View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University
Purdue e-Pubs

ECE Technical Reports | Electrical and Computer Engineering

12-19-2007

SeNDORComm: An Energy-Efficient Priority-Driven Communication Layer for Reliable Wireless Sensor Networks

Vinaitheerthan Sundaram
Purdue University, jsundar@purdue.edu

Jae-Woo Lee
Purdue University, jaewoolee@purdue.edu

Saurabh Bagchi
Purdue University, sbagchi@purdue.edu

Yung-Hsiang Lu
Purdue University, yunglu@purdue.edu

Zhiyuan Li
Purdue University, li@purdue.edu

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Sundaram, Vinaitheerthan; Lee, Jae-Woo; Bagchi, Saurabh; Lu, Yung-Hsiang; and Li, Zhiyuan, "SeNDORComm: An Energy-Efficient Priority-Driven Communication Layer for Reliable Wireless Sensor Networks" (2007). *ECE Technical Reports.* Paper 365.
http://docs.lib.purdue.edu/ecetr/365

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

SeNDORComm: An Energy-Efficient Priority-Driven Communication Layer for Reliable Wireless Sensor Networks

Vinaitheerthan Sundaram, Jae-Woo Lee, Saurabh Bagchi, Yung-Hsiang Lu, Zhiyuan Li*

School of Electrical and Computer Engineering, (*) Department of Computer Science

Purdue University, West Lafayette, IN 47907

{vsundar,jaewoolee,sbagchi,yunglu,li}@purdue.edu

Contact Author: Saurabh Bagchi

Abstract

In many reliable Wireless Sensor Network (WSN) applications, messages have different priorities depending on urgency or importance. For example, a message reporting the failure of all nodes in a region is more important than that for a single node. Moreover, traffic can be bursty in nature, such as when a correlated error is reported by multiple nodes running identical code. Current communication layers in WSNs lack efficient support for these two requirements. We present a priority-driven communication layer, called SeNDORComm, that schedules transmission of packets driven by application-specified priority, buffers and packs multiple messages in a packet, and honors latency guarantee for a message. We show that SeNDORComm improves energy efficiency, message reliability, network utilization and delays congestion collapse in a network. We extensively evaluate SeNDORComm using analysis, simulation and real experiments. We demonstrate the improvement in goodput of SeNDORComm over a default communication layer (134.78% for a network of 20 nodes), such as GenericComm in TinyOS.

Keywords: Wireless sensor network, reliable message delivery, congestion collapse, priority driven communication, channel utilization.

1. Introduction

Wireless Sensor Networks (WSNs) are maturing to be an invaluable tool for scientific research in various fields that require real-time sensing of parameters in the physical space.

There are two motivations that lead us to consider a communication layer for reliable operation of WSNs. First, in many WSN applications, messages have different levels of urgency or importance. Intuitively the messages of high urgency should be given high priorities for transmission by the communication layer without the application having to perform the book-keeping related to the different priority messages. Examples of this phenomenon of multiple priority levels are not hard to find. In many WSN applications certain sensed events are more important or urgent than other events. In surveillance applications [6], the alert sent for an intruding pedestrian is less urgent than that sent for an intruding motor vehicle. In an indoor climate control application presence of harmful gas in the air should be reported more urgently than the current CO_2 level. Also consider the growth of error monitoring software in WSNs [8][[7][10]. The approach is to monitor for violation of some properties at runtime (such as, available message rate at a cluster head is below a threshold) and report observed errors, typically to a base station for further manual processing. The severity of errors is often different, which necessitates different priorities for the error messages. As an example, our recent error detection framework called H-SEND [9] distinguishes two kinds of error messages. The advisory error messages (e.g. messages indicating a transient interfering source went through the network) have much lower priority than severe error messages which need attention as early as possible (e.g. messages indicating a fraction of nodes without a cluster head to align with). Likewise, messages from software components on an embedded node other than the error monitoring component, such as, a time synchronization component, may vie for higher or lower priority than different error messages. In the rest of the paper, we refer to the messages with the highest priority that need immediate attention as *immediate messages* and the other messages as *deferred messages*. A priority-driven communication layer is thus highly desired but yet unfortunately unavailable in the state-of-the-art.

The second observation is that errors in WSNs are often bursty in nature. Due to limited bandwidth availability in WSNs, the possibility of congestion is high. A typical WSN has common software code running on multiple nodes in the network and they respond to a congestion situation by generating alert messages concurrently destined toward the base station. These are often referred to as debug messages in the literature including in our previous work on the H-SEND protocol [7][9]. Other transient conditions that affect multiple nodes also give rise to bursty traffic. This bursty message traffic in today's WSN communication stack may lead to congestion collapse whereby the network throughput goes to zero with no message being able to reach the base station. Effective handling of bursty traffic is therefore an important feature in our desired communication layer.

In this paper, we present the design and evaluation of a communication layer called *SeNDORComm*, targeted to the

TinyOS embedded operating system. It provides support for priorities of messages and bursty traffic. Further, SeNDORCOMM is designed to satisfy a number of requirements. First, the messages must be delivered with a high reliability for all messages regardless of priorities. Second, the communication layer must obey the resource constraints of a WSN, most relevant ones being bandwidth utilization and energy optimization. Third, there should not be starvation of the deferred messages due to giving priority to the transmission of the immediate messages. A practical requirement we impose on us is to keep the interface between the application and the communication layer as undisturbed as possible. These requirements are satisfied through the following two design choices. SeNDORComm imposes a priority based scheme for transmitting messages from a node. The priority is dependent on the urgency of the message and can be specified by the application layer (as done currently) or automatically deduced. A deadline for sending messages is imposed based on the priority to avoid starvation of low priority messages and when the deadline is reached, the message is sent by itself as an *explicit packet*[1]. Second, SeNDORComm buffers the deferred messages destined to the same node and sends them piggybacked on the immediate messages. This takes advantage of the bursty traffic behavior and that the traffic is destined to the same node. This approach is enabled by the fact that the size of the packet that can be accommodated in a WSN without significant losses due to channel conditions is often large enough to fit multiple messages for a typical payload. Moreover, we found that under a wide range of message sizes, sending a separate message is more expensive than piggybacking it on another message since the latter approach can amortize the fixed cost of synchronization (such as sending preambles or requiring an additional timeslot) and other fixed costs incurred for any message transmission. This observation has been made earlier by several researchers [2][3][4] but we are the first to leverage this in a communication layer.

Through the evaluation, we show the SeNDORComm is able to meet and go beyond the requirements mentioned earlier. The reliability of the immediate messages is statistically identical to the baseline (GenericComm, the default communication layer in TinyOS) while that of deferred messages is improved by up to 231%. A contributory factor is an automatic retransmission method for deferred messages in case of failure. SeNDORComm makes better use of the limited bandwidth by amortizing the fixed byte cost of each transmission – the preamble and the header bytes. Third, each deferred message is guaranteed to become a candidate for transmission within a delay bound that can be specified or derived from the priority of the message. This may result in an explicit packet being generated if a suitable piggybacking opportunity does not arise. Finally, the interface of SeNDORComm is kept similar to that of GenericComm with the addition of a single priority parameter to the send call and the split phase operation semantic of send being preserved.

We perform a queuing theory based analysis of SeNDORComm to determine an upper bound on the number of explicit packets generated as a function of the latency guarantee. We integrate SeNDORComm in the LEACH protocol [5], a standard data gathering protocol for a hierarchical WSN, and use it for our experiments. The first experiment shows that for sample indoor and outdoor channel conditions, the wireless channel is reliable enough to support piggybacking of multiple messages in a packet. The second experiment shows the energy advantage of SeNDORComm over GenericComm for different levels of interference−59.3% reduction in energy for high interference condition. The third experiment shows the performance of SeNDORComm under heavy load conditions which leads to a congestion collapse in a data gathering application with the current communication layer. We show that SeNDORComm is able to delay congestion collapse. This experiment shows a 134.78% improvement in goodput for a network with 20 Mica2 motes. For demonstrating scalability, this experiment is also repeated in simulation with 100 nodes. The simulation study shows a 42% improvement in goodput and a 176.5% improvement in reliability of immediate messages under heavy load.

Summarizing, the key contributions of our paper are:

1. We provide a communication layer for a WSN, called SeNDORComm, that handles prioritized messages while optimizing the energy overhead of transmission.
2. We show the benefits of piggybacking multiple messages in a packet for amortizing the constant cost of transmission while avoiding starvation of deferred messages and respecting delay bounds.

The rest of the paper is organized as follows. In Section 2 we discuss related work. The detailed design of SeNDORComm is presented in Section 3. The case study with distributed debugging is in Section 4. The analysis and discrete-event simulations are explained in Section 5. Section 6 and 7 presents the results from the testbed and the simulation. A discussion of future work is in Section 8. Section 9 concludes the paper.

2. Related Work

Considering the traditional protocol stack, it is important to note that SeNDORComm lies between application and the network layer. It is useful to keep this in mind while looking at related work.

We classify the related work into five categories. We confine ourselves to literature in WSN since these are most relevant due to the domain specific challenges.

1. *Protocols that use priorities.*
2. *Protocols that do message pooling.*
3. *Protocols that address congestion control.*

[1] Through the paper, we use the term "message" to denote the application level unit of communication and "packet" to denote the unit actually sent out on the wireless channel. With piggybacking, a packet may include one or more messages.

4. *Investigation of message size on throughput.*

1. *Priorities*. RAP [1] is motivated toward real-time communication. It is an application layer that determines message priorities to control the latency of its journey to the base station. It could be structured on top of SeNDORComm which would perform the message transmissions according to the priorities. In [18] the authors present a protocol for sensor-actor coordination to control the quality of data collection. Actor interests may have different levels of importance and sensors arrange the use of their resources according to the importance of interests.

2. *Message pooling*. AIDA (Adaptive Application-independent Data Aggregation) [2] lies between the network and the data-link layer. It buffers messages from the network layer in an FCFS queue. It aggregates messages to different degrees depending on the MAC layer contention. Being at a lower layer, it does not concern itself with message priorities and the related issue of bounding latency for messages awaiting aggregation. The widely used CSMA-based MAC layer for WSNs called BMAC [4] provides a mechanism to amortize the cost of sending long preambles before each packet. If there are multiple packets destined to a receiver, once synchronization between the sender and the receiver is established, BMAC can omit sending the preamble for all but the first packet. However, the application has to maintain a message pool destined to the same node and decide when to send the accumulated messages. SeNDORComm removes this burden from the application. SeNDORComm can benefit from BMAC's batching at each intermediate hop.

The Sensornet protocol (SP) sits between the network and the link layer and provides a minimal interface needed so that different network protocols and different MAC-layer protocols can be plugged in. It supports the network layer indicating if a message is urgent in which case it is sent without pooling. In case it is not flagged as such, SP tries to batch multiple messages. Being lower down in the protocol stack, it does not concern itself with application priorities.

3. *Congestion control*. There exists a significant volume of work at the link layer to provide congestion avoidance or control [11][12][13]. The approach is to detect or anticipate congestion and prevent a congestion collapse by appropriately backing off communication. SeNDORComm can coexist with such techniques and also with hints from the application; it can further avoid a congestion collapse. Some work at the routing layer [20] distributes the traffic load to route around congestion.

4. *Message size and throughput*. One design decision of SeNDORComm is based on the fact that throughput improves with increasing message size as long as the channel losses do not outweigh the efficiencies. The impact of message size on throughput was first studied analytically and through simulation by Akyildiz *et al.* [14]. They uncovered the optimal packet size to maximize a metric called energy efficiency which takes the energy expended per packet and the reliability of the packet into account. A datalink layer protocol called SEDA [19] show that by reducing the granularity of retransmission to a smaller block rather than the entire MAC frame or packet, throughput can be improved even in lossy channels . This work is very germane to SeNDORComm. If SeNDORComm's decision to piggyback multiple messages in a packet increases the loss rate (we empirically show that we can operate in a conservative region where it does not), then it can use SEDA to recover the affected messages rather than retransmitting the entire packet.

3. SeNDORComm Design and Implementation

SeNDORComm is a layer that sits between the application and the network layer. The network layer interface in TinyOS is called GenericComm. The radio stack for WSN applications that use SeNDORComm is shown in Figure 1. Since SeNDORComm is implemented on top of GenericComm, it inherits the portability of GenericComm to different underlying protocol layers. We elaborate on the design and implementation of SeNDORComm in this section.

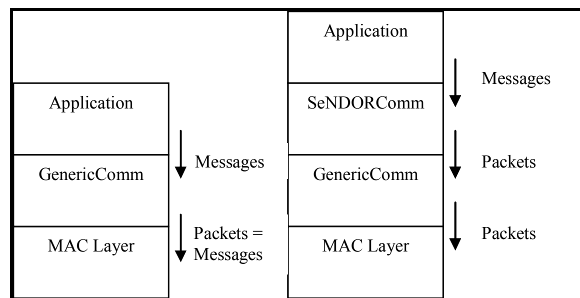

Figure 1. WSN application's radio stack interface respectively without and with SeNDORComm

3.1. Design Goals and Message Priority

We designed SeNDORComm with the following design goals in mind: 1) reduce deferred message traffic to conserve energy, 2) send critical messages promptly, and 3) keep the interface simple and close to GenericComm, so existing TinyOS applications can be easily integrated. We explain how these design goals are met.

SeNDORComm allows the application to specify priority for all messages. The allowable range of priority values is the range of a one-byte unsigned integer i.e., 0 to 255. Following the tradition of assigning lower values to denote higher priority, 0 denotes the highest priority (the immediate messages) and 255 denotes the lowest priority.

3.2. Queuing Policy

The policy for deciding when to send a message is at the heart of SeNDORComm's design. By our policy, the immediate messages are sent immediately and the lowest priority messages are not sent and are logged in the local persistent storage for later retrieval. Deferred messages are messages with priority $\in$ (1, 254) and are buffered by SeNDORComm in a priority queue. The deferred messages are stored in a priority queue. For multiple messages with the same priority, they are unordered. Later, they are either

piggybacked on other messages destined to the same node or sent out as a separate packet called an *explicit packet* as explained later. The guarantee from SeNDORComm is that a message with priority value i is always sent before or together with a message with priority value j, where $j > i$. However, if the message with priority value j is in the process of being sent when the message with priority value i arrives, then the sending is completed. Thus, the queuing discipline is a non-preemptive priority queue.

SeNDORComm Send. The pseudo code for the send and timer fired events are shown in Figure 2. To send an immediate message, SeNDORComm generates a packet and copies the immediate message into it. It piggybacks as many deferred messages as possible, the deferred messages being selected from the send queue in priority order. It then sends the packet comprising multiple messages down the stack. This corresponds to lines 3-6 in SeNDORComm.Send. Each deferred message is assigned a threshold for staying in the queue based on its priority value. In typical cases, the deferred messages in the priority queue are piggybacked on immediate messages. In cases when the deferred message has stayed for more than the threshold amount of time in the priority queue, an *explicit packet* is generated to send that message. SendQTimer.fired (Figure 2) is the periodic timer event that generates explicit packets for deferred messages that have stayed past their deadline and deletes those messages whose transmission has been attempted more than the allowable number of times. This explicit packet piggybacks as many deferred messages as possible again in priority order.

SeNDORComm Receive. When a packet is received from GenericComm, SeNDORComm demarshalls the packet to retrieve the constituent messages in the packet and delivers one message at a time to the application. Since there can be multiple messages in a single packet, SeNDORComm uses a circular list, called receiver buffer that is provided by the application at initialization, to temporarily store the messages in a packet until they are consumed by the application. By delivering one message at a time, the receive message handler in the application need not be modified from that for GenericComm.

```
Function Name: SeNDORSend.Send
Input: addr, len, urgency : integers
       msg: SC_TOS_MSG
Output: result: integer
Variables: sendQ: Priority Queue

1.    if (urgency is highest OR sendQ is full)
2.       if (radio busy)
3.          return FAIL
4.       create a packet containing msg
5.       pack deferred messages from the sendQ in the packet
6.       call GenericComm's Send to transmit over wireless channel
7.    else
8.       Enqueue msg in sendQ
9.       if (radio busy sending explicit packet )
10.         delay signalling sendDone until this packet is sent
11.      else
12.         post signalSendDoneTask( )
13.   return SUCCESS


Function Name: SendQTimer.fired
Input: None
Output: result : integer
Variables: sendQ: Priority Queue, MaxSendTries : integer

1.    delete messages that are tried MaxSendTries times
2.    age each message in the queue
3.    pick the first message,msg, that has stayed in the queue
4.    longer than threshold
5.    if ( radio not busy and explicit packet generation is not paused )
6.       remove msg from SendQ
7.       create a packet containing that msg
8.       pack deferred messages from SendQ in the packet
9.       Call GenericComm's Send to transmit over wireless channel
10.   return SUCCESS
```

Figure 2. Pseudocode of Send function and SendQTimer fired function of SeNDORComm

```
Function Name: Send.SendDone  (GenericComm)
Input: packet : TOS_MSG
       sendresult :integer
Output: result : integer
Variables: sendQ: Priority Queue

1.    If (packet is explicitly generated by SeNDORComm)
2.       turnoff explicit message in progress flag
3.       if (sendresult is FAIL )
4.          unpack packet and re-store messages in SendQ with priority 1
5.       if (application sendDone is delayed )
6.          signal SendDone with success
7.    else // immediate packet
8.       if (sendresult  is FAIL )
9.          unpack packet and store any deferred messages in SendQ
10.      signal sendDone with sendresult
11.   turn off radio busy flag
12.   return SUCCESS


Function Name: Receive.receive  (GenericComm )
Input: pkt_in : TOS_MSG
Output: pkt_out: TOS_MSG
Variables: recvQ: Circular List,
           pkt_buffer : TOS_MSG

1.    if ( recvQ is full )
2.       return pkt_in
3.    post  unpackTask  // uses pkt_in
4.    return  pkt_buffer
```

Figure 3. Pseudocode of the callback functions sendDone and receive from GenericComm to SeNDORComm.

3.3. Split-Phase Operation of Send

When an application requests SeNDORComm to send a deferred message, SeNDORComm stores the deferred message in the priority queue as long as it is not full and if no explicit packet is currently being sent. The latter

condition is to avoid application from overflowing the priority queue. It then signals sendDone to the application (Figure 3), with success indicating that the responsibility for sending the message has been taken over by SeNDORComm. If sending the deferred message on the wireless link fails due to collision or lossy link, SeNDORComm stores the message back into its priority queue with priority value 1 and attempts to send it again for a configurable number of times (3 in our experiments). If it still fails, SeNDORComm drops the deferred message. The implications of this failure handling policy are discussed in Section 8.

SeNDORComm attempts to prevent the priority queue from overflowing by rate controlling the application, the transmission attempts, and the current capacity of the wireless channel. Thus, if a deferred message send is requested and the priority queue is full, SeNDORComm sends this message immediately as part of an explicit packet. The explicit packet also drains some other messages from the queue in priority order. This allows the congested buffer condition to be partially relieved.

3.4. Interface to the application

The SeNDORComm interfaces are shown in Figure 4. In addition to the familiar send and receive interfaces, it provides a control interface SeNDORCommCtl that gives the flexibility for the application to turn on and off the explicit packet for deferred messages that have stayed in the queue past the threshold. The SeNDORSend and SeNDORReceive interfaces are very similar to GenericComm Send and Receive interfaces. The send command in the SeNDORSend interface takes an urgency parameter, which is synonymous with the priority value for the message. The sendDone callback event indicates that the application is free to reuse the buffer. Underneath, it has two different connotations—for immediate message, it has the same connotation as in GenericComm, that is, a successful transmission has been performed; for a deferred message, it means SeNDORComm has taken over the responsibility for sending the message and the application is free to proceed.

```
interface SeNDORSend {
  command result_t send( uint16_t  address, uint8_t  length,
                         SC_TOS_MsgPtr  msg, uint8_t  urgency);
  event result_t sendDone( SC_TOS_MsgPtr msg, result_t  success );
}

interface SeNDORReceive {
  event  SC_TOS_MsgPtr receive( SC_TOS_MsgPtr   msgPtr );
  command  void  receiveDone( SC_TOS_MsgPtr   msgPtr );
}

interface SeNDORCommCtl {
  command  result_t  receiveInit( SC_TOS_MsgPtr  recvQ, uint8_t  size );
  command  result_t  pause(  );
  command  result_t  resume(  );
}
```

Figure 4. SeNDORComm Interfaces: Send, Receive, Control

In SeNDORComm, the application uses the SC_TOS_Msg structure instead of TOS_Msg. The SC_TOS_Msg structure corresponds to the message whereas the TOS_Msg structure corresponds to the packet. The payload of TOS_Msg has been increased to 58 bytes (from 29 bytes) so that piggybacking is possible. In practice this would be determined by the channel conditions – what length of packet will not suffer significant losses. Empirical investigation of this question is presented in Experiment 1. However we do not recommend automatic reselection of this by SeNDORComm through sweep of the parameter space since that would involve far too much calibration overhead. Rather a conservative size can be chosen for a given class of environment (indoor line of sight, outdoor with no human presence, etc.).

The receive function in the SeNDORReceive interface has exactly the same syntax and semantics as the receive function in GenericComm's Receive interface except for the return value semantics. The return value of GenericComm receive command is a buffer where the next received message can be stored, while the return value of the SeNDORReceive function is indication of whether the passed buffer has been processed by the application or not The receive handler, the implementation of the receive function, for SeNDORReceive can return the message pointer passed to it to indicate the message buffer is processed or a NULL to indicate the message is being processed. In the latter case, the application uses the receiveDone callback event to inform SeNDORComm later that the message buffer is processed.

3.6. Implementation Details

2	|M1|+1	|M2|+1	M1 Type	M1 Payload	M2 Type	M2 Payload

Figure 5. An example for packing messages into a packet's payload. Here, the packet contains 2 messages (M1 and M2). |Mi| denotes Mi's payload length.

The implementation of SeNDORComm in TinyOS has two significant features. First is the use of a heap structure with pointers to implement a priority queue for the messages. It supports efficient (O(log (number of messages))) operations for inserting and deleting messages from the queue. The use of pointers avoids unnecessary message copies. The second feature is packing multiple messages in a packet. It amortizes the common information – the destination, since only messages destined to the same node are piggybacked, CRC, etc. An example is shown in Figure 5. The type is used to invoke the appropriate receive handler for each message. SeNDORComm has a low memory footprint of around 100 bytes and small code size of around 4 KB. Table 1 presents the code size and memory footprint of a LEACH-based data gathering application with and without SeNDORComm. The values are obtained by default compiler settings in TinyOS. Three configurations of buffers in SeNDORComm are shown, namely, minimum, medium or typical and large amount of buffers. To utilize the SeNDORComm functionality, the sender-side priority queue size must be at least 1 entry and the receive list size must be at least 2 entries to allow aggregation.

Table 1. Code and Memory Footprint of SeNDORComm integrated with LEACH

Components	ROM Size	RAM Size	Buffers Size
LEACH with GenericComm and Debugging	17884	811	0
LEACH with SeNDORComm and Debugging (1 buffer priority queue, 2 buffer receiver list)	21812	1118	138
LEACH with SeNDORComm and Debugging (5 buffer priority queue, 4 buffer receiver list)	21812	1351	426
LEACH with SenDORComm and Debugging (10 buffer priority queue, 4 buffer receiver list)	21812	1596	676

4. Case Study: Distributed Debugging using HSEND and LEACH

We study the usage and performance of SeNDORComm in a WSN running LEACH [5] as the data gathering protocol. In LEACH, the nodes organize themselves into clusters, with one node in each cluster acting as the cluster head for one round. Each round is divided into election slots, which are used by the sensing nodes to send data to the cluster head. The self-elected cluster heads advertise their status. Nodes that are not cluster heads choose one of the cluster heads to join depending on received signal strength.

Checking invariants during run-time is an effective and widely used approach for debugging distributed systems [15][16]. In distributed debugging, invariants are checked at run-time. To check global invariants, debug messages are exchanged among nodes and the effectiveness of distributed debugging is dependent on the relevance of the invariants chosen. An idea gaining ground in the WSN community is that distributed debugging is important for robust deployments of these networks [7][8][9]. However, there is concern for resource usage due to the additional network traffic introduced. SeNDORComm is an effort in the direction of providing a primitive for distributed debugging. In our earlier work, we proposed H-SEND (Hierarchical Sensor Network Debugger), a distributed debugging system for WSNs, that reduces significant amount of debug traffic by evaluating invariants as close to the source of the error as possible. However, to evaluate remote invariants, debug messages are still exchanged in H-SEND.

Since the interface of SeNDORComm to application is similar to that GenericComm, the modifications required to our LEACH with HSEND code to use SeNDORComm were minimal. We use LEACH with H-SEND on top of SeNDORComm and GenericComm and compare their performance. The detailed performance results are shown in Section 6.

5. Analytical Evaluation

To evaluate SeNDORComm, we perform an analysis to derive *an upper bound* on the additional traffic injected into the network due to the explicit packets generated for the deferred messages The analytical result could be useful for guiding the choice of the deadline for deferred messages in a real deployment.

5.1. Assumptions and Notations

We make the following assumptions to make the analysis tractable. (1) A three-level hierarchy is assumed with sensing node, cluster head, and base station. (2) The clusters are all identical. Thus, analyzing a single cluster is sufficient. (3) The rate of immediate messages generated at a node is exponentially distributed with mean $1/\mu_i$ and for deferred messages the exponential distribution has mean $1/\Lambda$. (4) The priority of deferred messages is uniformly distributed in [0, r]. (5) In one packet only one deferred message can be piggybacked with an immediate message. Without this assumption, the queuing theory formulation would be overly complex since the service time would depend on the state of the queue.

Let n denote the number of nodes in the network, k the number of clusters, and m the number of nodes in each cluster (= n/k). Let f denote the compression factor at the cluster head i.e., the data size arriving at the cluster head by the data size sent by the cluster head to the base station. Let the rate of deferred messages with a given priority be $\lambda = \Lambda/(r+1)$. Let the transmission time of a packet in a CSMA network be exponentially distributed with rate μ_t.

5.2. Upper bound on the overhead traffic generated by individual nodes

The key observation for the analysis is that we can view the priority queue maintained by SeNDORcomm as an M/G/1 non-preemptive (head-of-the-line) priority queue with the immediate messages being considered as the server. A deferred message is *serviced* when an immediate message arrives and piggybacks the deferred message. So, a deferred message at the top of the queue can be considered to be under service until an immediate message arrives, piggybacks it, and gets sent out on the wireless channel. Thus, the service time (B) is the sum of inter-arrival time of immediate messages and the transmission time for the packet. Thus B follows a hypo-exponential distribution with parameters μ_i and μ_t.

To keep the analysis tractable, we do not take into account the "draining" effect of explicit packets on the priority queue. This is one of the factors pushing the final result to be an upper bound. The expected number of explicit messages generated is equal to the expected rate of messages arriving at the queue times the probability that a message waits more than the threshold wait time in the priority queue.

Let W_a be the random variable denoting the actual waiting time of a deferred message, W_p the waiting time for the deferred message in queue with priority p and B_{imm} the inter-arrival time of immediate messages.

$$W_a = W_p + B_{imm} \quad (1)$$

$$E[W_a] = E[W_p] + (1/\mu_i) \quad (2)$$

Assuming B_{imm} and W_p are independent.

$$\sigma_{W_a}^2 = \sigma_{W_p}^2 + (1/\mu_i^2) \quad (3)$$

Let $E[N]$ denote the expected number of explicit messages generated by all queues in unit time and $E[N_p]$ the expected number generated in queue p per unit time. γ_p represents the threshold waiting time in queue p, $p = 1, \ldots, r$. Let γ represent the base threshold waiting time. $\gamma_p = \gamma + (p-1)/2$, $p = 1, \ldots, r$

$$E[N] = \sum_{i=1}^{r} E[N_p] \quad (4)$$

$$E[N_p] = \lambda P(W_a > \gamma_p) \quad (5)$$

To solve Equation (5), we need the distribution of waiting time W_a, which requires the probability distribution of waiting time W_p for queue p. The priority queue with non-preemptive priority for *M/G/1* system has been well-studied. The first two moments of W_p are given in [17]. With B_i as the service time distribution of the queue, we have

$$E[W_p] = \frac{\sum_{i=1}^{r} \lambda_i E[B_i^2]}{2(1-\sigma_{p-1})(1-\sigma_p)} \quad (6)$$

$$E[W_p^2] = \frac{\sum_{i=1}^{r} \lambda_i E[B_i^3]}{3(1-\sigma_{p-1})^2(1-\sigma_p)} + \frac{\left(\sum_{i=1}^{r} \lambda_i E[B_i^2]\right)^2}{2(1-\sigma_{p-1})^2(1-\sigma_p)^2} + \frac{\left(\sum_{i=1}^{r} \lambda_i E[B_i^2]\right)\left(\sum_{i=1}^{r-1} \lambda_i E[B_i^2]\right)}{2(1-\sigma_{p-1})^3(1-\sigma_p)} \quad (7)$$

where,

$$\sigma_p = \sum_{k=1}^{p} \rho_k = \sum_{k=1}^{p} \frac{\lambda}{\mu_s} = \frac{p\lambda}{\mu_s} \quad (8)$$

B_i follows HYPO(μ_i, μ_i), for all p. Therefore, we can obtain the first, second, and third moments of B_i. Putting these in (6) and (7), we obtain E[W_p] and $\sigma_{W_p}^2$ and from this E[W_a] and $\sigma_{W_p}^2$ using (2) and (3).

The mean and variance of waiting time distribution can be used to obtain an upper bound for $P(W_p > \gamma_p)$ by using Chebyshev's inequality. For all $\varepsilon>0$, the inequality gives,

$$P(|W_a - E[W_a]| > \varepsilon) \le \frac{\sigma_{w_a}^2}{\varepsilon^2} \quad (9)$$

Rewriting Chebyshev's inequality for $\gamma_p > E[W_a]$ (which is reasonable since the threshold should be longer than the average waiting time, otherwise the network will be overwhelmed with explicit packets), we have (using (5)):

$$E[N_p] = \lambda P(W_a > \gamma_p) \le \lambda \frac{\sigma_{W_a}^2}{(\gamma_p - E[W_a])^2} \quad (10)$$

where E[W_a] and $\sigma_{W_a}^2$ are given by (2) and (3).

5.3. Upper bound on relative network-wide overhead due to debugging

The traffic in the network is defined as the number of packets generated in the network per unit time. Let ζ_{no_debug} and ζ_{with_debug} denote the total traffic in a cluster respectively without and with SeNDORComm. The subscript "with debug" indicates that as an example the analysis takes all deferred messages are generated due to runtime monitoring (or debugging).

For ζ_{no_debug}, the traffic in the cluster, includes the packets generated by m cluster nodes and the packets sent by the cluster head.

$$\zeta_{no_debug} = m\mu_{data}(1+1/f) \quad (11)$$

With SeNDORComm, at a cluster node the traffic is due to the immediate data messages, the highest priority messages arriving with rate λ, and the explicit packets.

$$\beta_{node} = \mu_{data} + \lambda + E[N] \quad (12)$$

At the cluster head, we get an upper bound if every deferred message is forwarded to the base station instead of being consumed locally. This can occur say if the message is related to a detected error that needs to be forwarded to the base station for action. The traffic generated by a cluster head β_{head} is given by,

$$\beta_{head} = \frac{(m * \mu_{data})}{f} + (m+1)\lambda + (m+1)E[N] \quad (13)$$

The total traffic generated in a cluster with run-time debugging (ζ_{with_debug}) is given by,

$$\begin{aligned}\zeta_{with_debug} &= \beta_{head} + m\beta_{node} \\ &= \zeta_{no_debug} + (2m+1)\lambda + (2m+1)E[N]\end{aligned} \quad (14)$$

Therefore, the upper bound on the normalized overhead generated due to run-time debugging ($\zeta_{overhead}$) is,

$$\zeta_{overhead} = \frac{\zeta_{with_debug}}{\zeta_{no_debug}} = 1 + \frac{(2m+1)\lambda + (2m+1)E[N]}{\zeta_{no_debug}} \quad (15)$$

5.4. Results: Analysis and Discrete Event Simulation

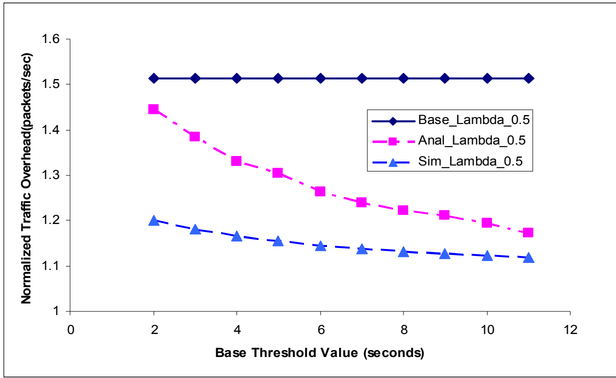

Figure 6. Normalized traffic overhead due to SeNDORComm and GenericComm (Base)

Using the above analysis, we plotted the normalized overhead traffic in a cluster against the base threshold waiting time (γ). Given a γ, γ_p is defined as $\gamma + (p-1)/2$, p is the priority value. For the plot μ_{data} = 1 message/s. The maximum packet size in SeNDORComm is 65 bytes (58 bytes payload + 7 bytes header). For mica2 nodes running B-MAC, the size of preamble at 2.2% duty cycle is 1212 bytes. Therefore, the minimum transmission time of nodes running at 2.2% duty cycle is 0.511 seconds at 20kbps. This gives us μ_t = 1.96 messages/s. A plot is drawn for high (0.5 messages/s) deferred message arrival rates and is shown in Figure 6.. The overhead incurred by the baseline approach in which every deferred message is sent to the cluster head immediately (*aka* GenericComm) can be calculated by using $E[N] = \Lambda$ in Figure 6. Normalized traffic overhead due to SeNDORComm and GenericComm (Base). These overheads are shown with the horizontal lines in Figure 6. Normalized traffic overhead due to SeNDORComm and GenericComm (Base). We also performed a discrete event simulation of the M/G/1 non-preemptive priority queue to validate that the analysis gives the upper bound. The simulation also cannot take into account the effect of an explicit message in draining the queues and therefore, it also gives an upper bound, albeit a tighter one than the analysis.

The result shows that for a high load scenario, SeNDORComm reduces the overhead of debugging by 26% even with a reasonably short baseline deadline of 10 seconds for explicit packet generation. The gains are less (about 7%) for a medium load scenario (0.1 deferred messages per second).

6. Experimental Evaluation

6.1. Experiment 1: Feasibility of Piggybacking

The objective of this experiment is to determine if the fundamental requirement of SeNDORComm, namely the ability to pack multiple messages in a packet is met in a sample indoor setting. For this, we assess the quality of the channel losses in an indoor laboratory setting as the payload size changes and repeat it for an outdoor setting.

Two Mica2 nodes are kept at approximately 7m distance in the lab with no interfering node present. One node is the sender and the other is the receiver. The sender sends packets at the rate of 4 packets/second destined for the receiver. The sender uses 8 different payload lengths between 28 and 240 bytes. These match closely with the default payload length in TinyOS and the maximum allowed. The sender sends packets of the 8 different payload lengths one after another in a 2 second period and repeats this over the span of the entire measurement period (8 hours). By cycling through payload lengths of size in a 2 second period, we ensure that the channel conditions remain same for all the payload lengths. We classify the result into packets lost on the channel, packets corrupted on the channel (incorrect CRC), and packets received correctly. The results are shown in Figure 7. The data is reinterpreted for different time points to see how the channel conditions vary over time. The results are shown in Figure 7. Each result represents the average over the preceding hour.

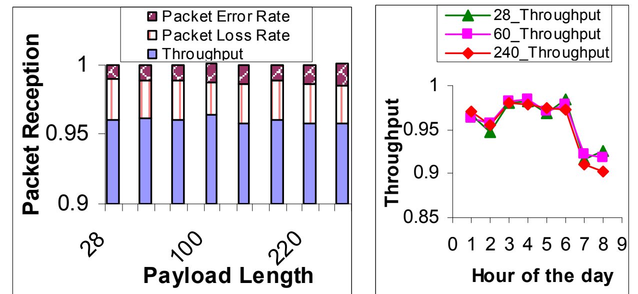

Figure 7. Effect of wireless link quality with varying packet size is shown in left figure. The variation of throughput over every hour during the span of experiment 1 is shown in the right figure.

In Figure 7, the throughput reduces gradually as the packet size is increased since larger packets are more susceptible to be corrupted or dropped. However, the reduction in throughput is insignificant. For example, the reduction in throughput of 240 byte payload compared to 28 byte payload is only 1.44% (from 91.72% to 90.28%). We observe that packet error rate is considerably smaller than packet loss rate and it increases very slowly with packet size (1.6% at 240 bytes compared to 0.1% at 28 bytes). We also conduct the experiment in an outdoor setting with sender receiver set 7 m apart, line of sight, and in the presence of brick walls and railings around the nodes. The throughput stayed almost constant with payload size (95.7% at 240 bytes versus 96.1% at 28 bytes).

In Figure 7, we notice that the link quality varied over time for the indoor setting but always affecting the different payload sizes in almost equal measure. Also, the throughput is always over 90%. For the sake of clarity, we show the result only for three payload lengths though all payload lengths had the same trend.

Thus this experiment shows that it is possible to pack multiple messages in a packet for the representative indoor and outdoor environments. If packet loss is a concern, then a

strategy as in SEDA [19] can be adopted whereby individual messages within a packet are recovered rather than the entire packet.

6.2. Experiment 2: Energy Expenditure under Interference

In this experiment, we evaluate the energy savings of SeNDORComm compared to GenericComm in the presence of interfering traffic.

Two Mica2 motes are kept at approximately 5 m distance and at 1 m height from the floor. The sender attempts to send 200 unique messages of which 25% are immediate and 75% are deferred. The sender has a retransmission mechanism which retries every message three times before dropping it. For SeNDORComm, the sender retransmits only the immediate messages to have a fair comparison with GenericComm case (since the SeNDORComm layer itself takes care of retransmitting the deferred messages thrice). The sender attempts to send a unique message every second if the radio is free (i.e., an earlier retransmission is not in progress); else, it waits for the next second. Therefore, the experiment period is the time taken to send 200 messages. The experiments are run in BMAC's LPL mode 3 (corresponding to 11.5% radio duty-cycle).

We perform three sets of experiments—with no interfering node, 3 interfering nodes and 5 interfering nodes. The second and the third can be taken to emulate low and high contention networks respectively. In each set, the experiment is repeated 6 times for each of SeNDORComm and GenericComm, which gives acceptably low variance.

To measure the current used by the mote, the sender node was connected to the HP Agilient 3458A Multimeter over the span of entire experiment, which was 6 minutes. The current was sampled every 5 milliseconds. The energy spent by the mote is the product of current measurement, voltage (3 volts) and time (5 milliseconds). The total energy spent by the mote over the span of the experiment is the sum of the energy of all samples.

We have used the two performance metrics: (1) the total transmission energy spent per useful receive byte, where useful bytes are from messages that are not duplicates, (2) fraction of deferred and immediate messages received correctly.

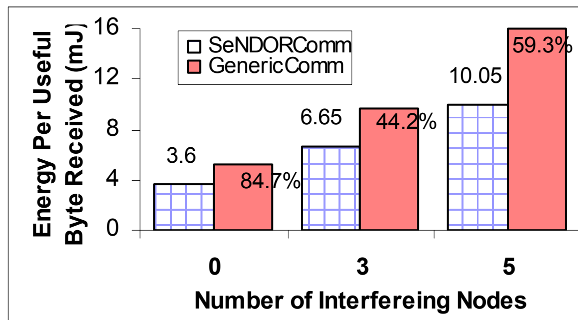

Figure 8. Energy spent by the sender node per useful byte received for different levels of interference in network. The percentage numbers on GenericComm denotes the increase relative to the corresponding SeNDORComm case. The number on SeNDORComm denotes the absolute energy value.

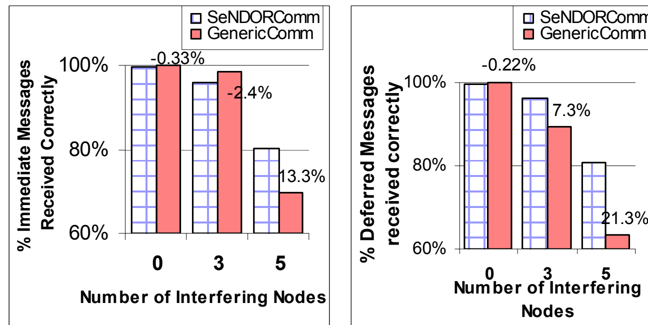

Figure 9. Fraction of immediate and deferred messages received successfully. The percentage number on GenericComm denotes the increase relative to the corresponding SeNDORComm case.

In Figure 8, we see that the energy required per useful byte received is considerably lower for SeNDORComm (43.5%, 44.1%, and 59.1% for low, medium, and high interference). This energy savings is due to piggybacking deferred messages on immediate messages, which reduces the fixed overhead cost associated with sending a packet. When the interference from other nodes increases, the energy spent increases for both communication layers due to the increased losses from collisions. However the increase is faster for GenericComm. By prioritizing and batching, SeNDORComm sends fewer packets in the network thereby reducing packet collisions and retransmissions

Since SeNDORComm piggybacks deferred messages on immediate messages, it increases the possibility of immediate message getting dropped due to channel losses. However, we see from Figure 9 that the percentage of immediate messages dropped is very low. The simple retransmission mechanism used by the application compensates for occasional losses. When interference increases, SeNDORComm achieves higher throughput with immediate messages than GenericComm as the packet losses due to collision starts dominating over channel losses.

Finally, we notice that the fraction of deferred messages received by SeNDORComm is much higher than in GenericComm (Figure 9(b)). Moreover, the fraction decreases much slower than in GenericComm with increasing amount of interference. This is due to the fact that deferred messages are piggybacked in SeNDORComm rather than each being sent as a separate packet. This causes less network contention and hence fewer losses.

6.3. Experiment 3: Handling Heavy Load

In this experiment, we evaluate the ability of SeNDORComm to handle bursty traffic and delay congestion collapse when the network is heavily loaded. Distributed debugging introduces additional traffic into the network. The additional traffic can be significant under many different scenarios, such as, a change in the environment that results in multiple concurrent invariant violations and correlated

failure of several sensor nodes. In these scenarios, it is important to detect and locate the error in the network promptly for a possible recovery. To achieve this, it is necessary to have the critical error information reach the base station. This is particularly difficult for the baseline communication layer to handle because the problem often manifests itself at the time when the available bandwidth is also constricted.

The performance metrics of interest are (1) *Goodput*, the rate of immediate messages that reaches the base station (2) T*ransmission success ratio*, the ratio of the number of messages received by all nodes in the network to the number of message sends attempted by all nodes including retransmission. This indicates how efficiently the channel is used for communication. (3) *Reliability of immediate (deferred) messages*, the total number of immediate (deferred) messages received by all nodes successfully out of the total number of immediate (deferred) messages sent by all nodes.

We created a 21 node network of Mica2 motes arranged in a 2x1 grid configuration with all nodes in the communication range of each other. We used LEACH [5] as the leader election protocol in TinyOS. Each round has 27 equal timeslots, 20 for sending data messages and 7 time slots for cluster formation. Each slot is 2 seconds long. We used 2 clusters and 9 nodes join a cluster on average. Therefore, each node gets 2.2 timeslots per round. The cluster head has a compression factor of 3, i.e., for every 3 messages it receives, it sends one to the base station. We created a simple WSN application that sends one data message to the base station in its designated slot. Each data message is 14 bytes, debug message is 8 bytes, and the maximum payload length is 58 bytes for SeNDORComm. The application ran respectively on top of GenericComm and SeNDORComm.

We used H-SEND [9] to create debug message traffic. H-SEND has an invariant that monitors the rate of successful transmision of sensed data (immediate messages) at each node. If the rate is below a certain threshold, it generates an error message with priority value 3. We set the threshold to be slightly higher than the node's normal sensed data rate so that on average a debug message is generated at every check. We vary the frequency of checking the invariant to vary the load in the network.

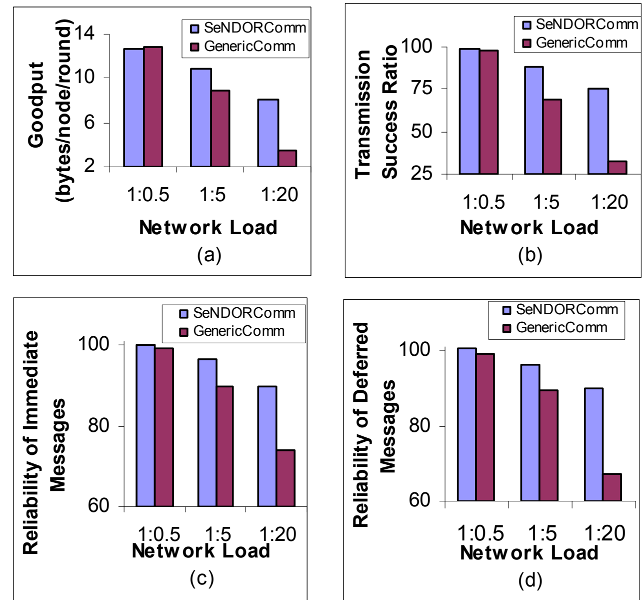

Figure 10. Behavior of SeNDORComm and GenericComm under varying load conditions on a 21 node test bed network

LEACH does not have a queue to store messages. A WSN application using LEACH has to make a choice between queueing the deferred messages generated until its timeslot or send the messages as and when generated. If the application implements a queue, queueing all messages can delay transmitting important messages such as sensed data messages. Moreover, due to limited storage available in WSN, queue overflow can occur frequently even under moderate load conditions. The queue overflow can be handled either by dropping a message or by sending a message out of the TDMA schedule. Moreover, under heavy load conditions, the benefits of implementing a simple FIFO queue in a WSN application running on top of LEACH are limited. Therefore, in our WSN application on top of LEACH, the application passes on the messages to the lower layer as and when generated. GenericComm sends the message off right away while SeNDORComm performs batching. We kept the slot size as 2 seconds, large enough to send up to 13 messages (each packet in BMAC with LPL mode 3 takes approximately 150 milliseconds). When a message transmission fails (as reported by SeNDORComm or GenericComm), the application retries the transmission three times before the message is discarded. Our results shown below indicate that the reliability is not affected by sending messages out of schedule unless the network is heavily congested.

We ran all the experiments for 20 rounds. We observe the three output metrics mentioned above for SeNDORComm and GenericComm for varying load conditions. The variation in load is shown as the ratio of number of data messages (immediate messages in this case) to the number of debug messages (deferred messages in this case) generated. Figure 10 shows the behavior of SeNDORComm and GenericComm under light (1:0.5), medium(1:5) and heavy

load(1:20) conditions. By piggybacking on immediate messages, SeNDORComm increases the likelihood of an immediate message being corrupted in the wireless channel. However, with a simple retransmission scheme, we see that for light load the reduction in goodput for SeNDORComm is low (-1.55%) and the reliability is almost the same for both (0.79% more than GenericComm). This corroborates our experiment 2 results in Section 6.2. Moreover, the goodput improves considerably as the load in the network increases (20.67% and 134.78% for medium and heavy loads). This is because the congestion is higher in GenericComm which affects the successful reception of immediate messages. In SeNDORComm, the batching of multiple messages into larger-sized packets alleviates the congestion to a certain extent. Likewise, the transmission success ratio of SeNDORComm improves (1.18%, 28.5%, and 131.99% for the light, medium and heavy loads) as it uses the network bandwidth more efficiently and therefore cuts down on the fruitless message sends that would collide and be lost on the congested wireless channel. Under heavy load, we see that the transmission success ratio for GenericComm has been reduced to 33% and this indicates congestion collapse as each message has to be sent four times.

In Figure 10(c), we see that with SeNDORComm the reliability improves as the load increases for both immediate messages (0.79%, 7.13%, and 21.20% for light, medium, and heavy loads) and deferred messages (1.53%, 7.24%, and 33.9% for the three loads). When the load is less, the packet losses are mainly due to channel losses and when the load increases, losses due to congestion start dominating. Hence, we see increased reliability benefit with SeNDORComm as the load increases.

7. Simulation Evaluation for Large Networks

To evaluate SeNDORComm for large WSNs, we used TOSSIM [22] to simulate experiment 3 for a 100 node MICA2 network. We used the same implementation of LEACH but with different parameters to scale it to 100 nodes. LEACH as described in the literature is not scalable to 100 nodes, our target network size, due to the requirement that every node is within communication range of each other resulting in interference between the clusters. To handle this, we increased the number of time slots allowed for sending the JOIN message and provided an application-level random back off mechanism to prevent collisions between multiple nodes sending in the same slot. Without this modification, only about 5% of the nodes were able to take part in data upload. The parameters we used are 5 clusters and 20 timeslots for sending data messages and 10 slots for cluster formation. The slot size was increased to 10 seconds to avoid inter-cluster collisions. To simulate wireless channel losses, we injected packet losses into the simulation as observed in our experiment (8%) described in Section 6.1.

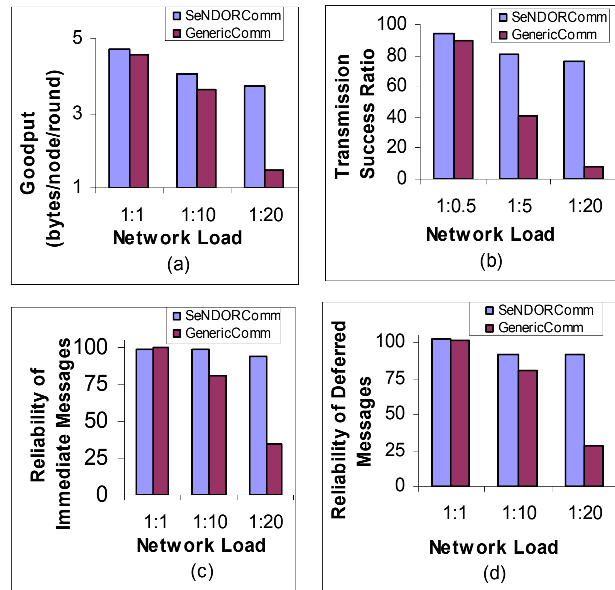

Figure 11. Behavior of SeNDORComm and GenericComm under varying load conditions on a 100 node network simulation.

Similar to experiment 3, we varied the network load by varying the number of debug message generated by HSEND. For each ratio, we ran the simulation for 20 rounds and averaged over the rounds for the results shown in Figure 11.

The results from the test bed experiment and the simulation follow a similar trend. We see that SeNDORComm improves the goodput (3.29%, 12.09%, and 154.42% for light, medium, and heavy loads) and transmission success ratio (4.86%, 99.48%, and 830.98% for the loads) as the load increases. We see that under light load conditions the reliability of immediate messages in SeNDORComm is slightly less than that of GenericComm (-0.79%) owing to the channel losses. However, the reliability increases under heavier loads for both immediate messages (22.05% and 176.35%) and for deferred messages (13.68% and 231%) for medium and heavy loads.

Under heavy load, we observe a congestion collapse in GenericComm. Thus can be concluded from the reliabilities achievable under this load. We can conclude that SeNDORComm is able to handle high loads better and delay congestion collapse with a 42% goodput improvement under congestion collapse conditions in GenericComm.

8. Discussion

Here we discuss some issues with the current design of SeNDORComm and methods to improve on them. First, when SeNDORComm signals sendDone to the application, it takes over responsibility for sending the message out. However, if it fails a designated number of times, then the message is dropped and the application receives no notification. One may argue that if this is a critical message for the application, it should have been sent it as a highest priority message in which case sendDone has the expected semantic that the message was transmitted successfully to the

next node. Alternately, we can add a third phase to the split phase send operation whereby the node gets a later callback event with a success status when the message is sent successfully, or a failure status if it is not sent successfully.

Second, the design of SeNDORComm is based on the premise that the data flows in the same direction as the debug messages. It is conceivable that there are applications where any to any communication between any two nodes in the network is frequent, while the debug messages still *always* need to flow to the base station for further action. In such a case, SeNDORComm should generate an explicit packet for the debug messages if it does not expect a data message to the base station soon. The expectation is based on a modeling of the traffic pattern in the network. The current mechanism of generating an explicit packet if the threshold wait time is crossed would be useful in this proposed design.

Third, SeNDORComm provides a guarantee that a message send will be attempted by the threshold waiting time. The guarantee *does not* cover delivery or even a successful send attempt. The guarantee is a weak one for several practical reasons—the condition of the wireless channel cannot be predicted and the single timer used for aging messages in the queue has a fixed granularity. To improve matters, SeNDORComm could estimate the channel condition based on its transmission attempts and try sending a message in advance of the deadline based on an estimation of the lossiness of the channel. Also, when the timer for a given message expires, SeNDORComm can search the current queues to piggyback, on the explicit packet being generated, all the messages with expired deadlines.

9. Conclusion

In this paper, we have presented the design and implementation of a communication layer called SeNDORComm that can handle messages with different priorities. It can buffer and piggyback messages which are not immediate so as to optimize the wireless channel usage. It respects latency bounds within which a message needs to be transmitted and it does not starve lower priority messages. Through experiments on a sensor network testbed, we show that packing multiple messages in a packet is possible without significant losses and the efficient use of the wireless channel results in lower energy consumption and increases the reliability of the end-to-end communication over the current default communication layer called GenericComm. In future work, we will be diagnosing problems in WSNs by correlating error messages. In addition, we are developing a compiler to automatically inject invariants in an application.

10. References

[1] Lu, C., Blum, B. M., Abdelzaher, T. F., Stankovic, J. A., and He, T. RAP: A Real-Time Communication Architecture for Large-Scale Wireless Sensor Networks. In RTAS '02.

[2] He, T., Blum, B. M., Stankovic, J. A., and Abdelzaher, T. AIDA: Adaptive application-independent data aggregation in wireless sensor networks. In ACM Trans. on Embedded Computing Sys. pp. 426-457, May 2004.

[3] J. Polastre, J. Hui, P. L. J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A Unifying Link Abstraction for Wireless Sensor Networks," SenSys 2005.

[4] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in Sensys, pp. 95-107, 2004.

[5] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan, "An application-specific protocol architecture for wireless microsensor networks," IEEE Trans on Wireless Communications, vol. 1, pp. 660-670, 2002.

[6] T. He *et al.*, "VigilNet: An integrated sensor network system for energy-efficient surveillance," ACM Transactions on Sensor Networks, pp. 1-38, February 2006.

[7] Herbert, D., Sundaram, V., Lu, Y., Bagchi, S., and Li, Z. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. ACM Trans. on Autonomous Adaptive Systems*,* Sep. 2007.

[8] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," Sensys 2005.

[9] D. Herbert, Y. H. Lu, S. Bagchi, and Z. Li, "Detection and Repair of Software Errors in Hierarchical Sensor Networks," SUTC, pp. 403-410, 2006.

[10] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," EWSN, pp.121-132, 2005.

[11] B. Hull, K. Jamieson, and H. Balakrishnan, "Mitigating congestion in wireless sensor networks," Sensys 2004 .

[12] A. Woo and D. E. Culler, "A transmission control scheme for media access in sensor networks," ACM Mobicom, pp. 221-235, 2001.

[13] C. Y. Wan, S. B. Eisenman, and A. T. Campbell, "CODA: congestion detection and avoidance in sensor networks," Sensys, pp. 266-279, 2003.

[14] Y. Sankarasubramaniam, I. F. Akyildiz, and S. W. McLaughlin, "Energy efficiency based packet size optimization in wireless sensor networks," in IEEE Workshop on Sensor Network Protocols and Applications (SNPA), pp. 1-8, 2003.

[15] M. Zulkernine and R. E. Seviora, “A Compositional Approach to Monitoring Distributed Systems,” IEEE International Conference on Dependable Systems and Networks (DSN'02), pp. 763-772, 2002.

[16] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach.," SRDS, pp. 18-30, 2004.

[17] R. G. Miller, "Priority Queues," The Annals of Mathematical Statistics, vol. 31, pp. 86-103, 1960.

[18] Chatzigiannakis, I., Kinalis, A., and Nikoletseas, S. Priority based adaptive coordination of wireless sensors and actors. Q2SWinet '06.

[19] Ganti, R. K., Jayachandran, P., Luo, H., and Abdelzaher, T. F. Datalink streaming in wireless sensor networks. In SenSys, pp. 209-222, 2006.

[20] T. He, J.A Stankovic, C. Lu, T. Abdelzaher, "SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks," ICDCS, 2003.

[21] A. Arora *et al.*, “A line in the sand: a wireless sensor network for target detection, classification, and tracking,” Computer Networks, pp. 605-634, December 2004.

[22] Levis,P., Lee, N., Welsh, M., and Culler, D. 2003. TOSSIM: accurate and scalable simulation of entire TinyOS applications. ACM SenSys 2003.