y. Xu, C. Lu, and Y. Chen, "End-to-end delay analysis for fixed priority scheduling in wirelesshart networks," in *Real-Time and Embedded Technology and Applications Symposium*, 2010.
[17] H. Zhang, P. Soldati, and M. Johansson, "Optimal link scheduling and channel assignment for convergecast in linear wirelesshart networks," in *Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, 2009.
[18] A. Saifullah, Y. Xu, C. Lu, and Y. Chen, "Real-time scheduling for wirelesshart networks," in *Real-Time Systems Symposium*, 2010.
[19] O. Chipara, C. Lu, J. Stankovic, and G. Roman, "Dynamic conflict-free transmission scheduling for sensor network queries," *IEEE Transactions on Mobile Computing*, vol. 10, no. 5, 2011.
[20] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: a tiny aggregation service for ad-hoc sensor networks," *SIGOPS Operating Systems Review*, vol. 36, 2002.
[21] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *Embedded Networked Sensor Systems*, 2005.
[22] N. Kimura and S. Latifi, "A survey on data compression in wireless sensor networks," in *Information Technology: Coding and Computing*, 2005.
[23] G. Zhou, T. He, J. A. Stankovic, and T. F. Abdelzaher, "RID: radio interference detection in wireless sensor networks," in *INFOCOM*, 2005.
[24] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.
[25] O. Chipara, C. Lu, and G.-C. Roman, "Real-time query scheduling for wireless sensor networks," in *Real-Time Systems Symposium*, 2007.
[26] I. Chlamtac and A. Farago, "Making transmission schedules immune to topology changes in multi-hop packet radio networks," *IEEE/ACM Transactions on Networking*, vol. 2, no. 1, 1994.
[27] J.-H. Ju and V. O. K. Li, "An optimal topology-transparent scheduling method in multihop packet radio networks," *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, 1998.
[28] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, "The flooding time synchronization protocol," in *Embedded Networked Sensor Systems*, 2004.
[29] I. Rhee, A. Warrior, J. Min, and L. Xu, "DRAND: Distributed randomized TDMA scheduling for wireless ad hoc networks," in *MobiHoc*, '06.
[30] M. Lacage, "Ns2 802.11b/e support. http://yans.inria.fr/ns-2-80211/."

Octav Chipara Octav Chipara is an Assistant Professor of Computer Science at University of Iowa. He received his PhD from Washington University in St. Louis. 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 Chenyang Lu is an Professor of Computer Science and Engineering at Washington University in St. Louis. He received the Ph.D. degree from University of Virginia in 2001, the M.S. degree from Institute of Software, Chinese Academy of Sciences in 1997, and the B.S. degree from University of Science and Technology of China in 1995, all in computer science. The author and co-author of more than 100 publications, Professor Lu received an NSF CAREER Award in 2005 and a Best Paper Award at International Conference on Distributed Computing in Sensor Systems in 2006. His research interests include real-time embedded systems, wireless sensor networks, and cyber-physical systems.

Gruia-Catalin Roman Gruia-Catalin Roman is Professor of Computer Science at University of New Mexico. Roman has an established research career with numerous published papers in a multiplicity of computer science areas including mobile computing, formal design methods, visualization, requirements and design methodologies for distributed systems, interactive high speed computer vision algorithms, formal languages, biomedical simulation, computer graphics, and distributed database. His current research involves the study of formal models, algorithms, design methods, and middleware for mobile computing and sensor networks.