Report Number: WUCSE-2007-10

2007

Real-time Query Scheduling for Wireless Sensor Networks

Octav Chipara, Chenyang Lu, and Gruia-Catalin Roman

Recent years have seen the emergence of wireless sensor network (WSN) systems that require high data rate real-time communication. This paper proposes Real-Time Query Scheduling (RTQS), a novel approach to conflict-free transmission scheduling for real-time queries in WSNs. We show that there is an inherent trade-off between prioritization and throughput in conflict-free query scheduling. RTQS provides 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... **Read complete abstract on page 2.**

Real-time Query Scheduling for Wireless Sensor Networks

Octav Chipara, Chenyang Lu, and Gruia-Catalin Roman

Complete Abstract:

Recent years have seen the emergence of wireless sensor network (WSN) systems that require high data rate real-time communication. This paper proposes Real-Time Query Scheduling (RTQS), a novel approach to conflict-free transmission scheduling for real-time queries in WSNs. We show that there is an inherent trade-off between prioritization and throughput in conflict-free query scheduling. RTQS provides 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. We provide schedulability analysis for each scheduling algorithm. The analysis and advantages of our scheduling algorithms are validated through NS2 simulations.

2007-10

Real-time Query Scheduling for Wireless Sensor Networks

Authors: Octav Chipara, Chenyang Lu, Gruia-Catalin Roman

Corresponding Author: ochipara@cse.wustl.edu

Abstract: Recent years have seen the emergence of wireless sensor network (WSN) systems that require high data rate real-time communication. This paper proposes Real-Time Query Scheduling (RTQS), a novel approach to conflict-free transmission scheduling for real-time queries in WSNs. We show that there is an inherent trade-off between prioritization and throughput in conflict-free query scheduling. RTQS provides 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. We provide schedulability analysis for each scheduling algorithm. The analysis and advantages of our scheduling algorithms are validated through NS2 simulations.

Type of Report: Other

Real-time Query Scheduling for Sensor Networks

Octav Chipara, Chenyang Lu, Gruia-Catalin Roman
Washington University in St. Louis

***Abstract*—Recent years have seen the emergence of wireless sensor network (WSN) systems that require high data rate real-time communication. This paper proposes Real-Time Query Scheduling (RTQS), a novel approach to conflict-free transmission scheduling for real-time queries in WSNs. We show that there is an inherent trade-off between prioritization and throughput in conflict-free query scheduling. RTQS provides 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. We provide schedulability analysis for each scheduling algorithm. The analysis and advantages of our scheduling algorithms are validated through NS2 simulations.**

I. Introduction

Recent years have seen the emergence of wireless sensor networks (WSNs) that must support real-time communication at high data rates. Representative examples include patient monitoring [1], mine worker search and rescue [2], and structural health monitoring [3]. Such systems pose significant challenges. First, the system must handle different 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 communication protocol must support *high throughput* since the system may generate high volumes of traffic. For example, during an earthquake, acceleration sensors may need to be sampled at high rates, generating high network loads when many sensors are deployed for fine-grained monitoring. Third, the systems we consider are typically mission critical requiring guarantees that packets are delivered before their deadlines. Therefore, the communication protocols must have *predictable temporal properties*.

A query service allows an application or user 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 novel approach to scheduling the transmissions of real-time queries in WSNs. In contrast to earlier work on TDMA scheduling for general-purpose wireless networks, RTQS takes advantage of the unique characteristics of WSN queries such as many-to-one communication, in-network aggregation, and periodic timing properties to construct conflict-free transmission schedules with differentiated and predictable temporal properties. This paper makes four contributions. First, through analysis and experiments, we show that query scheduling has an inherent tradeoff between prioritization and throughput. Second, we developed three scheduling algorithms which support preemptive, nonpreemptive, and slack stealing query scheduling. The nonpreemptive query scheduling algorithm achieves high throughput at the cost of some priority inversion while non-preemptive query scheduling algorithm achieves good prioritization by eliminating priority inversions. 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, by taking advantage of the predictability of the constructed schedules we derive theoretical upper bounds of the query latency of each scheduling algorithm to guarantee that the admitted queries meet their deadlines. Finally, through simulations, we show the advantages of RTQS over contention-based and TDMA-based protocols and we assess the tightness of the theoretical upper bounds on query latency.

The paper is organized as follows. Section II compares our approach to existing work. Section III describes the query and network models. Section IV details the design and analysis of RTQS. Section V provides simulation results. Section VI concludes the paper.

II. Related Work

Real-time communication protocols can be categorized into contention-based and TDMA-based protocols. In a contention-based approach, real-time communication is supported through probabilistic service differentiation. This is usually achieved by adapting various parameters of the CSMA/CA mechanism such as the contention window or initial back-off [4][5]. Rate and admission control [6][7] are often necessary to handle overload conditions in contention-based protocols. However, these protocols have two inherent drawbacks. First, packet latency is highly variable due to the random back-off mechanisms. Second, the throughput is low under heavy load due to high channel contention.

TDMA protocols can provide predictable packet latencies and support high data rates which makes them an attractive approach for real-time communication. The 802.15.4 standard for low data rate WSNs has a reservation mechanism for providing predictable delays in single hop networks. A more flexible slot reservation mechanism is proposed in [8] where slots are allocated based on delay or bandwidth requirements. Two recent papers proposed real-time communication protocols for robots [9][10]. 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 [11] provides prioritization in a one 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.

Two recent protocols that support real-time flows in WSNs have been proposed. In [2] a scheduling based solution is proposed to support voice streaming over real-time flows. In contrast, the real-time chains protocol [12] extends a contention-based scheme called Black Burst to support packet prioritization over real-time flows. 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 early work we proposed DCQS [13], 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.

III. System Models

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

A. Query Model

We assume a common query model in which source nodes produce data reports periodically. This model fits many applications that gather data from the environment at user specified rates. Such applications generally use existing query services such as TinyDB [14]. A query l is characterized by the following parameters: a function for in-network aggregation[14], the query period P_l, the start time of the query ϕ_l, a 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 u^{th} instance of query l whose release time is $r_{l,u} = \phi_l + u \cdot P_l$. For briefness, in the remainder of the paper we will refer to a query instance 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.

A query service usually works as follows: a user issues a query through a base station, which disseminates the query parameters to all nodes. To facilitate data collection and in-network aggregation, a *routing tree* rooted at the base station is maintained [14]. To perform in-network aggregation, each non-leaf node waits to receive the data reports from its children, produces a new data report by aggregating its data and the children's data reports, and then sends it to its parent.

B. Network Model

RTQS works by scheduling conflict-free transmissions in slots. To facilitate this we introduce the 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[15]. 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 [15], 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 is a fundamental service in WSN as many applications must time-stamp their sensor readings to infer meaningful information about the observed events. There exist several approaches for time synchronization in WSNs [2].

IV. Real-time Query Scheduling

RTQS supports real-time communication through conflict-free transmission scheduling that achieves predictable and differentiated query latencies. Our approach to query scheduling relies on two components: a *planner* and a *scheduler*. The planner constructs a *plan* for executing each query. All instances of a query are executed according to the same plan. 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 divides time into *slots*. The scheduler runs on every node to determine the slot when each step in a plan is executed. To improve the throughput, the scheduler may execute steps in the plans of different query instances in the same slot as long as no conflicting transmissions are executed in that slot.

RTQS works as follows: when a query is submitted, RTQS identifies a plan for its execution and its schedulability analysis. As discussed in Section IV-A, it is often the case that many queries can be executed using the same plan. Therefore, RTQS may reuse a previously constructed plan. When no plan may be reused, the planner constructs a new one. RTQS determines if a query meets its deadline using our schedulability analysis. If the query is schedulable, the parameters of the query are disseminated; otherwise, the query is rejected. At run-time the scheduler executes all admitted queries.

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 some priority inversion, while the *Preemptive Query Scheduling* (PQS) algorithm eliminates priority inversion for better prioritization. The *Slack-stealing Query Scheduling* (SQS) algorithm combines the benefits of NQS and PQS by improving the throughput while meeting all deadlines.

A. Constructing plans

A plan has two properties: (1) The plan 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 which are part of the routing tree while the dashed lines indicate interference edges. The plan in Fig. 1 is constructed assuming that the data report generated by a node can be transmitted in a single step. 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. The precedence constraints introduced by aggregation are respected. For example, nodes p and l are assigned in earlier steps than their parent o. In [13] 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. We omit the details of the algorithm due to space limit.

The plan of a query l depends only on the IC graph, the set of source nodes, and on l's aggregation function. The channel properties of a WSN remains stable within a time window as shown in a recent empirical study[16]. The issue of handling changes in the IC graph is discussed in Section IV-H. It is important to note that queries with the same aggregation function and set of sources but having different temporal properties (i.e., period, start time, deadline) or priorities can be executed according to the same plan. We note that real-time applications such as structural health monitoring often involve similar queries with different rates. Those systems therefore may use the same plan for multiple concurrent queries. Furthermore, even queries with different aggregation functions may be executed according to the same plan. Let $W_l[n]$ be the number of steps in which node n must be assigned to transmits l's data report. 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

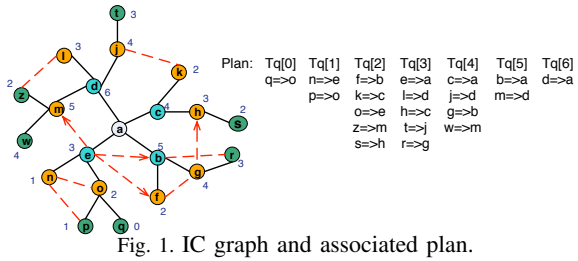


Fig. 1. IC graph and associated plan.

queries a node transmits one data report in a single step (i.e., $W_{max}[i] = W_{avg}[i] = 1$ for all nodes i) if the slot size is sufficiently large to hold two values. For the max query, the outgoing packet includes the maximum value of the data reports from itself and its children. For the average query, the packet includes the sum of the values and the number of data sources that contributed to the sum. Henceforth, we assume all queries share a same plan and minimum step distance when presenting the query scheduling algorithms and analysis. We discuss extensions to support queries with different plans in Section IV-G.

B. Overview of Real-time Query Scheduling

The scheduler executes a query instance according to the plan of its query. The scheduler improves the query throughput by executing multiple instances concurrently without conflict 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 properties 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. A nonpreemptive scheduler controls only the *start* of an instance; once an instance starts executing, a nonpreemptive scheduler cannot preempt it. In contrast, a preemptive scheduler may *preempt* an instance to allow a higher priority instance to execute when the two cannot be executed concurrently.

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 processing time of this approach is high, since each pair of steps must be checked for conflicts. Since the scheduler dynamically determines the steps that are executed in a slot, the scheduling algorithm must have low time complexity.

To reduce the time complexity of the scheduler we introduced the concept of *minimum step distance* in [13]. 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:

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

Therefore, to ensure that no conflicting transmission are executed in a slot, it is sufficient to enforce a minimum step distance between any two steps. Intuitively, the minimum step distance captures the degree of parallelism that may be achieved in query execution due to spatial reuse. In the worst case, when $\Delta(l,h) = L$, a single instance is executed at a time. A distributed algorithm for computing $\Delta(l,h)$ is presented in [13]. 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, despite the quadratic number of minimum step distances that must be stored the memory cost is small since the planner uses only few plans.

C. Nonpreemptive Query Scheduling (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 time of any two instances.

NQS maintains two queues: a *run* queue and a *release* queue. The *release* queue is a priority queue keyed by the query instance priority and contains all instances released but 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 operations). 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 time of any two instances by starting an instance in two cases: (1) when there are no instances 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 is 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 its execution 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. NQS provides prioritization by starting I_{hi} which is the highest priority instance in *release* in 8. Similarly, in slot 16, NQS starts I_{med} after I_{hi} completes $\Delta = 8$ steps.

```
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}}
  add I_{l,u} to all mayConflict[x] such that |I_{l,u}.i − x| < Δ
preempt(S):
  run = run − S; release = release ∪ S
  remove I_{l,u} from all mayConflict
may-resume(I_{l,u}):
  if (mayConflict[I_{l,u}.i] = ∅) 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 − Δ]=mayConflict[I_{l,u}.i − Δ + 1] − {I_{l,u}}
  mayConflict[I_{l,u}.i + Δ]=mayConflict[I_{l,u}.i + Δ] ∪ {I_{l,u}}
```

Fig. 2. PQS pseudocode

When a new instance is released, NQS inserts it in the *release* queue. Since the *release* queue is a priority queue which may be implemented as a heap, this operation takes $O(\log |release|)$. In each slot, NQS determines what instances should start executing. 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 send, receive, or sleep, NQS iterates through the instances in the *run* queue. This requires $O(|run|)$ time if each node maintains a bit vector indicating whether it transmits, receives, or sleeps in each step of a plan. Thus, the complexity of the operations performed in a slot is $O(|run|)$.

D. Preemptive Query Scheduling (PQS)

A drawback of NQS is that it introduces priority inversions. To overcome this, we devised PQS which preempts the instances that conflict with the execution of a higher priority instance eliminating priority inversions.

NQS's mechanism for enforcing the minimum step distance assumes that instances are not preempted. Therefore, we must derive a new and efficient mechanism for enforcing the minimum step distance that supports preemption for PQS. 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. In each slot, PQS determines the instances that will be executed and those that will be preempted in that slot and then executes the instance in the *run* 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 the 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 $I_{l,u}$ 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 other 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 instance 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 above, $I_{l,u}$ 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 *run* 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, *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 36 when *mayConflict*[4] becomes empty. The example shows that PQS achieves more effective prioritization than NQS by providing lower latencies for I_{hi} and I_{med}. However, the query throughput is lower because the degree of concurrency is lower (there is less overlap in the execution of instances). This exemplifies the fundamental tradeoff between prioritization and throughput in query scheduling. In the next section, we will characterize this tradeoff analytically.

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 **resume** and **preempt** functions take constant time since an instance $I_{l,u}$ may be a member of at most 2Δ *mayConflict* sets and Δ does not depend on the number of instances in *release* or *run*. 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 $O(|release| \cdot \log |run| + |run|)$.

E. Analysis of NQS and PQS

In this section, we present theoretical upper bounds on query latency for NQS and PQS. To determine if a query meets its deadline, we compute its worst-case response time, which is the maximum query latency of any of its instances. The base station calculates the worst-case response time of a query when it is issued. If a query's worst-case response time is smaller than its deadline then the query is schedulable and admitted for execution. We assume that the deadlines are shorter than the periods. For convenience, we use the slot size as the time unit and drop the instance count from our instance notation.

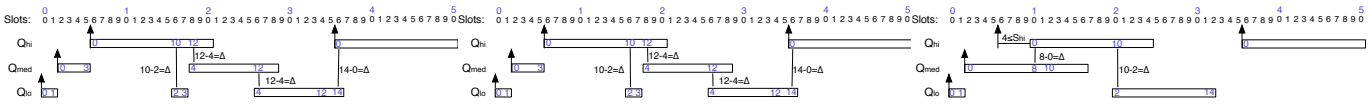
Analysis of NQS. Since NQS is a nonpreemptive scheduling algorithm, to compute the response time of a query l we must compute the worst-case interference of higher priority instances and the maximum blocking time of l due to the nonpreemptive execution of lower priority instances.

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 completed 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 can only be 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. ■



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

Fig. 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.

Note that since NQS is non-preemptive, the response time R_l of query l is the sum of its plan's length L (the execution time) and the worst-case delay W_l that any instance suffers (due to blocking and interference) before it is started: $R_l = W_l + L$.

To compute W_l we construct a recurrent equation similar to the approach used in response time analysis [17]. Consider the execution of an instance I_l. According to Property 1, a lower priority instance blocks I_l for at most $\Delta - 1$ slots. In addition, a higher priority instance I_h may delay the execution of I_l by at most Δ steps. 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 delay before I_l starts executing is:

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

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 [17].

Note that our analysis differs from the classical processor response time analysis in that multiple transmissions may occur concurrently without conflict in the 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[1]. 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 nonpreeptable 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:

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

where Δ is the length (execution time) of the preemptable part. After finding the worst-case interference, R'_l may be computed by solving (2) using a similar fixed point algorithm as the response time analysis [17]. Then, the worst-case response time of the query may be determined. Next, we determine the worst-case interference.

[1]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.

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 preempted it. We note that $x < \Delta$ since I_l is executing its preemptable part. There are three sub-cases. (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 $y < \Delta$ 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 $x < \Delta$ and $y < \Delta$, 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 cost 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.

F. Slack Stealing Query Scheduling (SQS)

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

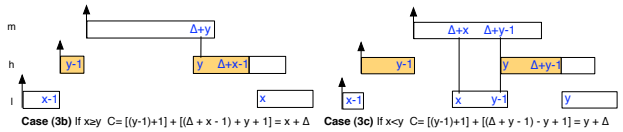


Fig. 4. Interference of I_h on I_l under PQS

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 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 executing at least $\Delta - S_h$ steps in their plan, SQS delays $I_{h,v}$ and adds it to the *release* queue. Otherwise, SQS adds $I_{h,v}$ to *release*, preempts all instances in *mayConflict*[0], and resumes the execution of the highest priority instance in *release* (which is not necessarily $I_{h,v}$).

Fig. 3(c) shows the schedule under SQS with the same workload as the one used to illustrate NQS and PQS. 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 6. This highlights that SQS dynamically decides when preemption is necessary 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 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 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:

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

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 $C_{max}(l,h)$ is the worst-case interference.

G. Handling Multiple Plans

We did not present our algorithms or analysis in the case when there are multiple plans due to the space limit. In the following we highlight the major changes necessary for handling multiple plans. When there are multiple plans the scheduler maintains a minimum step distance for each pair of plans. To ensure conflict-free transmission when there multiple plans, PQS starts the instance $I_{l,u}$ at the head of the *release* queue only if its step distance with *any* instance $I_{h,v}$ in the *run* queue is larger then $\Delta(l,h)$. To generalize PQS and SQS, we add a dimension to the *mayConflict* sets: we define *mayConflict*$[x][c]$ to include the instances $I_{h,v}$ in *run* which are executed according to plan c and conflict with any instance $I_{l,u}$ executing step x in its plan. The functions of PQS and SQS must be updated to consider C *mayConflict* sets instead of a single *mayConflict* set. We note that the time complexity of the algorithms does not change even when there are multiple plans.

To extended our analysis to handle multiple plans there are two key changes must be made. First, the interference and blocking terms must be computed in terms of pairs of plans. Second, for PQS and SQS, we split the execution of an instance $I_{l,u}$ into a preemptable part and a nonpreemptable part. The sizes of the two parts must be reevaluated. For PQS, the length of the preemptable part becomes $\Delta_M(l) = \max_x \Delta(l,x)$ since we must consider the worst-case minimum step distance for which an instance may interfere with one of l's instances. Accordingly, the length of the nonpreemptable part becomes $L - \Delta_M(l)$. Similarly, for SQS, the size of the preemptable part is $\Delta_M(l) - m_l$ while the size of the nonpreemptable part is $L - (\Delta_M(l) - m_l)$.

H. Handling Packet Loss and Topology Changes

RTQS can be extended to handle both transient and persistent packet loss. To handle persistent packet loss, the routing tree must be changed which in turn triggers a recalculation of the plans and minimum step distances. To reduce such reconfiguration overhead, we can modify the routing tree protocol to allow a node to have multiple parents and switch among them in response to packet loss. We modified the planner to construct plans as if a node transmits to all its parents even though in reality it transmits to only one parent at a time. As a result, RTQS can handle a range of topology changes without recomputing the plans or minimum step distances. Transient packet loss can be handled via Automatic Repeat-reQuest (ARQ) which RTQS supports by increasing the slot size to accommodate multiple transmissions. As in any other TDMA approach, this solution improves reliability at the cost of throughput due to the increased slot size.

V. 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 [15]. 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 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[13] 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 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 [19]. 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

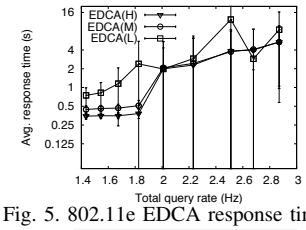


Fig. 5. 802.11e EDCA response time

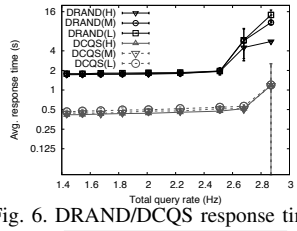


Fig. 6. DRAND/DCQS response time

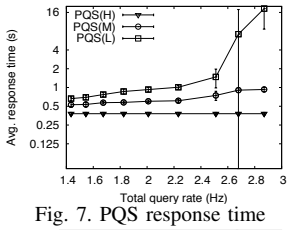


Fig. 7. PQS response time

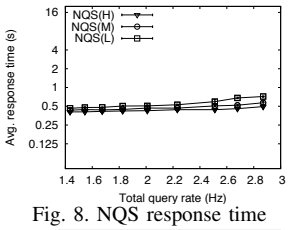


Fig. 8. NQS response time

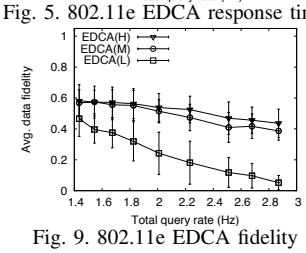


Fig. 9. 802.11e EDCA fidelity

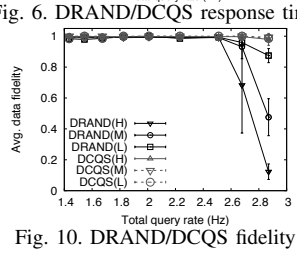


Fig. 10. DRAND/DCQS fidelity

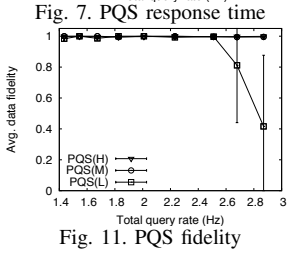


Fig. 11. PQS fidelity

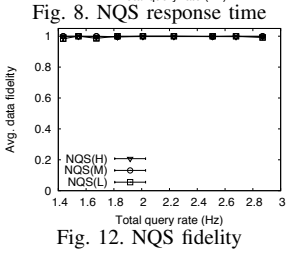


Fig. 12. NQS fidelity

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 V-A) and evaluate the RTQS algorithms under different workloads and validate our response time analysis (see Section V-B).

A. 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 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 1:2.2:4.7. The deadlines are equal to the periods.

Figs. 5 - 12 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). However, 802.11e EDCA has poor data fidelity for all queries (see Fig. 9). The poor performance of 802.11e EDCA is due to high channel contention, which results in significant packet delays and packet drops. This demonstrate the disadvantage of contention-based protocols for high data rate queries in WSNs.

The TDMA protocols, DCQS and DRAND (see Figs. 6 and 10), have significantly higher data fidelity than 802.11e EDCA. The data fidelity results indicate that DCQS provides a higher throughput than DRAND. Moreover, DCQS provides lower response time than DRAND (see Fig. 6). 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. 7). PQS achieves the same query throughput 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. 11). 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 8). 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.

B. Comparison of RTQS Algorithms

In this subsection we compare the performance of all RTQS algorithms and validate their response time analysis. In this section 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 1:1.2:2.2:3.2. To evaluate the RTQS algorithms under a broad range of workloads, we perform two experiments. In the first experiment, we fix the deadlines of the queries and vary their rates. In second experiment, we fix the rates of the queries and vary the deadline of the highest priority query.

Experiment 1. Figs. 13 - 15 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 throughput reduction. SQS achieves a slightly lower throughput than NQS because it is limited by the conservative response time analysis. When the admission algorithm decides that the queries are unschedulable, it cannot

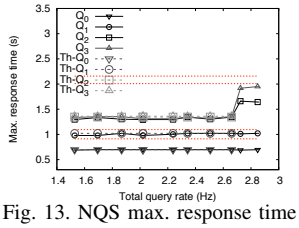


Fig. 13. NQS max. response time

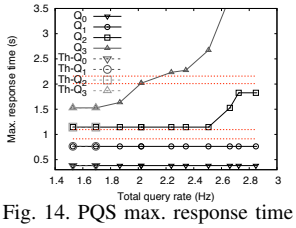


Fig. 14. PQS max. response time

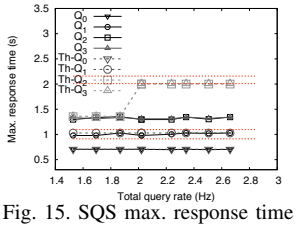


Fig. 15. SQS max. response time

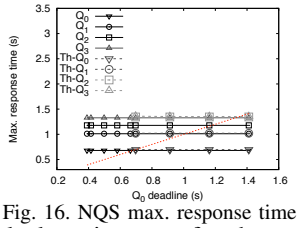


Fig. 16. NQS max. response time

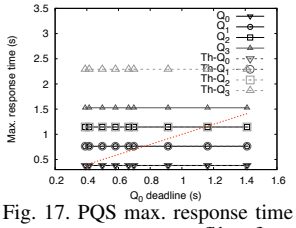


Fig. 17. PQS max. response time

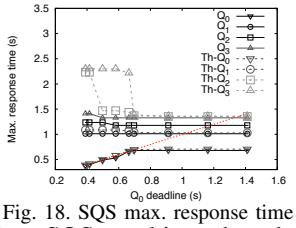


Fig. 18. SQS max. response time

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 throughput 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.

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. 16 - 18 show the maximum response times of NQS, PQS, and SQS, respectively. For clarity, only Q_0's deadline is plotted since the other queries always meet their deadlines in this experiment. PQS meets Q_0's deadline when it is 0.39s. In contrast, NQS meets its deadline only when Q_0's deadline reaches 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. 18). 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.

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.

VI. Conclusions

High data rate real-time queries are important services for a broad range of emerging sensor network applications. This paper proposes RTQS, a novel approach designed for real-time communication in WSN queries. We observe that there exists a tradeoff between throughput and prioritization under preemptive and non-preemptive 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 due to the high preemption cost in conflict-free query scheduling. 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 different workloads. The simulations also validate our worst-case response time analysis of each of the RTQS algorithms.

References

[1] 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*, vol. 3, no. 4, pp. 16–23, 2004.

[2] R. Mangharam, A. Rowe, R. Suzuki, and R. Rajkumar, "Voice over sensor networks," in *RTSS 06*, 2006.

[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, pp. 91–128, 2006.

[4] 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*.

[5] 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.

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

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

[8] 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.

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

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

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

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

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

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

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

[16] A. Meliou, D. Chu, J. Hellerstein, C. Guestrin, and W. Hong, "Data gathering tours in sensor networks," in *IPSN '06*.

[17] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.

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

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