Distributed Cross-Layer Scheduling for In-Network Sensor Query Processing

Hejun Wu Qiong Luo Wenwei Xue

Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{whjnn, luo, wwxue}@cs.ust.hk

Abstract

In-network sensor query processing is a cross-layer design paradigm in which networked sensor nodes process data acquisitional queries in collaboration with one another. As power efficiency is still one of the most severe constraints in this paradigm, we propose a distributed, cross-layer scheduling scheme for it. In this scheme, each node employs its MAC, routing, and query layers to negotiate with its parent its timing for transmission and constructs a schedule for its query processing. It then follows the schedule to compute, communicate, and sleep in each query processing cycle. This scheduling reduces wasted listening and receiving as well as the switching between active and sleeping modes. Consequently, it results in 50-60% of power saving on real sensor nodes in our experiments. Additionally, it outperforms two existing scheduling schemes both on schedule construction efficiency and on schedule quality.

1. Introduction

In-network sensor query processing systems such as Cougar [23] and TinyDB [12][13] are promising for data acquisitional applications of wireless sensor networks (WSNs) [1]. With these systems, a user injects SQL-style queries such as "*select temperature from sensors*", or "*select avg(light) from sensors*" into the network through a PC. The networked sensor nodes then work together to process the queries and send results back to the PC. This in-network query processing paradigm is more efficient and flexible than centralized query processing [1]. Nevertheless, power consumption remains a critical issue in these systems [2][3]. In this paper, we propose a distributed, cross-layer scheduling scheme for in-network sensor query processing to address the power efficiency problem.

An immediate solution to reducing the power consumption is to make the nodes sleep as much as possible [9][12][13][16][24]. Along this direction, a number of sleep scheduling schemes have been adopted to enable nodes to sleep periodically during sensor query processing [13][24]. These schemes roughly schedule active and sleeping modes at a certain layer of a WSN but do not use cross-layer information to construct a complete query processing schedule. There are other schemes that schedule the communication timings of nodes during sensory data collection [7][9][16], but they also ignore the operations in query processing such as query injection, computation, and aggregation.

To further optimize the power efficiency of in-network sensor query processing, we propose a distributed cross-layer scheduling scheme. It involves the interaction of the three layers, namely, the medium access control (MAC) layer [20][24], the routing layer [8][10][21], and the query layer [13][23]. As the query processing systems usually use tree networks [12][13][23], we focus on tree networks in this paper. Similar to the existing schemes, our scheme aims to reduce the energy waste [24] in WSNs.

It has been well established that *idle listening* [11][20], *overhearing*, *collision,* and *control packet overhead* are major sources of energy waste. Among these four sources, idle listening and overhearing are the dominating ones [17]. In our previous experience with the MICA2 [6] networks, these two factors cost more than 70% of the power during query processing [22]. Therefore, in our scheduling scheme, we focus on reducing idle listening and overhearing and identify a number of constraints for schedule construction to reduce collision.

Specifically, in our scheme a node first checks what transmission timing is applicable in its query processing cycle and sends this information to its parent. This applicable transmission timing is the possible time within which the node can transmit. Then, the parent sends the node the assigned transmission timing based on the applicable transmission timing information received. Next, the node arranges its schedule for its other query processing tasks in all layers. Finally, the node starts to follow its schedule for query processing and it no longer needs to listen or to send control packets before transmission. These steps require the information about neighbors only. Without causing confusion, we refer to the *neighbors* of a node as the nodes within its transmission range, including its parent and children.

We have implemented our scheduling scheme on real sensor nodes by modifying the source code of TinyOS and TinyDB [19]. We have also performed initial experiments on real sensor nodes as well as on an emulator to evaluate our scheduling scheme. Additionally, we have conducted simulation studies to compare our scheme with two other existing scheduling

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)
0-7695-2518-0/06 $20.00 © 2006 IEEE

schemes. The results show that our scheme significantly improves the power efficiency of in-network sensor query processing.

The remainder of this paper is organized as follows. Section 2 reviews the related work on scheduling schemes in WSNs. Section 3 gives an overview of our scheduling module. Sections 4 and 5 present the design and implementation of our proposed scheme. We discuss our experimental results in Section 6 and conclude in Section 7.

2. Related Work

Scheduling has been applied to different layers of a WSN to enable a sensor node to sleep without affecting its other activities. For instance, TinyDB schedules at the query layer: a node wakes up at the start time of each sample interval [13] and keeps active for a fixed period, typically four seconds [3]. S-MAC schedules at the MAC-layer: a node sleeps for some time, and then it wakes up and listens to the wireless channel [24]. As these scheduling schemes only roughly control the active and sleeping timings, they do not reduce the energy waste significantly. In the following, we discuss several scheduling schemes that operate at a finer granularity and are closely related to ours.

Florens et al. studied a centralized scheduling algorithm for data distribution and collection in WSNs [7]. In their algorithm, the sink allocates the transmission timing to all other nodes in a WSN. This centralized nature requires the sink to know the current network topology, which is usually difficult in practice. Furthermore, it is often costly to disseminate schedules from the sink to all other nodes in a network.

To avoid the problem in centralized scheduling, a number of distributed scheduling schemes have been proposed. A representative scheme is Flexible Power Scheduling (FPS), which is a distributed on-demand power-management protocol for tree networks [9]. In this protocol, a parent randomly chooses *reserved slots* and broadcasts the reserved slots. A child sends a request for a specific reserved slot if it has some message to send, and the parent confirms the request if the slot has not been requested by other children. The protocol helps reduce the collision between children of the same parent, a.k.a., *siblings*.

However, one problem of FPS is that it does not reduce the collisions between neighbors that are not siblings. Suppose two nodes *A* and *B* are neighbors, and node *A* receives at times 1 and 2 from its children. If node *B* transmits at time 1 or 2, collision will occur at node *A* at time 1 or 2. In such scenarios, collisions may become even worse with scheduling than without.

Another example of distributed scheduling is a scheme proposed by Sichitiu [16]. In this scheme, a source node first broadcasts a special route setup packet, RSETUP, to set up a route and a temporary schedule with a neighbor. If the RSETUP packet finally arrives at the sink, the nodes along the path will set their temporary schedules to be permanent schedules; otherwise, the temporary schedule of the source node will be removed. If collision occurs during this setup process, the RSETUP packet will be postponed to the earliest time when the node does not transmit or receive.

Nevertheless, Sichitu's scheme does not consider the scenarios in which nodes may not get a schedule. Because the nodes are synchronized under the scheduling scheme and their tasks are similar, collisions may occur frequently. Consequently, even after the RSETUP transmission of a node is postponed, collisions may still occur when the transmission is started again. In our experience with real WSNs, we often find that the sink cannot receive data from some nodes. These nodes are called *dead nodes*. The query result accuracy in sensor query processing is low if the number of dead nodes is large.

In consideration of the existing schemes and our goal of improving the power efficiency of in-network sensor query processing, we design our scheme to have the following three unique features. First, our scheme considers all tasks in query processing, including transmission as well as query injection, computation and in-network aggregation. Second, our scheme attempts to allocate consecutive sleeping and transmission timings to nodes. This consecutiveness helps save power; otherwise, frequent switching of running modes would increase power consumption [25]. Third, all layers of a query processing system are involved in the schedule construction and execution. This cross-layer design improves the efficiency of the resulting schedule as there is detailed timing information for each operation.

3. System Overview

A typical WSN setup for in-network sensor query processing is shown in Figure 1. In this WSN, every node has a unique ID. The server is used to post queries to the sensor network and to receive query results from the network via the sink connected to it. The sink forwards commands and queries to the sensor nodes. The sensor nodes process the queries and generate query results. Finally, query results are forwarded towards the sink, which in turn forwards these results to the server.

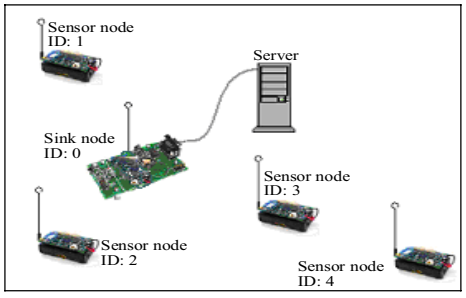


Figure 1. A typical WSN setup

In such a WSN, data acquisition is usually done in the form of continuous queries [13], which collect data at a fixed sample interval and repeat for a long time. Due to their long running nature, these continuous queries benefit more from a good schedule than snapshot queries, which return results only once. In this paper, we focus on continuous queries.

The architecture of a query processing system with our scheduling module is shown in Figure 2. The scheduling module is illustrated in the dashed box on the left. The system works as follows. First, the routing layer constructs a routing tree. Then, the sink may inject a query into the network. A node starts to construct a schedule when it receives a query. When all nodes involved in the query have constructed their schedules, the sink broadcasts a synchronization signal. In turn, these nodes synchronize with the sink and start to run following their schedules. During this process, both the construction of the routing tree and that of the schedules are reported from bottom up to the sink. When all children of the sink have reported, the sink knows that the routing tree or the schedule construction has completed.

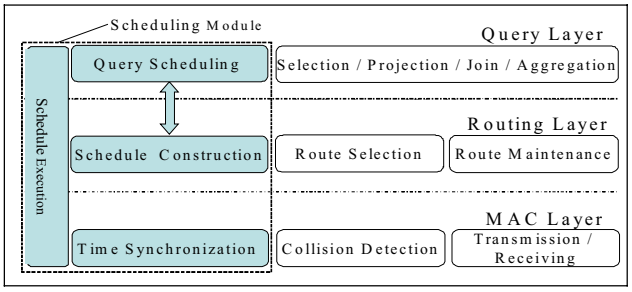


Figure 2. Architecture of scheduling module

In the system, the scheduling module is mainly responsible for three tasks. First, schedule construction. Even though schedule construction is performed at the routing layer, it requires information from both the query layer and the routing layer. Second, time synchronization. Time synchronization is done at the MAC layer. We adopt the synchronization mechanism designed by Su Ping, which has an accuracy of a few milliseconds [15]. Since the time unit is at the level of a hundred milliseconds in our scheme, this accuracy is sufficient. Third, schedule execution. The execution of the schedule on a node is to control the timing for each task at all three layers.

We use a *slot* as the time unit in a schedule and number slots in the form of periodic modular *m*, if a sample interval has *m* slots [9]. Specifically, the slot number *s* of the slot that starts at time *t* is computed in Equation (1), given the slot length *ls*, the schedule start time t_0, and the number of slots *m* in a sample interval.

$$s = \left\lfloor \frac{t - t_0}{ls} \right\rfloor \bmod m \qquad (1)$$

We define the length of a slot as the period within which the largest data packet can be successfully transmitted out of a node. Consequently, a slot assigned to a task in query processing may be longer than needed. Nevertheless, the power consumption caused by this difference is little (at the micro-joule level on MICA2 motes). Furthermore, this additional time can be used to tolerate the time synchronization errors.

Depending on the operations in a task, a slot can be a *sleeping* slot, a *transmission* slot, a *processing-listening / receiving* (*PL/R*) slot, or a *query injection / maintenance* (*Q/M*) slot. In a *transmission* slot, a node sends a packet. In a *PL/R* slot, a node listens to the wireless channel and receives data packets, if any. In a *Q/M* slot, a node listens to the channel for a new query, or sends or receives route maintenance packets. Most routing protocols need such route maintenance packets to manage routes between nodes [4][10].

In the following, we present schedule construction and execution in detail. We omit the details of time synchronization as it is not the focus of this paper.

4. Schedule Construction

Figure 3 shows the schedule construction module. The algorithm *NegotiateTransmissionSlots* negotiates the transmission slots for a node. It has two sub-procedures: *DetermineApplicableTransmissionSlots* that determines the applicable transmission slots for the node, and *AllocateChildrenTransmissionSlots* that allocates the transmission slots to the children of the node. After the transmission slots are allocated for a node, *ConstructSchedule* constructs the schedule on the node.

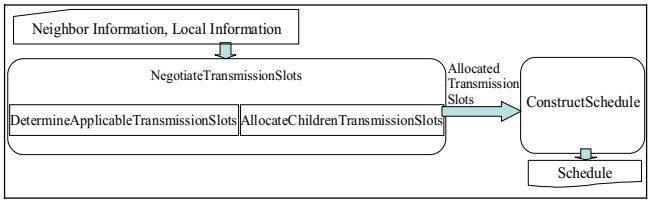


Figure 3. Schedule Construction

This schedule construction module attempts to follow a number of constraints so that in the resulting schedule query results can be successfully transmitted to the sink. These constraints are listed in Section 4.1.

4.1. Constraints for Schedule Construction

We formulate the following four constraints for schedule construction.

(i) Neighbor nodes have different transmission slots.
(ii) Siblings have different transmission slots.
(iii) For any two neighbor nodes A and B, all children of node A have different transmission slots from node B.

(iv) The transmission slots of a parent node in each sample interval are later than its children.

Constraint (i) ensures that the signals from a sender do not clash with its neighbors'. However, if siblings are not neighbors, they may simultaneously send packets to their parent, where collision would occur. Constraint (ii) is to avoid this kind of collision. Constraint (iii) prevents another kind of collision, illustrated in Figure 4. In this figure, *A* and *B* are neighbors, and *D* is the child of *A*. As B and *D* may not be neighbors, it is possible that *B* and *D* transmit simultaneously, which causes collision at node *A*. In addition to the first three constraints, Constraint (iv) is necessary to allow a parent to merge or to aggregate the query results from its children. However, if there is no packet merging or aggregation at the parent, Constraint (iv) is unnecessary.

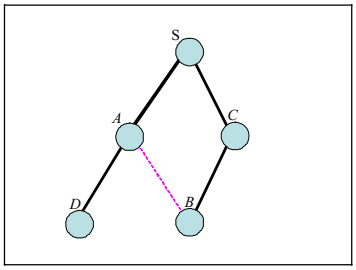


Figure 4. An example for Constraint (iii)

All of these constraints focus on collisions at the parent node of a sender instead of completely collision-free schedules because it is impractical for a WSN to set up completely collision-free schedules in a distributed way due to the high overhead. Moreover, most of the times completely collision-free schedules are unnecessary for query processing systems in practice; these systems can work well as long as the query results of a node can be successfully received by its parent. An example of this non-parent collision is shown in Figure 5. In this example, nodes *D* and *E* are both neighbors of *B*. If *D* and *E* transmit simultaneously, *B* may receive colliding messages from *D* and *E*. However, the parents *A* and *C* can still get correct messages since collisions do not occur at nodes *A* and *C*.

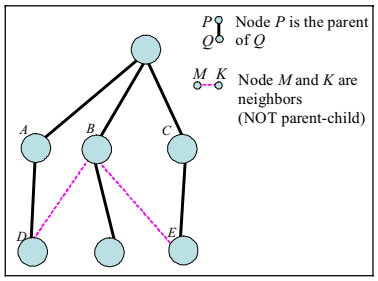


Figure 5. An example of the collision

Next, we prove that these constraints are sufficient to ensure no collisions occur at the parents of the senders.

Property 1. *Constraints (i) - (iii) ensure that no collisions occur at the parent of a sender during transmission.*

Proof. We prove by contradiction. Suppose a collision occurs at the parent *P* of a sender *S* and the four constraints are observed. By the definition of collision [18], there are two possible causes: (1) several neighbors of *P*, including *S*, are sending to *P* simultaneously, or (2) nodes *S* and *P* are sending simultaneously.

Assume that the collision occurs due to the first cause. If among all sending neighbors, only sender *S* is a child of *P*, then *S* has the same transmission slot as some neighbors of *P*, which violates Constraint (iii). Otherwise, it violates Constraint (ii). Hence, the first cause is impossible as long as the constraints are observed.

The second cause contradicts with Constraint (i) since nodes *S* and *P* are neighbors. ■

4.2. Negotiation of Transmission Slots

Following the constraints, we design the algorithm in the scheduling module to reduce the collisions. Algorithm 1 shows *NegotiateTransmissionSlots* in this module, which runs on each node. It has two inputs. One is the table of neighbor information, denoted as *NbrTbl*. This table stores the information of a neighbor node such as its node ID, hop count, parent node ID, timestamp of the last received route maintenance packet, and a flag indicating whether it is a child of the node.

Algorithm 1 . NegotiateTransmissionSlots

```
Input: NbrTbl , LocalInf
Output: TSI
1: switch (event)
2:   case: ScheduleTimer fired
3:     if ATS == null & (IsLeaf() or CTS != null)
4:       ATS= DetermineApplicableTransmissionSlots
                                        (NbrTbl , LocalInf)
5:       if ATS != null
6:         if !IsSink()
7:           send ATS to parent
8:         else
9:           TSI = AllocateSinkTransmissionSlots(ATS)
10:          output TSI
11:    if HasChildren() & CTS == null
12:      CTS = AllocateChildrenTransmissionSlots
                                        (NbrTbl , LocalInf, CATS)
13:      if CTS !=null
14:        send CTS to children
15:  case: received TSI_Q from a non child / parent neighbor Q
16:    mark TSI_Q as non-applicable slots for transmission
17:  case: received ATS_K from a child K
18:    insert ATS_K to CATS
19:  case: received CTS_P from the parent node P
20:    extract TSI from CTS_P and broadcast TSI to neighbors
21:    output TSI
```

The other input, denoted as *LocalInf*, is the information of the node itself, such as the node ID, the parent node ID and the hop count. The output is the information of the allocated transmission slots of the node.

There are four major variables in Algorithm 1. *Applicable Transmission Slots* (ATS) describes all possible slots for the node to transmit packets. *Children Applicable Transmission Slots* (CATS) is the set of the ATSs of all children of the node. *Children Transmission Slots* (CTS) are the slots allocated by the node to its children (if any), and the *Transmission Slot Information* (TSI) describes the transmission slots allocated by the parent of the node. These four variables are all initialized as *null*.

Algorithm 1 uses a timer called *ScheduleTimer,* which signals a timer event every time interval T_{sh}. T_{sh} is set to be *MAX_NBR_NUM* * *ls*, where *MAX_NBR_NUM* is a constant, the maximum number of neighbors of a node, and *ls* is the length of a slot. The length *MAX_NBR_NUM* * *ls* allows a node to have sufficient time to receive the packets containing the information of the transmission slots from its neighbors.

When there is a *ScheduleTimer* event, if the applicable transmission slots of the node are not determined yet, and if the node (1) is a leaf node or (2) has successfully allocated transmission slots to its children, *DetermineApplicableTransmissionSlots* is called to determine the applicable transmission slots (ATS) for the node. When the applicable transmission slots are successfully determined, the node sends them in a packet to its parent. After a parent node has received all applicable transmission slots of its children, *AllocateChildrenTransmissionSlots* is called to attempt to allocate the transmission slots for the children. If the allocation is successful, the parent sends the allocated transmission slots (CTS) to its children.

When a node receives the allocated transmission slots from its parent, Algorithm 1 on this node outputs the allocated transmission slots (TSI) and broadcasts them. If another node who is not the broadcasting node's child or parent receives this information, it marks these slots as non-applicable slots for transmission (Line 16). That is, it will not use these slots as transmission slots for its children or as applicable transmission slots for itself. Since the sink node does not have a parent, it allocates the transmission slots for itself when it has finished the allocation of the transmission slots to all its children (Line 9).

For reliability, a node replies an ACK when it receives the applicable transmission slots from its child or the allocated transmission slots from its parent.

4.2.1. The Sub-Procedures

Procedures 1 and 2 show two sub-procedures used in Algorithm 1, *DetermineApplicableTransmissionSlots* and *AllocateChildrenTransmissionSlots*.

Procedure 1 makes a node wait for two types of neighbors that have not been allocated transmission slots: (1) lower hop neighbors (Line 4); or (2) non-sibling neighbors at the same hop but with a smaller node ID than this node (Line 9). Lower hop nodes are the nodes that have larger hop counts than this node. This waiting mechanism makes the neighbors start at different times to determine their applicable transmission slots. The reason is that if the neighbor nodes start this process simultaneously, there may be conflicts in their applicable transmission slots. However, this waiting mechanism may cause deadlocks due to node failure or cyclic waiting. To break the deadlocks, Procedure 1 uses two timeouts, *HTimeout* and *NTimeout*. These timeouts will be discussed in detail in Section 4.2.2 together with those timeouts used in Procedure 2.

Procedure 1 . DetermineApplicableTransmissionSlots

Input: NbrTbl , LocalInf

Output: ATS if successful; otherwise null

```
1: HTimeout - -
2: if HTimeout > 0
3:    for (i =0; i< NbrTbl.Size; i++) do
4:       if IsLowerHopNbr( NbrTbl[i]) & !NbrTbl[i].TSI
5:          return null
6: NTimeout - -
7: if NTimeout > 0
8:    for (i =0; i< NbrTbl.Size ; i++) do
9:       if IsSameHopSmallIDNbr(NbrTbl[i]) & !NbrTbl[i].TSI
10:         return null
11: NeededPLRSlots = QueryLayer -> GetNeededPLRSlots()
12: ATS = FindAvailableSlots (NeededPLRSlots, LocalInf)
13: return ATS
```

Using *NeededPLRSlots* and the known allocated transmission slots of the neighbors, the function *FindAvailableSlots* applies the four constraints to determine the applicable transmission slots. *NeededPLRSlots* is acquired from the query layer via calling *QueryLayer->GetNeededPLRSlot*, which outputs all slots needed in processing and receiving. To follow Constraint (iv) in Section 4.1, *FindAvailableSlots* starts searching for the available transmission slots that are later than the latest allocated transmission slot of the children. As illustrated by the example in Figure 6, the operations of determining the applicable transmission slots are similar to memory allocation. In Figure 6, because slot 7 should be reserved for query processing as required by the query layer, the applicable transmission slots are 8-12.

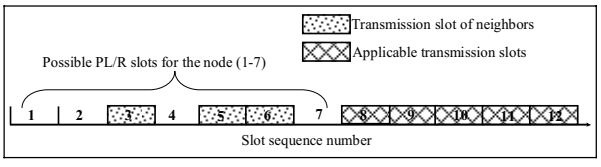


Figure 6. Search applicable transmission slots

Procedure 2 allocates transmission slots for the children of a node. It makes the node wait for lower hop non-children neighbors that have not been allocated transmission slots yet (Line 4). This is to ensure the transmission slots are allocated in the bottom-up order, to observe Constraint (iii) in Section 4.1. In addition, Procedure 2 does not allocate the transmission slots to the children until the parent receives all of the applicable transmission slots (*ATS*) from its children or *CTimeout* is decreased to 0. Two reasons are involved. First, it enables the children to receive their allocated transmission slots in a single message to save the communication cost. Second, it allows the parent to allocate consecutive transmission slots to its children.

Procedure 2. AllocateChildrenTransmissionSlots

```
Input: NbrTbl , LocalInf, and CATS
Output: CTS if successful; otherwise null
1:  PTimeout - -
2:  if PTimeout > 0
3:     for (i =0; i< NbrTbl.Size; i++) do
4:        if IsLowHopNonChildNbr(NbrTbl[i]) & !NbrTbl[i].TSI
5:           return null
6:  CTimeout - -
7:  if not received ATS of all children and CTimeout> 0
8:     return null
9:  CTS = FindAvailableChildrenSlots (CATS, LocalInf)
10: return CTS
```

The function *FindAvailableChildrenSlots* in Procedure 2 allocates all children of a node with transmission slots in consideration of their applicable transmission slots and the allocated transmission slots of the neighbors of the node. The constraints are observed during the search for available slots. Figure 7 shows an example of searching available slots for a child. In Figure 7, the applicable transmission slots of the child are 8-12 and the child needs two transmission slots. The parent starts searching from 8 and finds that slots 8 and 9 are occupied by the neighbors of the node. Therefore, the node allocates slots 10-11 to its child.

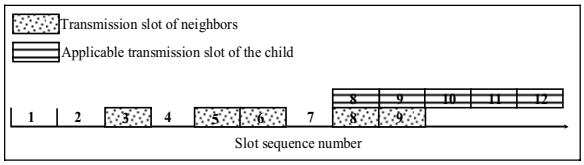


Figure 7. Transmission slots of a child

4.2.2. Timeouts in the Sub-Procedures

Because a node waits for some nodes to be allocated transmission slots in Procedure 1 or 2, deadlocks may occur due to node failure or circular waiting between neighbors. To break deadlocks, we use four timeouts in these two procedures. Table 1 shows these timeouts.

HTimeout and *NTimeout* in Table 1 are used in Procedure 1. *HTimeout* is designed for breaking the deadlocks in waiting for lower hop nodes. Its value is calculated using the hop count of the node and the maximum number of hops (*MaxHop*). *MaxHop* is obtained by making nodes broadcast their newest knowledge about the maximum number of hops in the network. With this initialization, the difference between the *HTimeout*s of nodes at two consecutive hops is *MAX_NBR_NUM*. This difference allows lower hop nodes to receive sufficient information from neighbors so that they can finish their schedule construction before upper hop nodes start the schedule construction.

Table 1. Timeout variables

HTimeout	(MaxHop - HopCount) * MAX_NBR_NUM
NTimeout	NodeID
PTimeout	NodeID
CTimeout	max (ChildrenNodeID)

We set the value of *NTimeout* as the node ID. The motivation is to make nodes time out at different times so that they do not compete with each other when they determine their applicable transmission slots.

The two timeouts in Procedure 2 are *PTimeout* and *CTimeout*. The value of *PTimeout* is also the node ID. Similar to *NTimeout*, this value enables the neighbors to start the allocation of transmission slots for their children at different times. The value of *CTimeout* is the largest of the node IDs of the children. This value allows the parent to wait for the child with the largest node ID, which has the largest *NTimeout*.

In summary, the timeouts enable nodes to get allocated transmission slots at different times and to allow neighbors to exchange the information about their slots. Hence, the transmission slots of nodes rarely have conflicts.

4.3. Complete Schedule for Query Processing

With the allocated transmission slots, the procedure *ConstructSchedule* arranges the time slots for PL/R, Q/M, and sleeping of a node. The PL/R slots are arranged to be sooner than the transmission slots so that the query result is ready for transmission. The Q/M slots are located before the end of every sample interval. This arrangement of Q/M slots is to allow the nodes to finish the transmission of the query results before they receive new queries.

For reliability, *ConstructSchedule* is equipped with the following two mechanisms. First, there are two buffer slots before the PL/R slots in the schedule. These buffer slots are used to accommodate synchronization inaccuracy in query processing. Second, the sink is allowed to be always active or to have a schedule similar to other nodes, depending on if it is powered by an external outlet or by battery. This differentiation is to allow other nodes to receive a query from the server as soon as possible with the sink's power consumption considered.

An example of complete schedules for a simple data collection query in a 4-node and 3-hop WSN is illustrated in Figure 8. Nodes 1 and 2 are leaf nodes at the same hop count. Node 3 is the parent of nodes 1 and 2, and needs three transmission slots to transmit the query results of nodes 1 and 2 and its own. The sink is always active since it is powered by an external outlet.

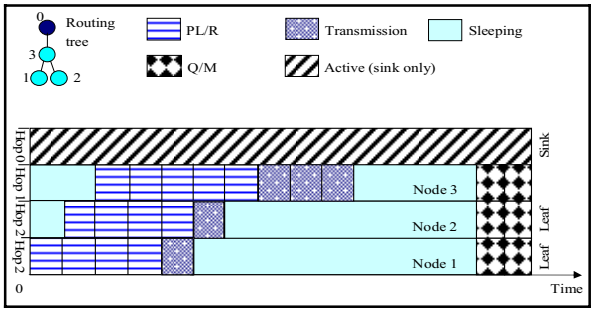


Figure 8. An example of complete shedules

In addition to constructing a single query schedule as shown in Figure 8, our scheme can construct schedules for multiple queries. The multi-query construction process runs query by query. After the schedule for one query is constructed, the nodes will construct the schedule for the next query, until the schedules of all queries are constructed. During this process, the algorithm will consider only the sleeping slots in the existing schedules as applicable slots for transmission and receiving.

5. Schedule Execution

After schedules are constructed on nodes involved in query processing, the sink starts time synchronization. The nodes begin to execute their schedules as soon as they finish the time synchronization. The execution of a schedule involves the timing control of all layers in a WSN.

We adopted the query layer of TinyDB and modified the service scheduler of TinyDB so that the scheduler can roughly control the timing of query result transmission. This timing control is necessary because the MAC layer of a WSN does not allow long delays.

In our scheme, the MAC layer first checks whether it is time for transmission when it receives a packet from the query layer through the routing layer. If the time is earlier than the allocated transmission time, the MAC layer copies the message to a memory buffer, and sets up a timer to automatically transmit the message when it is time for the transmission. Note that, the interval of waiting should be shorter than the interval between two transmissions. Otherwise, the buffer may be overwritten by another message. Therefore, the query layer is designed to roughly control the transmission timing. With this rough timing control in the query layer, the delay in the MAC layer will not cause the overwriting problem in the memory buffer.

The task of the routing layer in the schedule execution is to control the timing of transmission and receiving the route maintenance messages. The timing control process in the routing layer is similar to that in the query layer.

6. Evaluation

To evaluate our scheme, we first used simulation to compare the schedules of our scheme (denoted as *DCS*) with those of Flexible Power Scheduling (*FPS*) [9] and Sichitiu's Scheduling (*SS*) [16]. Choosing simulation is for a fair comparison of the schemes since Sichitiu's scheduling scheme is not applicable to the MICA2 motes [6] due to the memory limitation of the motes.

We then used a real MICA2 sensor network and an emulated network to measure the performance of our scheme. The real sensor network enabled us to study the applicability and power consumption of our scheme in real world in-network query processing. VMNet, on the other hand, can provide us detailed runtime information about the neighbors of each node, the routing tree, and the schedules in a network.

Our scheme was implemented into TinyDB, a well-known query processing system [12][13]. We denote TinyDB with our scheme *Optimized TinyDB* and the original version of TinyDB *Original TinyDB*. FPS and SS are not applicable to query processing systems, as they do not support queries but only schedule the communication from sensor nodes to the sink. Hence, we studied the performance improvement on query processing for our scheme only.

6.1. Scheme Comparison

6.1.1. Simulation Setup

To compare the schemes, we used the same experimental setup as that Sichitiu used [16]. The simulated network consisted of 100 nodes randomly deployed in an 80m*80m rectangular area and the transmission range was 25m. Although the nodes were randomly deployed, there was at least one path from each node to the sink in the simulated network. For fairness, we used the same slot length in FPS and SS as in ours. In our measurement, the time required for transmission the largest data packet on MICA2 motes was about 80ms. Therefore, we set the slot length to be 120ms, which is sufficient to tolerate a time difference of ± 20ms in synchronization.

Since FPS and SS are inapplicable to query processing, we ran the three schemes to construct schedules for a simple data collection application that sampled the temperature of each node every 60 seconds. In this application, we disabled the allocation of slots for

the computation of query processing in our scheme since this application did not have a query layer.

6.1.2. Scheduling Overhead

First, we measured the time of constructing a schedule in the three schemes. We regard the schedule construction as finished when the schedule has been unchanged for 10 simulated days (i.e., the time in the simulated nodes elapsed 10 days). The schedule construction time was the interval from the start time of the schedule construction to the time the schedule was last updated. The results are shown in Table 2.

Table 2 shows that SS took the longest time to construct a schedule whereas FPS the shortest. This result is because a node running SS cannot get a schedule until the sink sends an ACK to the node, which takes a long time especially when the node is far from the sink. As a node running our DCS needs to wait for some nodes, the construction time is also longer than that of FPS. FPS is the simplest among the three and omits some constraints considered in our scheme. Consequently, it is the fastest in schedule construction.

Table 2. Scheduling overhead

Scheme	Schedule construction time (seconds)
FPS	420
SS	7140
DCS	900

6.1.3. Schedule Comparison

To thoroughly compare the constructed schedules, we chose two metrics that are directly related to the performance of WSNs.

The first metric is the *number of dead nodes.* A dead node is one from which the sink never receives packets, either directly or indirectly. As sensory results are obtained from individual nodes in a network, the number of dead nodes affects the accuracy of query results. Dead nodes can be further divided into two classes. The first class is *non-scheduled* nodes, which have no allocated transmission slots in a schedule. The second class is *conflict-scheduled* nodes, which have allocated transmission slots but these slots conflict with those of their neighbors.

The second metric is the *average frequency of switching between an active slot and a sleeping slot (AFS).* We chose this metric to evaluate the schemes because frequent switching between active and sleeping modes in a node increases power consumption [14]. Given the number of switches between active and sleeping, S_{as}, within a sample interval *ls*, *AFS* is computed in Equation (2).

$$AFS = \frac{S_{as}}{ls} \quad (2)$$

Table 3 shows the number of dead nodes in the 100-node network running the schemes. It can be seen that the network running FPS had more conflict-scheduled nodes than running DCS. The reason is as follows. FPS only considers the collision avoidance among siblings. However, neighbors that have different parents may take the same transmission slots. These nodes are the conflict-scheduled nodes. In comparison, DCS may allocate conflict transmission slots to neighbors only when the transmission slot information of neighbors is lost.

Table 3 also shows that SS had no conflict-scheduled nodes whereas FPS and DCS had no non-scheduled nodes. The reason is rooted in the principles of the scheduling schemes. In SS, after a node sends RSETUP packet, the node often fails to receive the ACK packet from the sink due to collisions. When this happens, the receiving and transmission slots of the nodes along the path are all wasted since the source node will not send packets in these slots. These wasted slots may in turn cause some nodes to have no slots to send or to receive and to become the non-scheduled nodes. In contrast, FPS and DCS do not result in such wasted slots, and can always allocate schedules to nodes of sufficiently long sample intervals.

Table 3. Number of dead nodes

Scheme	Conflict-schedule nodes	Non-scheduled nodes
FPS	15	0
SS	0	13
DCS	4	0

The average frequency of switching between an active slot and a sleeping slot (AFS) is shown in Table 4. It demonstrates that DCS has the lowest frequency of switching. This is because our scheme makes effort to allocate consecutive transmission slots to nodes.

Table 4. Average frequency of switching

Scheme	AFS (#switches / second)
FPS	0.203
SS	1.318
DCS	0.093

6.2. Query Processing Performance

6.2.1. Experimental Setup

We used 10 Crossbow MICA2 motes [6] to run the optimized TinyDB and the original TinyDB. The tool for measuring the power consumption of the motes was an HP- 4156 oscilloscope in an electronic lab (Figure 9).

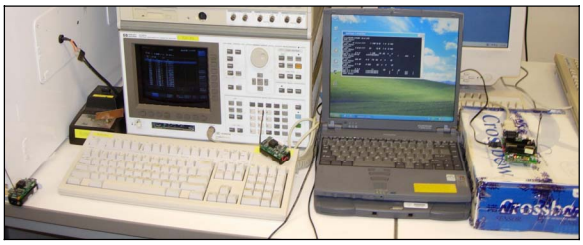


Figure 9. The power measurement setup

We also used VMNet [22] to emulate the MICA2 motes and evaluated our scheme on this realistic testbed. We chose VMNet because it can provide detailed runtime information of emulated nodes whereas a real WSN cannot due to the limitation of the real sensor nodes and the measuring equipment. VMNet emulates networked sensor nodes at the CPU instruction level. It directly executes binary code that is compiled for real sensor nodes.

The configuration of the MICA2 motes that we used and that VMNet emulated were the same. The sensor board was MTS300CA and the processor board MPR410CA. The sink consisted of an MIB510 interface board and an MPR410CA processor board. The transmission power of the radio circuit (CC1000) was 11.1mA [5] and the transmission range remained constant in the experiments. The source code of TinyOS and TinyDB running on the motes or on the emulated nodes was version 1.1.0 [19].

The network topology of the emulated network in VMNet is shown in Figure 10. An emulated network in VMNet is called a *VMN*. There were three hops in this VMN. This topology can be used to test Constraint (iii) of Section 4, the most complex one among the four constraints. In addition, node 6 was the neighbor of nodes 8 and 9, and node 7 was the neighbor of nodes 1 and 5. These neighboring nodes were useful for testing constraint (i) of Section 4. Finally, node 2 had two children, which can be used to test the other two constraints of Section 4.

We attempted to deploy the 10 real motes to have the same topology as that of the VMN in Figure 10. However, the resulting real network topology was not exactly the same as that of the VMN. In particular, the number of neighbors of a node in the real WSN may be different from the corresponding node in the VMN, because the transmission range of a real sensor node is irregular [5]. Nevertheless, the two topologies were similar enough for performance validation purposes.

We ran Query 1 with three representative sample intervals - 2 seconds, 10 seconds and 60 seconds. We chose this query to show whether the transmission slots enable partial aggregation at the parent, since it involves an aggregation function.

Query 1: *SELECT avg(light) FROM sensors*

6.2.2. Schedule for In-Network Aggregation

We examined the debugging messages in VMNet that show the schedule when running the Optimized TinyDB in the VMN. The schedules constructed by our scheme on the emulated nodes are shown in Figure 10, with the sleeping slots omitted for simplicity. The resulting schedules observed all four constraints of Section 4 and allowed the parent nodes to perform partial aggregation. In particular, each parent node needed to transmit only one packet in every sample interval due to the partial aggregation.

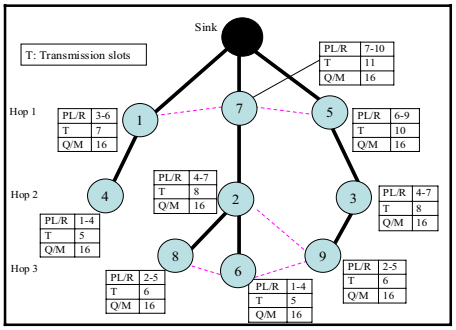
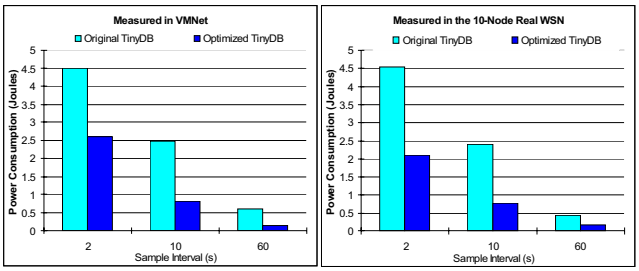


Figure 10. The VMN topology with schedules

6.2.3. Power Consumption

After the schedules were constructed and the nodes started query processing in the VMN, we measured the average one-minute power consumption of nodes running the Optimized TinyDB and the Original TinyDB. Figure 11a shows the results. It can be seen that the Optimized TinyDB reduces the average node power consumption by 42%, 67%, and 75% at the sample interval of 2s, 10s, and 60s, respectively. The improvement is mainly due to scheduling: in each sample interval, a node keeps active only for a few slots, and sleeps in the other slots. Note that the original TinyDB also made nodes sleep, but a node in the original TinyDB was put into sleep only after it had been active for at least 4096 milliseconds in a sample interval. Due to this condition, even though such a node would be sleeping for the remainder of the interval, the total sleeping time in the original TinyDB was much shorter than that in the optimized TinyDB.

Similarly, we measured the one-minute power consumption of nodes in the 10-node real WSN. Because it is slow to copy the measured power consumption information out of the oscilloscope, we picked nodes 2, 4, and 7 as the representative nodes and measured them. The average power consumption of these nodes was improved by 53%, 67%, and 64% at the sample interval of 2s, 10s, and 60s, respectively, as shown in Figure 11b. The difference between the power consumption measured in VMNet and that in the real WSN was within ±15%. Possible reasons for this difference include the difference in the topologies and the measurement errors of the oscilloscope [22].



a. **b.**

Figure 11. Power consumption improvement

7. Conclusion and Future Work

We have presented the design and implementation of our distributed, cross-layer scheduling scheme for power-efficient in-network sensor query processing. This scheme requires interaction between neighbors only. Moreover, our scheme utilizes cross-layer information: each node determines its slots from the information of both the network and the query; similarly, parents allocate slots to their children considering the requirements of the queries being processed. With this distributed and cross-layer design, our scheme is able to reduce the number of dead nodes and the switching frequency between active and sleeping modes, and to support both data collection and aggregation queries with significant power saving.

We have evaluated our scheme using simulation, emulation, and real sensor nodes. The simulation results show that our scheme outperforms the other two existing schemes on both the number of dead nodes and the switching frequency in the resulting schedule. The emulation and the real WSN results demonstrate that our scheme significantly reduces power consumption of in-network sensor query processing.

Although the schedule construction time is relatively short (at the minute level) for long-running queries, it is still inefficient to reconstruct a schedule from scratch when there is a change in the network topology or a new query arrives. As one direction of future work, we are studying incremental schedule update.

Acknowledgements

Funding for this work was provided by the Hong Kong Research Grants Council (RGC) through HKUST6263/04E.

References

[1] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri, "Querying the Physical World", IEEE Personal Communications, vol. 7, no. 5, pp. 10-15, October, 2000.

[2] Athanassios Boulis and Mani B. Srivastava, "Node-Level Energy Management for Sensor Networks in the Presence of Multiple Applications", PerCom, 2003.

[3] Phil Buonadonna, Joseph Hellerstein, Wei Hong, David Gay, and Samuel Madden, "TASK: Sensor Network in a Box", EWSN, 2005.

[4] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris, "Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks", MOBICOM, 2001.

[5] Chipcon, http://www.chipcon.com.

[6] Crossbow Inc, http://www.xbow.com.

[7] Cédric Florens and Robert McEliece, "Packet Distribution Algorithms for Sensor Networks", INFOCOM, 2003.

[8] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan, "Adaptive Protocols for Information Dissemination in Wireless Sensor Networks", MOBICOM, 1999.

[9] Barbara Hohlt, Lance Doherty, and Eric Brewer, "Flexible Power Scheduling for Sensor Networks", IPSN, 2004.

[10] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks", MOBICOM, 2000.

[11] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS", NSDI, 2004.

[12] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", OSDI, 2002.

[13] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, "The Design of an Acquisitional Query Processor for Sensor Networks", SIGMOD, 2003.

[14] Kresimir Mihic, Tajana Simunic, and Giovanni De Micheli, "Reliability and Power Management of Integrated Systems", Euromicro Symposium on Digital Systems Design, 2004.

[15] Su Ping, "Delay Measurement Time Synchronization for Wireless Sensor Networks", IRB-TR-03-013 , Intel Research Berkeley Lab, 2003.

[16] Mihail L. Sichitiu, "Cross-Layer Scheduling for Power Efficiency in Wireless Sensor Networks", INFOCOM, 2004.

[17] Mark Stemm and Randy H Katz, "Measuring and Reducing Energy Consumption of Network Interfaces in Hand-Held Devices", IEICE Transactions on Communications, vol. E80-B, no. 8, pp. 1125–1131, August, 1997.

[18] Andrew S. Tanenbaum, *Computer networks*, Prentice Hall, Hardcover, 4th edition, August 2002.

[19] TinyOS, http://www.tinyos.net.

[20] Alec Woo and David Culler, "A Transmission Control Scheme for Media Access in Sensor Networks", MOBICOM, 2001.

[21] Alec Woo, Ternence Tony, and David Culler, "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks", SenSys, 2003.

[22] Hejun Wu, Qiong Luo, Pei Zheng, and Lionel M. Ni, "VMNet: Realistic Emulation of Wireless Sensor Networks", Technical Report HKUST-CS05-05, HKUST, 2005.

[23] Yong Yao and Johannes Gehrke, "Query Processing for Sensor Networks", CIDR, 2003.

[24] Wei Ye, John Heidemann, and Deborah Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks", INFOCOM, 2002.

[25] Mohamed Younis and Tamer Nadeem, "Chapter 9: Energy Efficient MAC Protocols of Ad Hoc Networks", Wireless Ad-Hoc and Sensor Networks, Ed. Ahmed Safwat, Kluwer Academic Publishers (to appear).