Real-time Query Scheduling for Wireless Sensor Networks

Octav Chipara, Chenyang Lu, Gruia-Catalin Roman
Department of Computer Science and Engineering
Washington University in St. Louis
{ochipara, lu, roman}@cse.wustl.edu

Abstract

Recent years have seen the emergence of wireless sensor network systems that must support high data rate and real-time queries of physical environments. 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 prioritization and throughput in conflict-free query scheduling. We then present three new real-time scheduling algorithms. The non-preemptive query scheduling algorithm achieves high throughput while introducing priority inversions. The preemptive query scheduling algorithm eliminates priority inversion at the cost of reduced throughput. The slack stealing query scheduling algorithm combines the benefits of preemptive and non-preemptive scheduling by improving the throughput while meeting query deadlines. Furthermore, we provide schedulability analysis for each scheduling algorithm. The analysis and advantages of our scheduling algorithms are validated through NS2 simulations.

1 Introduction

Recent years have seen the emergence of wireless sensor networks (WSNs) that must support real-time data collection at high data rates. Representative examples include patient monitoring [13], emergency response [17], and structural health monitoring [15] and control. Such systems pose significant challenges. First, the system must handle various types of traffic with different deadlines. For example, during an earthquake, the acceleration sensors mounted on a building must be sampled and their data delivered to the base station in a timely fashion to detect any structural damage. Such traffic should have higher priority than temperature data collected for climate control. Thus, a real-time communication protocol should provide *effective prioritization* between different traffic classes while meeting their respective deadlines. Second, the system must support *high throughput* since it may generate high volumes of traffic. For example, structural health monitoring require a large number of acceleration sensors to be sampled at high rates generating high network loads. Furthermore, the system must deliver data to base stations or users within their deadlines. Therefore, it is important for the system to achieve *predictable and bounded end-to-end latencies*.

Many WSN applications use query services to periodically collect data from sensors to a base station. In this paper, we propose ***R**eal-**T**ime **Q**uery **S**cheduling* (RTQS), a transmission scheduling approach for real-time queries in WSNs. To meet this challenge, we present a set of new real-time query scheduling algorithms and associated schedulability analysis, which *bridge the gap between WSNs and real-time scheduling theories*. Our scheduling algorithms exploit the unique characteristics of WSN queries including many-to-one communication, in-network aggregation and periodic timing properties.

This paper makes four contributions: First, we show through analysis and experiments that query scheduling has an inherent tradeoff between prioritization and throughput. Second, we developed three scheduling algorithms: (1) The nonpreemptive query scheduling algorithm achieves high throughput at the cost of some priority inversions. (2) The preemptive query scheduling algorithm achieves good prioritization by eliminating priority inversions. (3) The slack stealing scheduling algorithm combines the advantages of preemptive and non-preemptive scheduling algorithms by improving the throughput while meeting all query deadlines. Third, we derive latency upper bounds for each scheduling algorithm. This enables us to guarantee that the admitted queries meet their deadlines. Our analysis enables query services to handle overload conditions. through online admission and rate control. Finally, we provide simulations that demonstrate the advantages of RTQS over contention-based and TDMA-based protocols in term of both real-time performance and throughput.

The paper is organized as follows. Section 2 compares our approach to existing work. Section 3 describes the query and network models. Section 4 details the design

and analysis of RTQS. Section 5 provides simulation results. Section 6 concludes the paper.

2 Related Work

Real-time communication protocols can be categorized into contention-based and TDMA-based protocols. Contention-based protocols support real-time communication through probabilistic differentiation. This is usually achieved by adapting the parameters of the CSMA/CA mechanism such as the contention window and/or initial back-off [8][14]. Rate and admission control [9][1] have also been proposed for contention-based protocols to handle overload conditions. However, contention-based approaches have two inherent drawbacks that make them unsuitable for high data rate and real-time applications. First, packet latencies are highly variable due to the random back-off mechanisms. Second, their maximum throughput is low due to channel contention under heavy load.

TDMA protocols can provide predictable packet latencies and achieve higher throughput than contention-based protocols under heavy load. The IEEE 802.15.4 standard for WSNs has a reservation mechanism for providing predictable delays in single hop networks. A more flexible slot reservation mechanism is proposed in [10] where slots are allocated based on delay or bandwidth requirements. Two recent papers proposed real-time communication protocols for robots [7][12]. Both protocols assume that at least one robot has complete knowledge of the robots' positions and/or network topology. While the protocols may work well for small teams of robots, they are not suitable for queries in large-scale WSNs. Implicit EDF [4] 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 even in large multi-hop networks without requiring multiple frequencies.

Two recent protocols that support real-time flows in WSNs have been proposed. In [17] a scheduling based solution is proposed to support voice streaming over real-time flows. The real-time chains protocol [3] extends a contention-based scheme called Black Burst to support packet prioritization. However, these protocols only support real-time *flows* involving only one or a few data sources. In contrast, RTQS is optimized for real-time *queries* that collect sensor data from many sources.

In earlier work we proposed DCQS [6], a TDMA protocol that achieves high throughput by exploiting explicit query information provided by the query service. However, DCQS does not support query prioritization or real-time communication, which is the focus of this paper.

3 System Models

In this section, we characterize the query services for which RTQS is designed and describe our network model.

3.1 Query Model

RTQS assumes a common query model in which source nodes produce data reports periodically. This model fits many applications that gather data from the environment at user specified rates. A query l is characterized by the following parameters: a set of sources, a function for in-network aggregation [16], the start time ϕ_l, the query period P_l, the query deadline D_l, and a static priority. 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 uth 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 given by the priority of its query. If two instances have the same query priority, the instance with the earliest release time has higher priority. For each query instance a node i needs $W_l[i]$ slots to transmit its (aggregated) data report to its parent.

A query service works [16] as follows: a user issues a query to a sensor network through a base station, which disseminates the query parameters to all nodes. The query service maintains a *routing tree* rooted at the base station. The query service supports in-network data aggregation. Accordingly, 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 workload dynamics efficiently.

3.2 Network Model

RTQS models a WSN as an Interference-Communication (IC) graph. The IC graph, IC(E,V), has all nodes as vertices and has two types of directed edges: *communication* and *interference* edges. A *communication edge* $\overrightarrow{ab}$ indicates that 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[19]. The IC graph may be computed and stored in a distributed fashion: a node needs to know *only* its incoming/outgoing communication and interference edges. In [19], Zhou et al. present RID, a practical solution for constructing the IC graph of a WSN. A node can use RID to determine its adjacent communication and interference edges.

We assume that clocks are synchronized. Clock synchronization [17] is a fundamental service in WSN as many applications must time-stamp their sensor readings to infer meaningful information about the observed events.

4 Real-time Query Scheduling

RTQS achieves 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 determines the time slot in which each step in a plan is executed. To improve the throughput, the scheduler may execute steps from multiple query instances in the same slot as long as they do not conflict with each other.

RTQS works as follows: (1) When a query is submitted, RTQS identifies a plan for its execution. As discussed in Section 4.1, usually multiple queries be executed using the same plan. Therefore, RTQS may reuse a previously constructed plan for the new query. When no plan may be reused, the planner constructs a new one. (2) RTQS determines if a query meets its deadline using our schedulability analysis. The schedulability analysis is performed on the base station. If the query is schedulable, the parameters of the query are disseminated; otherwise, the query is rejected. (3) At run-time the scheduler running on each node executes all admitted queries in a localized fashion.

In contrast to DCQS which does not support real-time communication, the key contribution of RTQS is the design and analysis of three *real-time* scheduling algorithms. Each scheduling algorithm achieves a different tradeoff between query prioritization and throughput. The *Nonpreemptive Query Scheduling* (NQS) algorithm achieves high throughput at the cost of priority inversion, while the *Preemptive Query Scheduling* (PQS) algorithm eliminates priority inversion at the cost of lower throughput. The *Slack-stealing Query Scheduling* (SQS) algorithm combines the benefits of NQS and PQS by improving the throughput while meeting all deadlines.

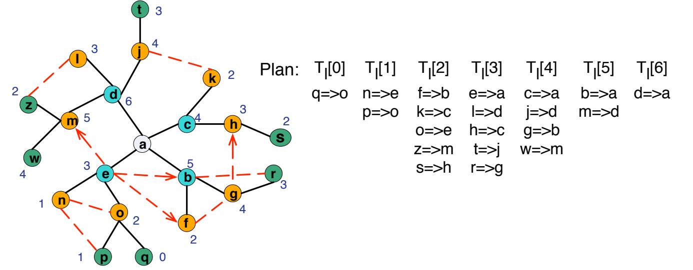


Figure 1. IC graph and associated plan.

4.1 Constructing plans

A plan has two properties: (1) it respects the precedence constraints introduced by data aggregation: a node is assigned to transmit in a later step than any of its children. (2) Each node is assigned in 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. To facilitate in-network aggregation, a node waits to receive the data reports from all its children before transmitting the aggregated data report to its parent. Therefore, to reduce the query latency, the planner assigns the transmissions of a node with a larger depth in the routing tree to an earlier step in the plan. This strategy reduces the query latency because it reduces the time a node waits for the data reports from all its children.

Fig. 1 shows an IC graph and the plan 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. 1 is constructed assuming that the data report generated by a node can be transmitted in a single step for each instance. 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. The precedence constraints introduced by aggregation are respected. For example, nodes p and q are assigned in earlier steps than their parent o. In [6] we proposed a distributed algorithm for constructing plans based on the IC graph. Upon the completion of the algorithm each node knows in what steps it transmits and receives. The details of the algorithm can be found in [6].

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 [6]). 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 $W_l[i]$ 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 $W_l[i] = W_h[i]$ 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., $W_{max}[i] = W_{avg}[i] = 1$ 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.

4.2 Overview of Scheduling Algorithms

The scheduler executes a query instance according to the plan of its query. The scheduler improves the query throughput 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 each plan are executed in order: if step $I_{l,u}.i$ is executed in slot s_i, step $I_{l,u}.j$ is executed in slot $s_j < s_i$ then $I_{l,u}.j < I_{l,u}.i$. This ensures that the precedence constraints required by aggregation are preserved.

The scheduler maintains a record of the start time, period, and priority of all admitted queries. Additionally, the scheduler knows the step numbers in which the host node is assigned to transmit or receive in each plan and the plan's length. RTQS supports both preemptive and nonpreemptive query scheduling.

We first consider 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. Unfortunately, the time complexity of this approach is high, since each pair of steps must be checked for conflicts. Since the scheduler dynamically determines the steps executed in a slot, it must have low time complexity.

To reduce the time complexity of the scheduler we introduced the concept of *minimum step distance* in [6][1]. Let $I_{l,u}.i$ and $I_{h,v}.j$ be two steps in the plans of any instances $I_{l,u}$ and $I_{h,v}$, respectively. 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 the two steps $I_{l,u}.i$ and $I_{h,v}.j$ may be executed concurrently without conflict:

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

where, L_l and L_h are the lengths of the plans of queries l and h, respectively. Therefore, to ensure that no conflicting transmissions are executed in a slot, it is sufficient 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. For simplicity 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 [6]. 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, in practice the number of minimum step distances that must be stored the memory cost is small since the planner uses only few plans.

4.3 Nonpreemptive Query Scheduling

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

[1]This is called the minimum inter-release time is [6].

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. If a node does not transmit or receive in slot s, it turns off its radio for the duration of the slot.

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) is larger Δ. When an instance starts, it is moved from the *release* queue to the *run* queue.

Consider the example shown in Fig. 3(a) where three queries, Q_{hi}, Q_{med} and Q_{lo} are executed according to the shown workload parameters. Each query is executed according to 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_{lo} (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_{med} and Q_{hi} are released in slots 2 and 6, respectively. However, neither may start until slot 8 when I_{lo} completes 8 steps (i.e., when $I_{lo}.i = 8 \geq \Delta$) resulting in priority inversions. I_{hi} then starts at slot 8 since it is the highest priority instance in *release*.

4.4 Preemptive Query Scheduling

A drawback of NQS is that it introduces priority inversions. To eliminate prioritization inversion, 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: *mayConflict*$[x] = \{I_{h,v} \in$ *run* $| |x - I_{h,v}.i| < \Delta\}$.

PQS (see Fig. 2) maintains a *run* queue and a *release* queue which are keyed by the query instance priority. 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 $I_{l,u}.i$ of $I_{l,u}$ 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* queue to the *run* queue. The membership of $I_{l,u}$ in the *mayConflict* sets is updated to reflect that $I_{l,u}$

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($I_{l,u}$)

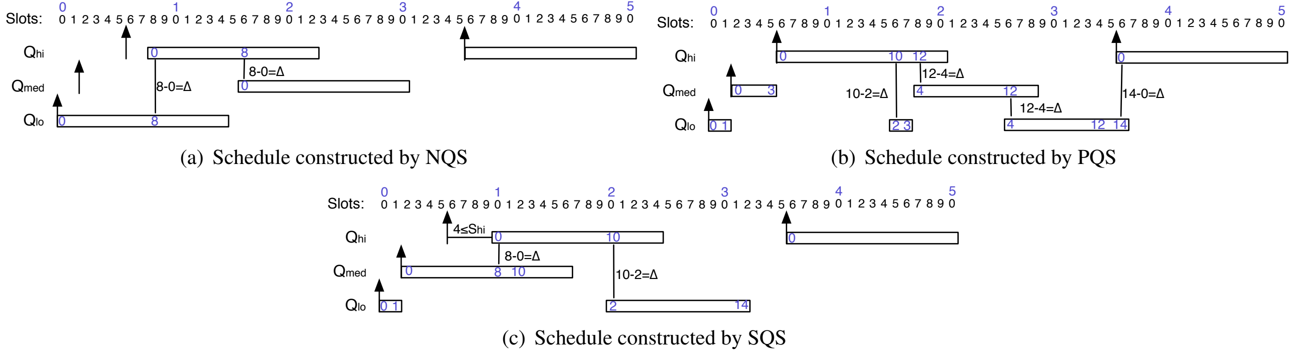
resume($I_{l,u}$):
  run = *run* $\cup \{I_{l,u}\}$; *release* = *release* $- \{I_{l,u}\}$
  add $I_{l,u}$ to all mayConflict$[x]$ such that $|I_{l,u}.i - x| < \Delta$
preempt(S):
  run = *run* $- S$; *release* = *release* $\cup S$
  remove $I_{l,u}$ from all mayConflict
may-resume($I_{l,u}$):
  if (mayConflict$[I_{l,u}.i] = \emptyset$) 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$[I_{l,u}.i - \Delta + 1] - \{I_{l,u}\}$
  mayConflict$[I_{l,u}.i + \Delta]$=mayConflict$[I_{l,u}.i + \Delta] \cup \{I_{l,u}\}$

Figure 2. PQS pseudocode

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* queue to the *release* queue 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. 3(b) shows the schedule of PQS for the same workload used in the example for NQS. Instance I_{lo} starts in slot 0 since no other instances have been released (*mayConflict*[0]= $\emptyset$). I_{med} is released in slot 2. Since *mayConflict*[0]= $\{I_{lo}\}$ and I_{med} has higher priority than I_{lo}, PQS preempts I_{lo}. Consequently, I_{lo} is removed from the *run* queue and all *mayConflict* sets, and it is added to the *release* queue. I_{med} is added to *run* queue and to all *mayConflict*$[x]$ sets where $0 \leq x < 8$. I_{hi} is released in slot 6. Since *mayConflict*$[0] = \{I_{med}\}$ and I_{hi} has higher priority then I_{med}, PQS preempts I_{med} and starts I_{hi}. The *mayConflict* sets are updated accordingly. An interesting case occurs in slot 16, when I_{hi} executes step 10. At this point,



(a) Schedule constructed by NQS

(b) Schedule constructed by PQS

(c) Schedule constructed by SQS

Figure 3. Scheduling with different prioritization policies. Workload: P_{hi}=30, D_{hi} =20,P_{med}=65, D_{med} =28, P_{lo}=93, D_{lo} =93.

mayConflict$[2] = \emptyset$ since I_{med} was preempted and I_{hi} completed 10 steps ($|10-2| \geq 8$). As a result, I_{lo} may execute step 2 in its plan while I_{hi} executes step 10 without conflict. I_{hi} and I_{lo} are executed concurrently until step 18 because their step distance exceeds the minimum step distance. In the beginning of slot 18, *mayConflict*[4]=$\{I_{lo}\}$. Note that I_{hi} is not a member of this set since $|12-4| \geq 8$. Since the step counter of I_{med} is 4 and I_{med} has higher priority than I_{lo}, PQS preempts I_{lo} and resumes I_{med}. PQS then updates the conflict sets by removing I_{lo} from all of them and adding I_{med} to *mayConflict*$[x]$ sets where $|x-4| < 8$. I_{lo} resumes in slot 26 when *mayConflict*[4] becomes empty. The example shows that by eliminating priority inversion PQS achieves lower latencies for I_{hi} and I_{med} than NQS. However, the query throughput is lower because it allows less overlap in the execution of instances. This exemplifies the tradeoff between prioritization and throughput in query scheduling. In the next section, we will characterize this tradeoff analytically.

4.5 Analysis of NQS and PQS

In this section we present worst-case response analyses for PQS and NQS. The worst-case response time of a query is the maximum query latency of any of its instances. Our analysis can be used for admission and rate control at the base station when a query is submitted. In our analysis we assume that the deadlines are shorter than the periods.

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: $R_l = W_l + L$. Note that for convenience we use the slot size as the time unit.

To compute W_l, we construct a recurrent equation similar to the response time analysis for processor scheduling [2]. 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. Their proofs are not included here due to space limit but may be found in [5].

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

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

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. W_l can be computed by solving (1) using a fixed point algorithm similar to that of the response time analysis [2].

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[2]. 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: $R_l = L - \Delta + R'_l$.

[2]Our analysis assumes that every instance is released in the beginning of a slot, which is the time granularity of our scheduling algorithms. Strictly speaking, a higher priority instance may still be blocked by at most one slot. This blocking term can be easily incorporated into our analysis.

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 [2]. 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.*

Proof:
We analyze I_h's interference on I_l in the following cases. (1) If I_h is released no later than I_l, then I_h's interference on I_l is at most Δ, since I_l may start when I_h completes Δ steps.
(2) If I_h is released while I_l is executing its nonpreemptable part, the interference is zero.
(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 $x \leq \Delta$ since I_l is executing its preemptable part. There are three subcases. (3a) If I_h is not preempted by any higher priority instance, then I_l will be resumed after I_h completes $\Delta + x$ steps to enforce the minimum step distance between I_l and I_h. Thus, the interference is $C = \Delta + x$. If I_h is preempted after executing $y \leq \Delta$ steps we must consider two cases as illustrated in Fig. 4. Recall that plans start with step 0. (3b) If $x \geq y$, PQS resumes I_h before I_l due to the minimum step distance constraint. In this case, I_h's interference on I_l is $C = \Delta + x$. (3c) If $x < y$, then I_l is resumed before I_h and it may execute up to $(x - y)$ steps until I_h is resumed. Thus, I_h's interference on I_l is $C = \Delta + y$.

From all the above cases, I_h's worst-case interference on I_l is $C = \Delta + \max(x, y)$. Since $x \leq \Delta$ and $y \leq \Delta$, then $C_{max} \leq 2\Delta$. However, when $L < 2\Delta$, I_h finishes before I_l reaches 2Δ; in this case the interference is only L. Thus, I_h's worst-case interference on I_l is $C_{max} = \min(2\Delta, L)$.

It is important to note that preempting an instance results in higher interference than the nonpreemptive case. As shown in the above proof, the interference in the preemptive case is $C = \Delta + \max(x, y)$ compared to Δ in the nonpreemptive case. Therefore, preemption incurs $\max(x, y)$ slots of additional interference compared to the no preemption case. The additional interference in the preemptive case results in a lower degree of concurrency and hence lower query throughput. This shows the inherent trade-off between prioritization and throughput in conflict-free query scheduling.

4.6 Slack Stealing Query Scheduling

SQS combines the benefits of NQS and PQS in that it improves query throughput while meeting all deadlines. The design of SQS is based on the observation that preemption lowers throughput, 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 and determines 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 vth instance of h. 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 throughput 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 and (3) moves all instances from the *pending* queue to the *release* queue.

Fig. 3(c) shows the schedule under SQS with the example workload. Assume that the admission algorithm of SQS determined that Q_{hi} and Q_{med} have slacks $S_{hi} = 5$ and $S_{med} = 2$, respectively. I_{lo} is released and starts its execution in slot 0. I_{med} is released in slot 2. SQS preempts I_{lo}, because even if I_{med} would be postponed for $S_{med} = 2$ slots, I_{lo} would not complete $\Delta = 8$ steps. I_{hi} is released in slot 6. SQS decides to continue executing I_{med} because in $4 \leq S_{hi}$ slots, I_{med} will complete executing $\Delta = 8$ steps, i.e., SQS avoids preempting I_{med} by allowing it to steal 4 slots from I_{hi}. SQS uses preemption in slot 2 but not in slot

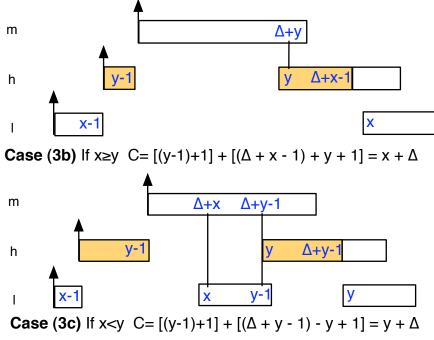


Figure 4. Interference of I_h on I_l under PQS.

6. This highlights that SQS can adapt preemption decisions to improve throughput 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 a query 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:

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

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 $m_l = \min_{x \in hp(l)} S_x$.*

The proof of the theorem is not included due to space limit and can be found in [5].

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:

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

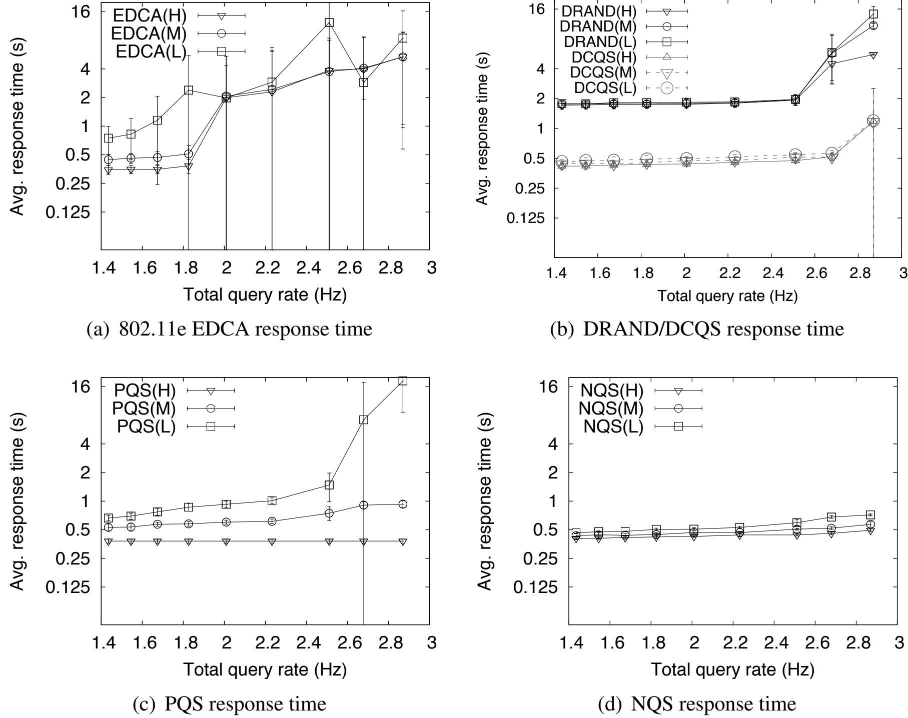
where, $\Delta - m_l$ is the maximum length (execution time) of the preemptable part, S_l is the maximum time interval when I_l may be blocked by a lower priority instance due to slack stealing, and $C_{max}(l, h) = \min(2\Delta - m_l, L)$ is the worst-case interference when slack stealing is used.

5 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 [15]. 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 [19]. 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 assume that a node samples its accelerometer at 100Hz and buffers 50 16-bit data points before transmitting its data report to its parent. To reduce the number of transmissions, data merging is employed: a node waits to receive the data reports from its children and merges their readings with its own in a single data report which it sends to its parent. In our experiments, the maximum number of descendants of any node is 20, so the maximum size of a data report containing 16-bit measurements is 2KB. Accordingly, 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[6] and DRAND[18]. 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 con-



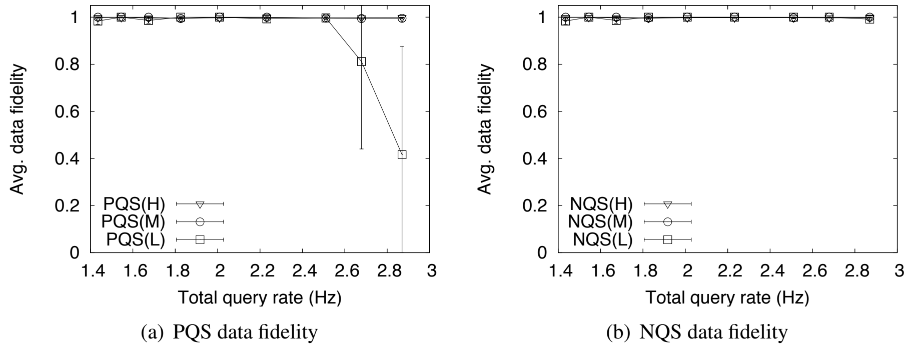
(a) 802.11e EDCA response time

(b) DRAND/DCQS response time

(c) PQS response time

(d) NQS response time

Figure 5. Response time of baselines, PQS, and NQS



(a) PQS data fidelity

(b) NQS data fidelity

Figure 6. Data fidelity of baselines, PQS, and NQS

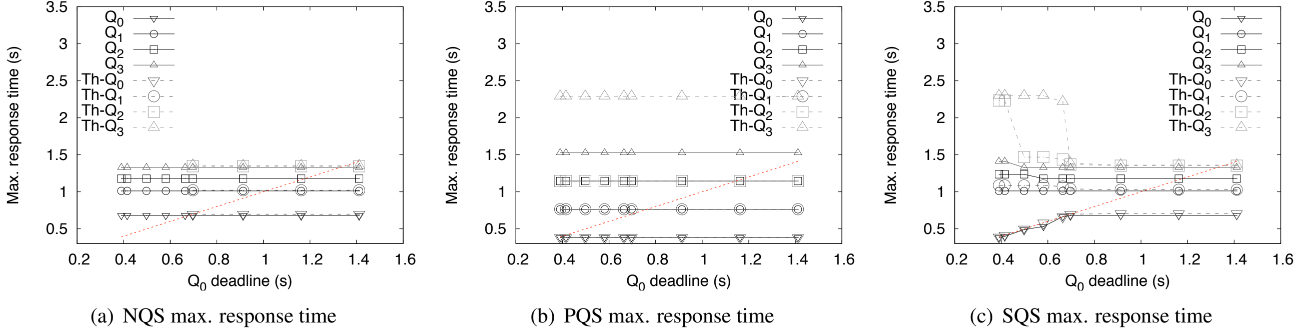
tention window, and maximum contention window of the CSMA/CA protocol. We configured these parameters according to their defaults in 802.11e. We used the 802.11e NS2 module from [11]. DRAND is a recently proposed TDMA protocol. DCQS is a query scheduling algorithm that constructs TDMA schedules to execute 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 (see Section 5.1) and evaluate the RTQS algorithms under different workloads and validate our response time analysis (see Section 5.2).

5.1 Comparison with Baselines

The results presented in this section are the average of five runs on different topologies. The 90% confidence interval of each data point is also presented. All experiments are performed in a 750m $\times$750m area divided into 75m $\times$ 75m grids in which a node is placed at random. We simulate three queries with high, medium and low priorities. The



(a) NQS max. response time

(b) PQS max. response time

(c) SQS max. response time

Figure 7. SQS adapts to different deadlines

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. The deadlines are equal to the periods.

Figs. 5 and 6 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. 5(a)). However, 802.11e EDCA has poor data fidelity for all queries (figure not included due to space limit). 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 Fig. 5(b)), have significantly higher data fidelity than 802.11e EDCA. DCQS provides lower response time than DRAND (see Fig. 5(b)) 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. 5(c)). PQS achieves close to 100% fidelity when the total query rate is lower than 2.51Hz (see Fig. 6(a)). For higher query rates, the fidelity drops because the offered load exceeds PQS's 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. 5(d)). In contrast to PQS, NQS has close to 100% data fidelity for all queries when the total query rate is as high as 2.87Hz. Therefore, NQS achieves higher throughput than PQS. The comparison of PQS and NQS shows the tradeoff between prioritization and throughput predicted by our analysis.

5.2 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.

Figs. 7(a) - 7(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. 7(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 throughput than PQS since it uses preemption only when it is necessary for meeting packet deadlines.

In all experiments, the measured response times of all RTQS algorithms are lower than the worst-case response times derived using our analysis. Hence, our analysis is correct. The difference between the simulation results and the theoretical bounds are expected because the analysis is based on worst-case arrival patterns which do not always occur in simulations.

6 Conclusions

High data rate real-time queries are important to many sensor network applications. This paper proposes RTQS, a novel transmission scheduling approach designed real-time queries in WSNs. We observe that there exists a 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 results 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.

Acknowledgement

This work is funded by NSF under ITR grant CCR-0325529 and under NeTS-NOSS grant CNS-0627126.

References

[1] G.-S. Ahn, A. T. Campbell, A. Veres, and L.-H. Sun. Swan: service differentiation in stateless wireless ad hoc networks. In *INFOCOM '02*.

[2] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 1993.

[3] B. D. Bui, R. Pellizzoni, M. Caccamo, C. F. Cheah, and A. Tzakis. Soft real-time chains for multi-hop wireless adhoc networks. *RTAS '07*.

[4] M. Caccamo, L. Y. Zhang, L. Sha, and G. Buttazzo. An implicit prioritized access protocol for wireless sensor networks. In *RTSS*, 2002.

[5] O. Chipara, C. Lu, and C.-G. Roman. Real-time query scheduling for wireless sensor networks. Technical Report WUCSE-2007-10, Washington University in St. Louis.

[6] O. Chipara, C. Lu, and J. A. Stankovich. Dynamic conflict-free query scheduling for wireless sensor networks. In *ICNP*, 2006.

[7] T. Facchinetti, L. Almeida, G. C. Buttazzo, and C. Marchini. Real-time resource reservation protocol for wireless mobile ad hoc networks. In *RTSS '04*.

[8] V. Kanodia, C. Li, A. Sabharwal, B. Sadeghi, and E. Knightly. Distributed multi-hop scheduling and medium access with delay and throughput constraints. In *MobiCom '01*.

[9] K. Karenos, V. Kalogeraki, and S. Krishnamurthy. A rate control framework for supporting multiple classes of traffic in sensor networks. In *RTSS*, 2005.

[10] 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 *ECRTS*, 2006.

[11] M. Lacage. Ns2 802.11b/e support. http://yans.inria.fr/ns-2-80211/.

[12] H. Li, P. Shenoy, and K. Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *RTAS '05*.

[13] K. Lorincz, D. J. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, 2004.

[14] C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *RTAS*, 2002.

[15] 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*, 38(2):91–128, 2006.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[17] R. Mangharam, A. Rowe, R. Suzuki, and R. Rajkumar. Voice over sensor networks. In *RTSS 06*, 2006.

[18] I. Rhee, A. Warrior, J. Min, and L. Xu. DRAND: Distributed randomized TDMA scheduling for wireless ad hoc networks. In *MobiHoc*, '06.

[19] G. Zhou, T. He, J. A. Stankovic, and T. F. Abdelzaher. RID: radio interference detection in wireless sensor networks. In *INFOCOM*, 2005.