TinyOS-based Quality of Service Management in Wireless Sensor Networks*

Nina Peterson[1], Lohith Anusuya-Rangappa[1], Behrooz A. Shirazi[1], Renjie Huang[2], Wen-Zhan Song[2], Michael Miceli[3], Devin McBride[4], Ali Hurson[5], Rick LaHusen[6]

Abstract

Previously the cost and extremely limited capabilities of sensors prohibited Quality of Service (QoS) implementations in wireless sensor networks. With advances in technology, sensors are becoming significantly less expensive and the increases in computational and storage capabilities are opening the door for new, sophisticated algorithms to be implemented. Newer sensor network applications require higher data rates with more stringent priority requirements. We introduce a dynamic scheduling algorithm to improve bandwidth for high priority data in sensor networks, called Tiny-DWFQ. Our Tiny-Dynamic Weighted Fair Queuing scheduling algorithm allows for dynamic QoS for prioritized communications by continually adjusting the treatment of communication packages according to their priorities and the current level of network congestion. For performance evaluation, we tested Tiny-DWFQ, Tiny-WFQ (traditional WFQ algorithm implemented in TinyOS), and FIFO queues on an Imote2-based wireless sensor network and report their throughput and packet loss. Our results show that Tiny-DWFQ performs better in all test cases.

*This work is partially supported by NASA AIST Grant #106269, NSF-ITR grant IIS-0326505, and NSF-ITR grant IIS-0324835.

[1] School of Electrical Engineering and Computer Science Washington State University
{npicone/lanusuya/shirazi}@eecs.wsu.edu

[2] School of Engineering and Computer Science Washington State University-Vancouver
{renjie_huang/songwz}@wsu.edu

[3] School of Computer Science; Louisiana State University
mmicel2@lsu.edu

[4] School of Computer Science; Seattle University
mcbrided@seattleu.edu

[5] School of Computer Science; Missouri University of Science and Technology; hurson@mst.edu

[6] USGS Cascades Volcanic Observatory
rlahusen@usgs.gov

1. Introduction

The major contribution of this work is introduction of a novel dynamic scheduling algorithm to handle prioritized data communication in wireless sensor networks. Additionally, wireless sensor network environments impose significant communication and computation limitations that have forced us to come up with new algorithms to implement the proposed dynamic scheduling algorithm. Tiny-DWFQ dynamically adjusts its parameters in order to fulfill the Quality of Service (QoS) requirements of the data being sent throughout the network in the midst of continuously changing network conditions and requirements. One of the major obstacles that we overcame in achieving this was designing and implementing a lightweight algorithm, specifically for resource constrained wireless sensor networks, that is robust enough to handle continuous real-time data flows. Due to the limited computational capacity of wireless sensor networks, most significantly memory, computational capacity, and bandwidth, our algorithm is designed specifically to meet these demanding requirements. Bandwidth limitations in any wireless network provide a restricted amount of resources to be shared amongst competing data flows. Thus, in order to maximize the available resources, we designed Tiny-DWFQ to be a congestion reactive, scheduling algorithm.

In order to meet the overall needs of many different types of communication, it is necessary for data to be categorized based on its importance, or priority level within the network. For example, in our network scenario (OASIS [8]), we are deploying wireless sensors on the crater of Mount St. Helens for volcanic monitoring and prediction. The scientists who monitor the data do not view each type as equally important; rather, specific data (ex. Seismic RSAM data) is significantly more important than other data (ex. Infrasonic inter-event data). Ideally we would be able to transmit all of the data all of the time. However when network congestion occurs due to the limited available bandwidth, a decision must be made as to which data should be sent and in which

978-0-7695-3450-3/09 $25.00 © 2009 IEEE

order. These decisions are the responsibilities of our Tiny-DWFQ scheduling algorithm.

Within the design of Tiny-DWFQ there are several critical items which factor into the algorithms execution: data priority, node priority, current congestion, and available resources. In our model, we consider both the importance of the type of data as well as the importance of the node generating the data. This allows for a fine-grained granularity in our prioritization scheme giving the user a more powerful tool to utilize. For example, if a hot spot occurs on the volcano, the data from the node(s) within the hot area will be of more importance at that particular point in time. Additionally, the data from the nodes of a specific area of interest still needs to be prioritized, possibly in a different manner than under routine operations. Thus, in order to capture this, we need to distinguish between data priority and node priority. In Section 4 we will describe the method that we use to assign a Dynamic Weighted Priority (DWP), which incorporates both data and node priorities.

The remainder of the paper is organized as follows: Section 2 describes the related works. Section 3 defines the architecture of our Tiny-DWFQ algorithm. The details of the workings of Tiny-DWFQ are discussed in Section 4. In Section 5 we present the results of our performance study on our testbed of Imote2 sensors. Finally, the conclusions are presented in Section 6.

2. Related works

The design of scheduling algorithms for networks has been studied and implemented in depth for many years. Additionally, many of these works have focused specifically on prioritized scheduling algorithms, which instead of treating every flow or data packet as equal, create a scheme to rank or prioritize them. Depending on the situation for which the network is designed, the prioritization scheme's basis will vary. For example, some scheduling algorithms prioritize data based on the source, while others prioritize based on the type of data being transmitted [7].

Two well-known scheduling algorithms are Fair Queuing [2] and Weighted Fair Queuing (WFQ) [10]. Both Fair Queuing and WFQ are being used today. WFQ is a static scheduling algorithm, in which every flow/queue is associated with a fixed weight and packets are scheduled based on the Type of Service requirements of the network. For example, Emergency Services are given a higher priority within the system compared to General Service requests. This static scheduling algorithm is unable to adapt to changes in congestion, making it incapable of handling run time changes in the priorities of the flows. Additionally, due to the complex computational load of this algorithm, it has not been designed for or implemented on wireless sensor networks. Until recently it was not practical or feasible for scheduling algorithms to be implemented on sensor networks, due to their extremely limited resources.

New advances in wireless sensor technology have opened the door for advancement and implementation of lightweight scheduling algorithms in wireless sensor networks. Rather than focusing on prioritization, many of these scheduling algorithms focus on reducing the network's power consumption, [15] often by regulation of sleep and wake periods of each node [1] or by turning off redundant sensors [13]. P-WFQ [14] determines the next packet to be sent through the use of a random number. [5] prioritizes data based on the error rates of the data flows. While some scheduling algorithms have been designed and tested on simulators [3] [13] [12], to the best of our knowledge, Tiny-DWFQ is the first implementation of a prioritized scheduling algorithm on a real wireless sensor network.

3. Tiny-DWFQ architecture

Since our goal is to create a dynamic QoS [4] [6] [16] scheduling algorithm [9] that is lightweight enough to be implemented on resource constrained sensor nodes, we designed our components accordingly. In WFQ [10] each priority level is mapped to a distinct physical queue in order to fulfill its QoS requirements. Thus, each priority level is associated with one static queue. Additionally, full implementation requires virtual time and complex lookup tables. The limited resources of wireless sensors prohibit this from being implemented in this environment; therefore we have come up with a new lightweight algorithm.

We overcame the resource limitation problem in sensor networks through the use of virtual queues and virtual sub-queues. All virtual queues or sub-queues are implemented on a single, physical memory space using doubly-linked lists and proper pointers for identifying the head, tail, and current virtual queue elements. We implemented our algorithm on Imote2 sensors using TinyOS.

Each message queue is composed of a "free," "processing," and "pending" virtual queue. The "free" virtual queue stores the available space for new packets. Once a packet arrives at the middleware layer, the packet is allocated to the "processing"

virtual queue. The "processing" virtual queue is used to prioritize the packets. To do this we chose to subdivide the "processing" queue into virtual sub-queues. Therefore, the "processing" queue is composed of n (where n is the number of priority levels) sub-queues, each of which corresponds to one priority level. Our association of one priority level with one sub-queue is similar to WFQ's association of one priority level with one queue. However, unlike WFQ our algorithm is dynamic in its sub-queue assignments in order to give preference to higher priority packets in the midst of congestion. The details of this are discussed in Section 4.1.

After the packet has been processed it is inserted into the "pending" virtual queue. The "pending" virtual queue is used to store the prioritized packets in the proper order before they are sent on to the next layer. This new lightweight queue structure is only required and maintained by the middleware component. Thus, knowledge of the lightweight queue is only required by the middleware and not by the applications.

One of our main goals in the implementation of the middleware component is to minimize the memory requirement necessary at each node, while implementing a sophisticated prioritization scheme which dynamically assigns a priority to each packet. One way we are addressing this is through the local storage of only the node's own information. For example, node 1 will only store its own priority information, having no knowledge of any other nodes priority information. In addition to containing information about the priority of each type of data (at a specific period of time for a specific node) the node must also have knowledge of the importance of both itself (node priority) and its data's priorities, respectively.

We have n node priorities (typically $8 \geq n \geq 2$) and m data priorities (typically $8 \geq m \geq 2$), thus we will have up to $m \times n$ priorities for each message. In addition, we need to specify the relative importance of the node and data priorities (through a weighting scheme). For example, for a node further away from the crater of Mount St. Helen's,, seismic data may be less important than a node that is closer to the crater. Therefore, $w_1 m \times w_2 n$ represents the possible weighted priorities for each message (where w_1 and w_2 represent the relative weights of the data and the nodes, respectively). The $m \times n$ priorities will have to be represented by i bits, where $m \times n = 2^i$ bits, per message. For example for $m = n = 8$, we will need 6 bits for representation of priorities. However, for our Mount St. Helen's implementation in TinyOS, we can only afford to allocate 3-bits to priorities in a message. Therefore, $m \times n$ priorities need to be represented by 3 bits.

We solved this problem by defining a dynamic weighted priority, DWP. Within DWP, the relative importance of the node and data priorities is implemented as integer percentages between 0 and 100, shown below:

Node priority 1 – n (n levels)
Data Priority 1 – m (m levels)
Relative Importance of Node 0 – 100 %
Relative Importance of Data 0 – 100 %

In order to compute an overall priority associated with each packet, DWP, we need to first compute the maximum possible weighted priority, MWP. This is the highest priority that is allowable within the system, shown below in Equation 1. The computation of the weighted priority for any node consists of two parts, the node's priority (n) and the data's priority (m). Additionally, each of these priorities is multiplied by its relative importance within the network. Thus, in the equation for the maximum weighted priority both the node's priority and the data's priority are multiplied by 100 to obtain the maximum importance within the system. In order for a packet to have the maximum weighted priority, it would need to have been generated by the highest priority node within the network, and it would also need to be the highest priority data type. It should be noted that this is not a static, one-time computation for all networks; rather it is dynamic based on the values of n and m chosen for both the node and data priorities. For our implementation we chose both n and m to be 8, as previously discussed, based on our 3 bit limitation. Thus, the maximum weighted priority is 1600.

$$MWP = n * 100 + m * 100 \quad (1)$$

With the limited memory of the sensor nodes, it is not feasible to have each of the possible weighted priorities (1600) be their own level associated with a virtual sub-queue. Thus, we designed our lightweight algorithm so that each weighted priority is associated with a range. In order to be able to place each of the different weighted priorities within a range, we first need to determine the number of priorities that each range will contain, C. Note, that C will vary depending on the choices of n and m. The calculation of C, shown below in Equation 2, is the maximum weighted priority divided by the number of desired priority levels, x. Based on our 3 bit implementation, we chose x to be 8, thus C = 1600 / 8 = 200.

Therefore each of the ranges will contain 200 individual priorities.

$$C = MWP / x \quad (2)$$

Each range is determined by combining the dynamic weighted priority, DWP, and the number of priorities per priority level, C. The range, R, can be computed by Equation 3, shown below. Note that this is the calculation for the first and last terms in the range and not the difference of two terms. For our current implementation, we have shown these calculations in Table 1.

$$R = DWP(C+1) - (DWP+1)C \quad (3)$$

Table 1: Priority levels and ranges

DWP	Range	Type	Bits
0	0 – 200	Very-low priority	000
1	201 – 400	Low priority	001
2	401 – 600	Medium-low priority	010
3	601 – 800	Medium priority	011
4	801 – 1000	Medium-high priority	100
5	1001 – 1200	High priority	101
6	1201 – 1400	High-urgent priority	110
7	1401 – 1600	Urgent priority	111

In addition to implementing a prioritization scheme which takes into account a dynamically weighted sum of both the data and node priorities, Tiny-DWQ must additionally take into account the way in which the scheduling algorithm will determine both its enqueuing and dequeuing algorithms. This ensures that the maximum QoS for high priority data is achieved.

4. Tiny-DWFQ workings

Tiny-DWFQ is subdivided into two main portions, the enqueuing phase and the dequeuing phase. It should be noted that all of the ordering of packets is done in a combination of the enqueuing and dequeuing phases, requiring no physical re-ordering to ever take place. Within the enqueuing phase, the packets are inserted according to their DWP into the appropriate virtual sub-queue based on: a combination of the data priority, the node priority, the relative importance of the node, and the relative importance of the data. It should be noted that the enqueuing phase and the dequeuing phase occur, both repeatedly as well as intermittently. The dequeuing phase consists of an automated process for choosing the proper number and the proper order for sending out packets. In order to accomplish this, Tiny-DWFQ takes into account the current network congestion, the available virtual queue resources, the number of virtual sub-queues, and the weight of each virtual sub-queue.

4.1. Enqueuing

When a packet is generated at a node, say node x, the packet's priority is determined. This determination is based on the current network congestion (congestion is defined in Section 4.2), the node's current priority level, the data priority, and the relative importance of both the node and the data, in accordance with the methods described in Section 3. Once the priority of the packet is determined it will remain constant. However, one of the dynamic facets of the algorithm is that two packets of the same data type generated at the same node at two different times may have different priorities since they will reflect the current state of the network and the environmental conditions (e.g. lava flowing or not) at those particular instances.

When a packet is received by the middleware layer, the packet is assigned to a virtual sub-queue based on its DWP. The number of virtual sub-queues is based on the system requirements as well as the sensors being used and their resource constraints. Obviously when more space is used to buffer packets, it will result in a decrease in packet drop and an increase in throughput. However, due to the limited resources, the decision regarding the size of each of the queues, and number of virtual-sub queues must be carefully considered. In our implementation we choose to have a queue of size 40 (allowing for 40 packets) with eight virtual sub-queues. Note each virtual sub-queue will be of size 5. Our decision to have eight virtual sub-queues was determined based on the limited 3-bits of memory used for the DWP, thus each of the priority levels will be associated with one virtual-sub queue.

With Tiny-WFQ each priority level is associated with one static queue. If a new packet arrives at a node and the queue with which it is associated is full, the packet is dropped. However, with Tiny-DWFQ, when a packet arrives and the queue with which it is

associated with is full, it is placed in the queue with the next highest priority which has available space.

If an incoming packet arrives and there is no space available in any of the virtual sub-queues, then the incoming packet replaces the oldest lowest priority packet in the queue, unless it is of a higher priority than the incoming packet. For example, if a packet arrives with priority 5 and all of the virtual sub-queues are full, then it is compared with the oldest lowest priority packet. Note that the oldest lowest priority packet will be the last element in the lowest priority virtual queue, thus, no searching is necessary. Since the incoming packet is priority 5, it will replace the last element unless the old packet has a higher priority (priority 6 or greater). We ensure that if the last element is also priority 5, that the newer incoming packet replaces the older packet to preserve the freshness of the data. Thus, an incoming packet will only be dropped if none of the virtual sub-queues have available space and the incoming packet is of a lower priority than all other packets in the queue.

4.2. Dequeuing

The dequeuing portion of the algorithm takes into account four distinct parameters in its computation: current network congestion (packets per virtual sub-queue pv_1, pv_2, …, pv_n), the free or available space in the pending virtual queue (packets in pending virtual queue pp), the number of virtual sub-queues (n), and the weight of each virtual sub-queue (w_1, w_2, w_3, …, w_n).

The virtual queue weights are defined as a function of congestion. Congestion is defined as the total used space in the lightweight queue divided by the size of the lightweight queue. We implemented the congestion levels as ranges. The number of ranges is application specific and can be altered to fit the user's needs. For our application we implemented four congestion levels: 0-40%, 41-75%, 76-90%, 91-100%. Thus, each of the four congestion levels is associated with a set of weights, one for each of the virtual sub-queues.

Assignment of weights ($w_1 - w_8$), for each virtual queue, is determined as a function of both the free space in the virtual pending queue (pp) and the congestion levels, as shown in Table 2. Note, that the sum of the weights for each congestion level must equal pp. By assigning the weights of each virtual sub-queue based on the current free space in the virtual pending queue, we continually and dynamically re-adjusting the weights, as the network changes in order to reflect its current conditions.

Table I: Virtual queue weights based on congestion level

Index	Level 1: 0-40%	Level 2: 41-75%	Level 3: 76-90%	Level 4: 91-100%
VQ1	0.20*pp	0.30*pp	0.40*pp	0.50*pp
VQ2	0.20*pp	0.20*pp	0.20*pp	0.15*pp
VQ3	0.15*pp	0.15*pp	0.15*pp	0.10*pp
VQ4	0.15*pp	0.10*pp	0.10*pp	0.10*pp
VQ5	0.10*pp	0.10*pp	0.05*pp	0.05*pp
VQ6	0.10*pp	0.05*pp	0.05*pp	0.05*pp
VQ7	0.05*pp	0.05*pp	0.025*pp	0.025*pp
VQ8	0.05*pp	0.05*pp	0.025*pp	0.025*pp

4.2.1. Dequeuing algorithm details. There are four steps that take place in the dequeuing algorithm. First, the current congestion level at the node must be determined. Second, the weight of each of the virtual sub-queues must be calculated based on the current congestion, the available space in the virtual pending queue, and the application constants associated with each congestion level. Once the weight of each virtual sub-queue is calculated, the third step is to determine the number of packets to be dequeued from each virtual sub-queue. The number of packets to be dequeued from virtual sub-queue 1 (DP_1) is determined by dividing the free space in the virtual pending queue by the number of virtual sub-queues, then multiplying the result by the weight of virtual sub-queue 1. For example, if the network is 50% congested (note: w_1 = .30*pp), there is room for 24 packets in the virtual pending queue, and there are 8 virtual sub-queues, then the most packets that can be dequeued from virtual sub-queue 1 is (.30*24) * (24/8) = 21. However, we cannot dequeue more packets from the virtual sub-queue than it has (S_1). Say that there are only 4 packets in virtual sub-queue 1, thus, we can only take 4 packets from virtual sub-queue 1. Thus S_1 = 21- 4 = 17. This is shown below in Equations 4 and 5.

$$DP_1 = \min\left(pv_1, w_1 * \frac{pp}{n} \right) \quad (4)$$

$$S_1 = \max\left(pv_1, w_1 * \frac{pp}{n} \right) - DP_1 \quad (5)$$

Now, the number of packets that we can dequeue from virtual sub-queue 2 (DP_2) is (note w_2 = .20*pp) (.20*24) * (24/8) = 14, but we can also add the 17

packets that we did not dequeue from virtual sub-queue 1. Thus, the most we can dequeue from virtual sub-queue 2 is 14 + 17 = 31. However, if we only have 3 packets in virtual sub-queue 2 then that is the most that we can dequeue (S_2). This is depicted below in Equations 6 and 7.

$$DP_2 = \min\left(pv_2, w_2 * \frac{pp}{n} + S_1 \right) \quad (6)$$

$$S_2 = \max\left(pv_2, w_2 * \frac{pp}{n} + S_1 \right) - DP_2 \quad (7)$$

This process continues until the number of packets to be removed form the last virtual priority sub-queue is determined. The general forms of the equations are shown below in Equations 8 and 9.

$$S_{n-1} = \max\left(\begin{array}{l} pv_{n-1}, w_{n-1} * \frac{pp}{n} + \\ \left(\max\left(pv_{n-2}, w_{n-2} * \frac{pp}{n} \right) - DP_{n-2} \right) \end{array} \right) \quad (8)$$

$$DP_n = \max\left(pv_n, w_n * \frac{pp}{n} + S_{n-1} \right) \quad (9)$$

The final step is to remove the appropriate number of packets form each of the virtual sub-queues. This is done according to the number of packets to be removed from each virtual priority sub-queue described above.

5. Results

In order to evaluate our Tiny-DWFQ algorithm we designed and ran a series of tests. Our goal was to design a Quality of Service [11] management algorithm for wireless sensor networks. Thus, in order to accurately test whether or not we had accomplished this we felt that it was necessary to forgo all simulations and physically implement this algorithm on a real wireless sensor network. Naturally, implementation on real sensors brings to the forefront all of the limitations and obstacles associated with these resource constrained devices. For example, measuring the packet loss, for each priority level, with simulations is a relatively easy task. It simply requires the counting and recording of the number of packets sent and received for each priority level which can be displayed through print or output statements. However, due to the limited resources of the individual nodes, this addition of a print statement (which is actually an additional packet) for each packet, can cause serious congestion and in some cases failure of nodes. This caused us to use alternative measures to compute the packet loss for each priority level while not overloading the nodes. Our approach is discussed in detail in Section 5.2.

Evaluation of our algorithm was done by measuring two network statistics, packet loss and throughput. Each of these was measured for each of the different priority levels. In our evaluations we compared our Tiny D-WFQ algorithm with Tiny WFQ. Additionally, we also implemented a simple FIFO queue for additional comparison which, when full, drops newly arriving packets. For each algorithm we implemented a queue size of 40, in order to achieve a fair comparison.

5.1. Network scenario

Our experiments were all conducted using a testbed consisting of 10 Imote2 wireless sensor nodes. The testbed was setup inside our laboratory. Due to the limited space and the goal of capturing a realist scenario we turned down the radio range of each node so that we could induce the multihop nature which would be present in the field. The nodes were arranged in a random or scattered fashion. The tesbed includes Panorama (see Figure 3) which we designed and used as a monitoring tool for simulation of the senor nodes on Mount St. Helens. Each node's generated and multihop data was sent to a sink node (through a dynamic multihop path) and then forwarded to its final destination which was a laptop computer. This laptop computer is representative of the computer which would be housed at the command and control center in a real volcanic monitoring scenario.

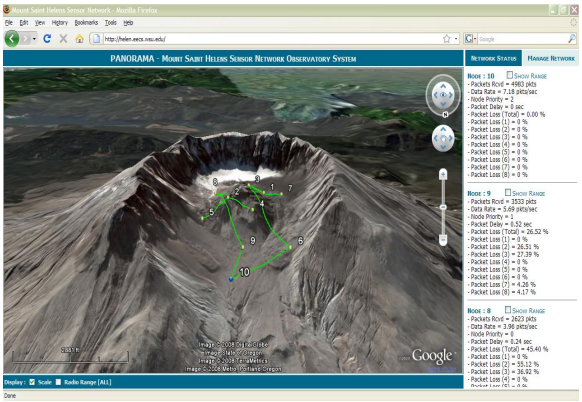

Figure 3: Panorama monitoring tool

We implemented the tests by generating seven different types of data streams, emulating data collected at a volcanic site, at each node: seismic Real-Time Seismic-Amplitude Measurement (RSAM), seismic triggered event data, seismic inter-event data (continuous), infrasonic RSAM data, infrasonic triggered event data, infrasonic inter-event data (continuous), and lightning data. The reason for the simulation of these data types is due to the cost of acquiring 10 of each of these sensors. Additionally, we were able to adjust the data rates to that which would be sampled from the actual sensors thus causing no difference in the evaluation of our algorithms. The sampling rates and the priorities assigned to each of the different data types are shown below in Table 3.

Table 3: Data rates and priorities

Data Type	Sampling Rate (bytes/sec)	Priority
Seismic RSAM	4	6
Seismic Event	0-200	4
Seismic Inter-Event	100	2
Infrasonic RSAM	4	5
Infrasonic Event	0-200	3
Infrasonic Inter-Event	100	1
Lightning	2	5

It should be noted that these sampling rates and priority assignments were not arbitrary, rather there were used as recommended by Earth Scientists who specialize in volcanic monitoring. Additionally, for the event data for both seismic and infrasonic the sampling rates are 0-200 bytes per second. This is due to the fact that event data refers to data that is representative of a triggered or actual physical event or activity. Thus, at some times there is not event data (0 bytes/sec), however, when event data is being generated it can be sampled at a very high rate (200 bytes/sec).

5.2. Packet loss

We had two goals to accomplish in evaluating the packet loss. First, we wanted to make sure that our experiment was performed in as realistic an environment as possible. In order to accomplish this we ran all of our tests on a real testbed of Imote2 sensors. Additionally, we ran the tests with data types, rates, and priorities defined by Earth Scientists as discussed in Section 5.1. These are real scenario parameters that are being used in volcanic monitoring scenarios. Second, we wanted to evaluate the packet loss for each individual priority level for each of the three algorithms. In order to compute packet loss (PL_i), for each individual priority level i, we measured the total number of packets generated at each node for each priority level, x_k where k is the node's id. Next, we measured the total number of packets received by the destination (laptop) for each priority level, y. This allowed us to compute the packet loss for each priority level by computing one minus the sum of each of the number of packets generated at each node divided by the number of packets received by the destination (laptop) for each priority level. The total packet loss for each priority level is displayed below in Equation 10.

$$PL_i = 1 - (\sum_{k=1}^{11} x_k / y) \quad (10)$$

In order to accurately capture a realistic scenario we felt that it was necessary take the average of multiple runs. More specifically, since we used realistic data generated on real motes we were not necessarily getting the exact same data set for each run, which introduces some variances in the results. To reduce the affects of this we ran 10 experiments each for 20 minutes and took the average of these 10 runs. The results are shown in Figures 1-6.

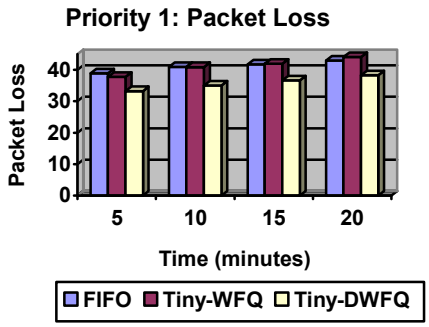

Figure 1

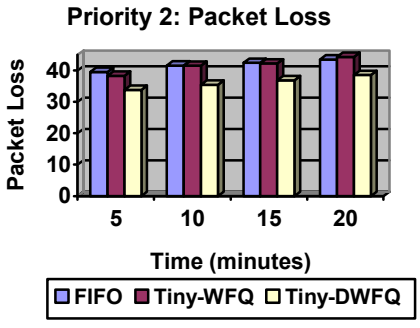

Figure 2

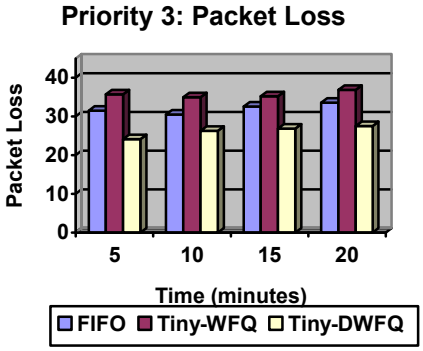

Figure 3

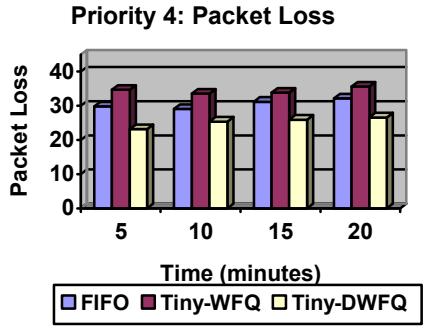

Figure 4

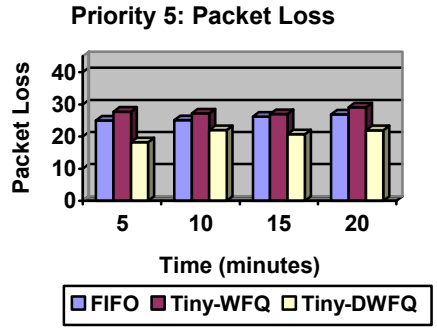

Figure 5

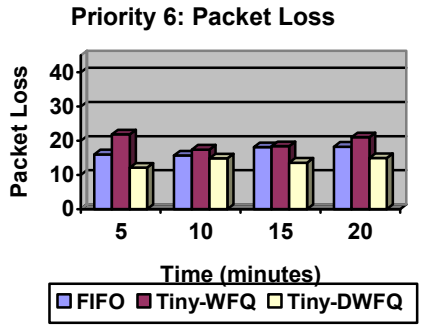

Figure 6

It can be seen in Figures 1-6 that Tiny-DWFQ performs better than both Tiny-WFQ and FIFO algorithms in terms of reducing the packet loss. It is interesting to note that a simple FIFO algorithm outperforms the Tiny-WFQ in most cases, especially for low priority messages (Figures 5 and 6) or messages with higher sampling rates (Figures 3 and 4). Additionally, it can be seen that Tiny-DWFQ's best relative improvements are for the mid range priorities (priorities 3 and 4 in Figures 3 and 4). This is due to the high rate at which these priority packets were generated. It can be seen from Table 3 that priorities 3 and 4 have the highest sampling rates. Therefore, as the number of packets increases so does the relative improvement of Tiny-DWFQ over the other algorithms.

However, Tiny-DWFQ's primary goal is to ensure a high level Quality of Service for high priority data. Thus, we are particularly interested in the performance of Tiny-DWFQ for the high priority data (note priority 6 is the highest). Figure 6 shows that for the highest priority data Tiny-DWFQ only lost between 12.09% and 14.93% of the data while Tiny-WFQ lost between 17.09% and 21.82% of the data and the FIFO queue lost between 15.68% and 18.25% of the data.

5.3. Throughput

Computation of the total throughput of our system using each of the three algorithms was measured with the same goals in mind as we had with the packet loss. We wanted to ensure a realistic environment and compute the total throughput for each individual priority level. The total throughput (TP_i) for each priority level i is defined as the number of packets, p_k, which traverse from the source to the destination per unit of time, u. This is shown below in Equation 11.

$$TP_i = \sum_{k=1}^{11} p_k / u \qquad (11)$$

However, since we are using realistic data generation, as discussed in the previous section, the precise number of packets generated for each of the different runs varied slightly. Thus, in order to accurately evaluate the throughput it was necessary for us to take the throughput, TP_i , and divide it by the total number of packets generated g_i for priority level i during that experimental run. The throughput percentage, TPP_i , is shown below in Equation 12.

$$TPP_i = TP_i / g_i * 100 \qquad (12)$$

As described for the packet loss we also ran 10 experiments each for 20 minutes and took the average of the 10 runs for throughput. The results are shown below in Figures 7-12.

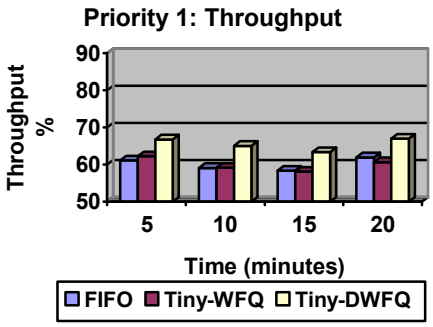

Figure 7

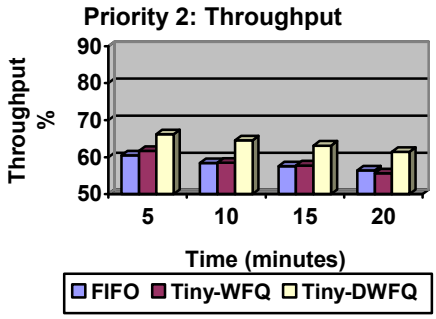

Figure 8

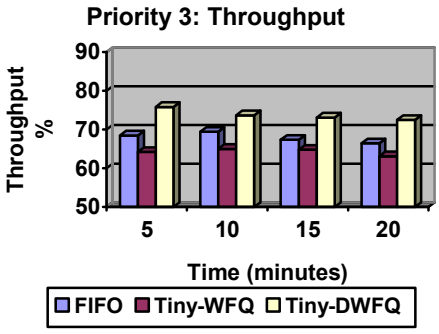

Figure 9

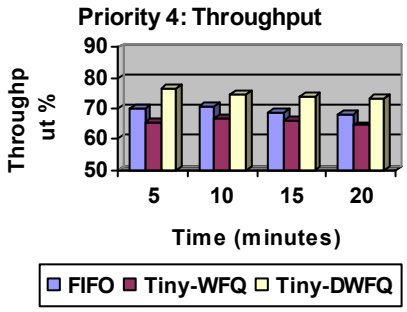

Figure 10

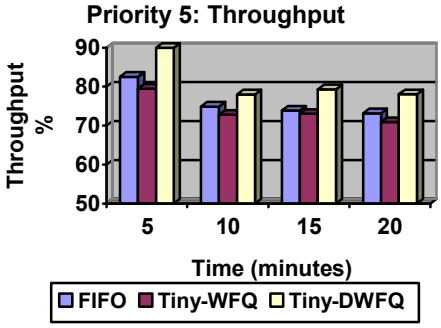

Figure 11

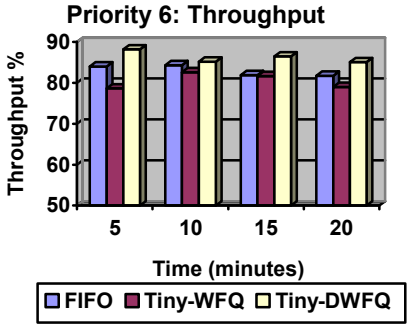

Figure 12

In all cases, Figures 7-12 show that Tiny-DWFQ outperformed both of the other algorithms in terms of improving the network throughput. Similar to the previous results, the FIFO algorithm performs better than Tiny-WFQ for high frequency data (Figures 9 and 10) or low priority data (Figures 11 and 12). Additionally, it can be seen that Tiny-DWFQ's best relative improvements are for the mid range priorities (priorities 3 and 4 in Figures 9 and 10) because of the very high sampling rates of these data.

While we are primarily concerned with the highest priority data, we also do not want to starve the lower level data. If we did not care about any of the lower priority data we would just not transmit it at all. Therefore, it is good that we are able to maintain better Quality of Service than the other algorithms even for low level traffic.

For the highest priority data, Figure 12 shows that Tiny-DWFQ achieved a throughput percentage between 85.07% and 88.18% while Tiny-WFQ only achieved a throughput percentage between 78.63% and 82.59% and the FIFO queue's throughput percentage was between 81.75% and 84.32%. Hence, Tiny-DWFQ was able to show improvement over the other algorithms for all priorities of data but most importantly for high priority data.

We believe that the reason for this improvement is twofold. First, our enqueuing algorithm allowed us to fully utilize the entire queue structure for high priority packets in the midst of congestion. Secondly, our dequeuing algorithm utilized not only information about the DWP of the packet but also the current network congestion as well as the available network resources. This allowed us to essentially store the higher priority packets in the queue until there was room for them in the network.

6. Conclusions and future work

In conclusion, we have designed and implemented a lightweight dynamic weighted fair queuing algorithm, Tiny-DWFQ on a real wireless sensor network testbed. Our goal was to introduce a novel dynamic scheduling algorithm to increase the bandwidth for high priority data. We accomplished this by ensuring a high level of Quality of Service for high priority data without starving the lower priority data. In order to compare our algorithm against others we implemented a lightweight version of WFQ, Tiny-WFQ, as well as a simple FIFO queue. The metrics that we used for evaluation were packet loss and throughput. After conducting our tests we were able to show improvement in both an increased throughput percentage as well as decreased packet loss for all types of data.

We are currently in the process of implementing a large-scale Optimized Autonomous Space In-Situ Sensorweb (OSAIS), for monitoring Mount St. Helens, with Tiny-DWFQ integrated into the middleware component. This sensorweb will be deployed continuously for one year (2009-2010).

10. References

[1] A. Berfield and D. Mosse, “Efficient Scheduling for Sensor Networks”, Mobile and Ubiquitous Systems: Third Annual International Conference on Networking & Services, July 2006, pp. 1-8.

[2] A. Demers, S. Deshav, and S. Shenker, “Analysis and Simulation of a Fair Queuing Algorithm”, In Proceedings of SIGCOMM ‘89, September 1989, pp. 1-12.

[3] Y. Drougas and V. Kalogeraki, “RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications”, Parallel and Distributed Processing Symposium IPDPS 2007, March 2007, pp. 1-10.

[4] J. Frolik, "QoS Control for random access wireless sensor networks", WCNC 2004 IEEE Wireless Communications and Networking Conference, 2004, pp. 1510-1515.

[5] J-Y. Huang, Y-K. Tseng, M.S. Lin, and W-S. Hsieh, "Error Rate-Based Dynamic Weighted Fair Queuing In Wireless Networks," The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05), 2005, pp. 546-553.

[6] R. Iyer, L. Kleinrock, “QoS control for sensor networks”, IEEE International Conference on Communications ICC ’03, 2003, pp. 517 – 521.

[7] N-S. Ko, and H-S Park, “Emulated Weighted Fair Queuing Algorithm for High-speed packet-switched networks”, In Proceedings of 15th International Conference on Information Networking, 2001, pp 52-58.

[8] Optimized Autonomous Space In-Situ Sensorweb, http://sensorweb.vancouver.wsu.edu/wiki/index.php/Main_Page.

[9] N. Ouferhat, and A. Mellouk, "A QoS Scheduler Packets for Wireless Sensor Networks," IEEE/ACS International Conference on Computer Systems and Applications, 2007, pp. 211-216.

[10] A. Parekh, Control in Integrated Services Networks, PhD Thesis, MIT, February 1992.

[11] S. Tai, R.R. Benkoczi, H. Hassanein, and S. G. Akl, “QoS and data relaying for wireless sensor networks”, Journal Parallel Distribution Computing, 2007, pp. 715-726.

[12] S. Wang, Y. Wang, and K. Lin, “A Priority-Based Weighted Fair Queuing Scheduler for Real-Time Network”, In Proceedings of the Sixth international Conference on Real-Time Computing Systems and Applications, IEEE Computer Society, Washington, DC, 1999, pp. 312.

[13] K. Wu, Y. Gao, F. Li, and Y. Xiao, “Lightweight Deployment-Aware Scheduling for Wireless Sensor Networks”, Mobile Networks and Applications, December 2005, Volume 10, Issue 6, pp. 837-852.

[14] D-B. Yin and J-Y. Xie, “Probability Based Weighted Fair Queueing Algorithm with Adaptive Buffer Management for High-Speed Network”, Lecture notes in computer science. International Conference on Natural Computation No2, CHINE, 2006, pp. 992-998.

[15] X. Zhang, Y. Chen, and X. Wang, “PAS: A Power-Aware Static Scheduling Algorithm for Sensor Network”, Communications and Networking in China ChinaCom ‘06, October 2006, pp. 1-4.

[16] Q. Zhao, and L. Tong, “QoS Specific Medium Access Control for Wireless Sensor Networks with Fading”, In Proceedings of 8th International Workshop on Signal Processing for Space Communications (SPSC ’03), 2003, pp.1-8.