

Priority Scheduling in TinyOS - A Case Study

Venkita Subramonian, Huang-Ming Huang, Seema Datar, Chenyang Lu
{venkita,hh1,sd7,lu}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis. MO

December 16, 2002

Abstract

In recent years, networked sensors are finding use in a variety of different applications ranging from temperature monitoring to battlefield strategy planning. Advances in fabrication techniques have led to the development of sensor-actuator devices called MEMS. It has now become possible to move software closer to where the “action” is, i.e. the sensors themselves. These sensor devices typically have a micro-controller, instruction and data memory, a radio module for wireless communication and an operating system. These devices are severely resource constrained in terms of memory, processing power and energy, since most of these devices are battery driven. A sensor network consists of a number of sensors spread across a geographical area. Each sensor has wireless communication capability and sufficient intelligence for signal processing and networking of the data.

In this paper, we discuss a mechanism to achieve prioritized task scheduling in TinyOS. We investigate into the details of how the task scheduling is currently done in TinyOS and the adverse effects that this could cause. We show use-cases where certain “important” tasks could be given less preference when compared to other “not-so-important” tasks. We propose a new mechanism at the programming model level to introduce the concept of priority for a task and explain how the scheduling module has to be changed to enforce this priority scheme. Finally, we show empirical results to justify our solution.

Keywords: Sensor Networks, Scheduling, Wireless Networks, TinyOS

1 Introduction

We are currently in the midst of a distributed revolution, where distributed ways of communicating, processing, sensing and computing are dislodging more traditional centralized architectures. The trend is to go away from a centralized, super-reliable single-node platform to a dense and distributed multitude of cheap, lightweight and potentially less reliable individual components that, as a group, are however capable of far more complex tasks and inferences than any individual super-node.

A classical example of this is in distributed sensing, where it is desirable to have high sensor density for reliability, accuracy and cheaper deployment. Advances in device technology, networking and information processing have allowed the emergence of wireless sensor network technology: highly reliable, modular, ubiquitous devices that can form a network. Each sensor device typically comprises of a microcontroller, instruction and data memory, a radio module for wireless communication and an operating system. These devices have the capability to sense elements of the environment, make computations, and communicate with other devices/nodes or a centralized observer. However, the devices are severely resource constrained in terms of memory, processing power and energy, since most of these devices are battery driven.

Sensor networks typically are multi-hop networks where data propagates through multiple sensors to the base station. The topology could be designed so that the data is compiled and aggregated before being sent to the base station or each node may send the raw data to the base station where it is compiled and analyzed to derive the relevant information. Considering the above mentioned constraints of the sensors, the efficiency and effectiveness of the nodes needs to be studied during higher processing requirements, due to heavy routing and local traffic.

This paper is a case-study of the effect of local computation on packet processing in sensor networks. In the process we found that there are situations when a sensor node could become overloaded. We devised a way to relatively increase the throughput of the node on overload and finally conducted experiments to verify the enhancement. The rest of this paper is organized as follows: Section 2 explains the problem of overload in a sensor network and describes the TinyOS packet processing mechanisms in detail and some pitfalls. Section 3 describes our proposed solution to these pitfalls. Section 4 describes the experiments that we conducted and show results to justify our enhancements. We conducted our case study using Berkeley MICA motes running TinyOS 1.0 [1].

2 Overload in sensor networks

Sensor networks use radio signals for communication between nodes. Typically, a node in a sensor network would have three tasks at hand:

- Receive packets from other nodes to be forwarded for routing purposes.
- Send packets that are received for forwarding.
- Process the locally sensed data and send the data.

The volume of the above tasks would vary depending on whether the nodes are sending the raw data to the base station or the data is being aggregated and processed before being sent to the base station. It can be expected that the nodes would have more of routing tasks in the former while it would need to do comparatively more local processing in the latter case for aggregation of the data. An overload would typically happen when the node has much more processing to be done than it can handle. In the case where the nodes are sending the raw data, an overload could happen if the rate at which the nodes are transmitting data is very high or if the network is very dense leading to high traffic at each node. In the case where the data is being aggregated, an overload scenario could happen if the volume of data to be processed is very large. In order to gracefully handle overload conditions, there should be some mechanism by which critical tasks are still executed while the non-critical tasks are not. We investigate in to the current mechanisms for scheduling tasks in TinyOS 1.0, extend the mechanisms so that the critical tasks are given more preference over the non-critical ones.

2.1 TinyOS overview

The following discussion is based on the Berkeley motes running TinyOS 1.0. TinyOS uses an event-driven layered architecture, where the lower layer sends events to the upper layer as shown in Figure 1.

TinyOS is interrupt driven and typically there are two kinds of interrupts in TinyOS - clock and radio. The clock/timer interrupt occurs periodically and the radio interrupt occurs when the radio device is in RX mode [1] and there is some radio communication received at the mote. Typically interrupts are propagated up as events and there will be appropriate event handlers to process these events. These event handlers typically post tasks to a task queue, which is a FIFO queue, and the TinyOS scheduler schedules these tasks on a FIFO basis [1]. There is a limit on the number of tasks which can be queued which is determined by the queue size. Currently, the number of tasks that can be queued is set to 7.

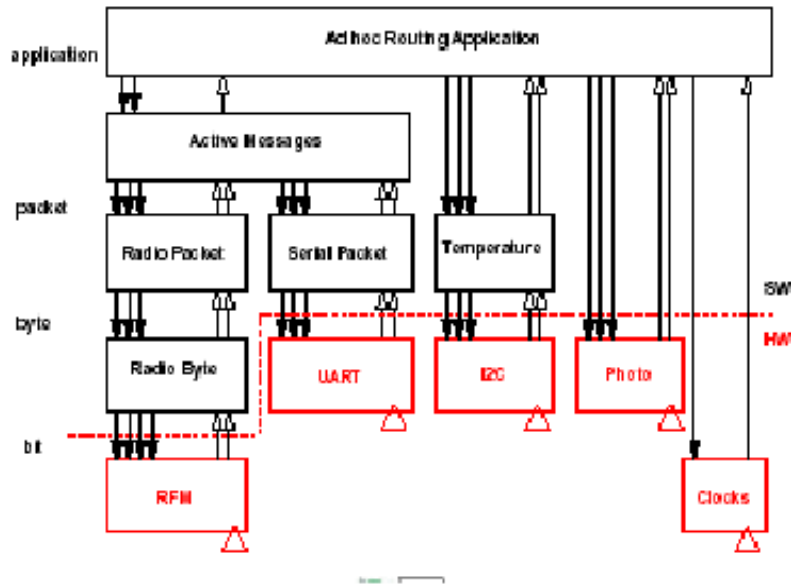


Figure 1: Radio Stack in TinyOS

The architecture of a Berkeley mote requires the CPU to be involved in processing every interrupt. The implication of this is that for every radio bit that is received, the CPU needs to be involved in processing that bit, since there is no network interface processor which will handle this job without involving the CPU. The radio bits are converted to bytes, the bytes converted to packets and packets converted to messages by the Active Messaging layer [2]. All the above steps would involve the CPU. Since interrupts are given higher priority, any task under execution would be preempted by the CPU and the interrupt handler is executed by the CPU.

2.2 Overload in TinyOS

When the arrival rate of the interrupts is very high, at some point the CPU is not able to execute any tasks other than the interrupt handlers. This situation is variously referred to as receiver livelock, overload etc. The rate at which the system is able to complete its tasks is less than the rate of incoming tasks so that there is a point when the queue of incoming tasks keeps increasing till no more requests/tasks can be processed by the system. The rate of sensing/sampling of data, being a localized task, can be controlled. However, incoming traffic for data routing is something that could lead to the above mentioned overload problem. The CPU does process packets,

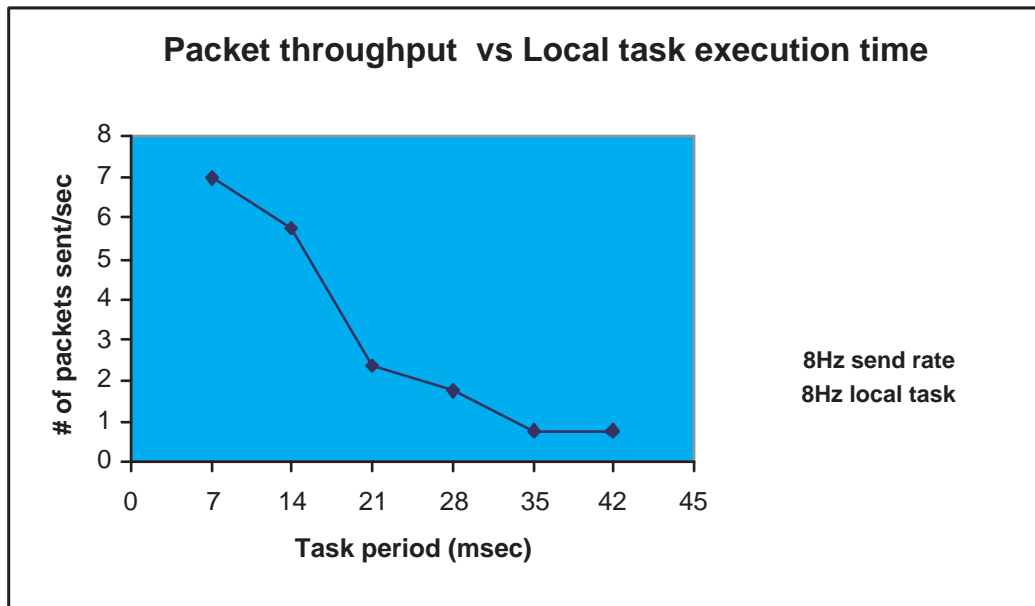


Figure 2: Packet throughput reduces in the presence of local tasks

but it is not able to process tasks which have been queued. It is not able to forward any packets.

Figure 2 shows how packet throughput is reduced in the presence of a 8Hz local task with varying execution times. When the execution time of the local task increases, the radio packet throughput suffers.

2.3 Packet Processing in TinyOS

The reason for the behaviour shown in Figure 2 could be attributed to the manner in which packets are sent and received in TinyOS. Figure 3 shows the sequence of events that take place when a packet is sent over the wireless medium and Figure 4 shows the sequence of events that take place when a packet is received over the wireless medium. Please note that we have omitted a lot of intricate details and have presented only the bare minimum interactions in order to retain the clarity of the discussion.

As shown in Figure 3, the *MicaHighSpeedRadio* component forms the central area of activity coordinating the activities of the other layers like *SpiByteFifo* and *SecDedEncoding*. When a *send* call is made on a communication component at the application layer, it will eventually end up as a request to the *MicaHighSpeedRadio*

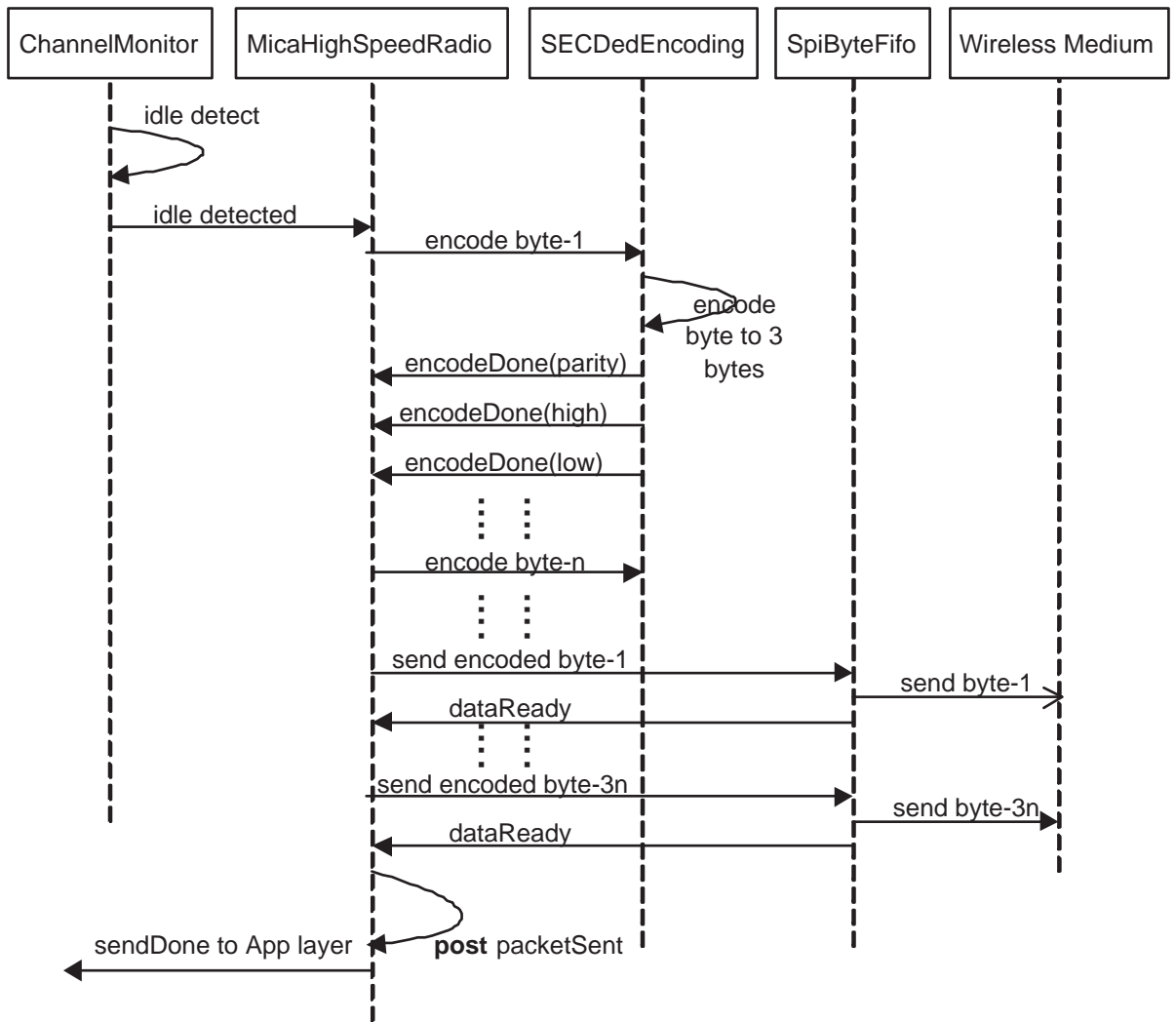


Figure 3: Sequence of Interactions when a packet is sent

component. This component waits for the channel to become idle which would be indicated by an event from the *ChannelMonitor* component. Each byte is encoded into three bytes before sending it over the wireless medium. Similarly, three bytes are decoded into a single byte on reception. After a packet has been sent, an internal task is posted which will signal the application layer that the packet has been sent. While all this processing is taking place in the lower layers, the application layer would be waiting on a notification from the lower layer that the current packet has been sent. Typically this would be done using a boolean variable which will be set to *FALSE* before the *send* call is made on the lower layer and setting the flag to *TRUE* when the *sendDone* notification is received from the lower layer.

Since the application layer is waiting on a notification event which gets triggered from a task posted by *MicaHighSpeedRadio*, the application won't be able to send another packet until the task gets executed. If there are tasks being posted as part of local processing, the *sendDone* notification task will be queued at the end of the task queue and becomes eligible for execution only when all the other tasks before it in the queue has been executed. The send rate for the packets will get affected and this is the reason for the drop in packet throughput shown in Figure 2. Moreover, if the queue is already full, the *sendDone* notification task could get dropped and the application won't be able to send another packet. This is a overload condition in which a relatively more important task is dropped to accommodate a less important task.

Packet reception is very similar to sending a packet. The sequence of events for reception of a packet is shown in Figure 4. When the preamble symbols are detected by the channel, it sends an event notification to *MicaHighSpeedRadio*, which in turn, requests the *SpiByteFifo* component to start reading the bytes from the medium. While sending, one byte was encoded into three bytes and hence during reception, three bytes are received and buffered and then decoded into a single byte. This continues until the whole packet is received, when a *packetRecvd* task is posted by *MicaHighSpeedRadio* to itself. This task, when executed, will signal a *receive* event to the application layer. Until this task gets processed, there won't be any more attempt for the channel to start detecting preamble symbols, i.e. no more packets could be received from the medium. If there are tasks being posted by local processing, there could be a drop in the number of packets received within an specific interval of time. Furthermore, if the posting of the *packetRecvd* fails because the task queue is full with local computations, the *packetRecvd* will be dropped and the application would never be notified of the reception of a packet. Moreover, *ChannelMonitor* will never listen to the channel again, i.e. no more packets can be received.

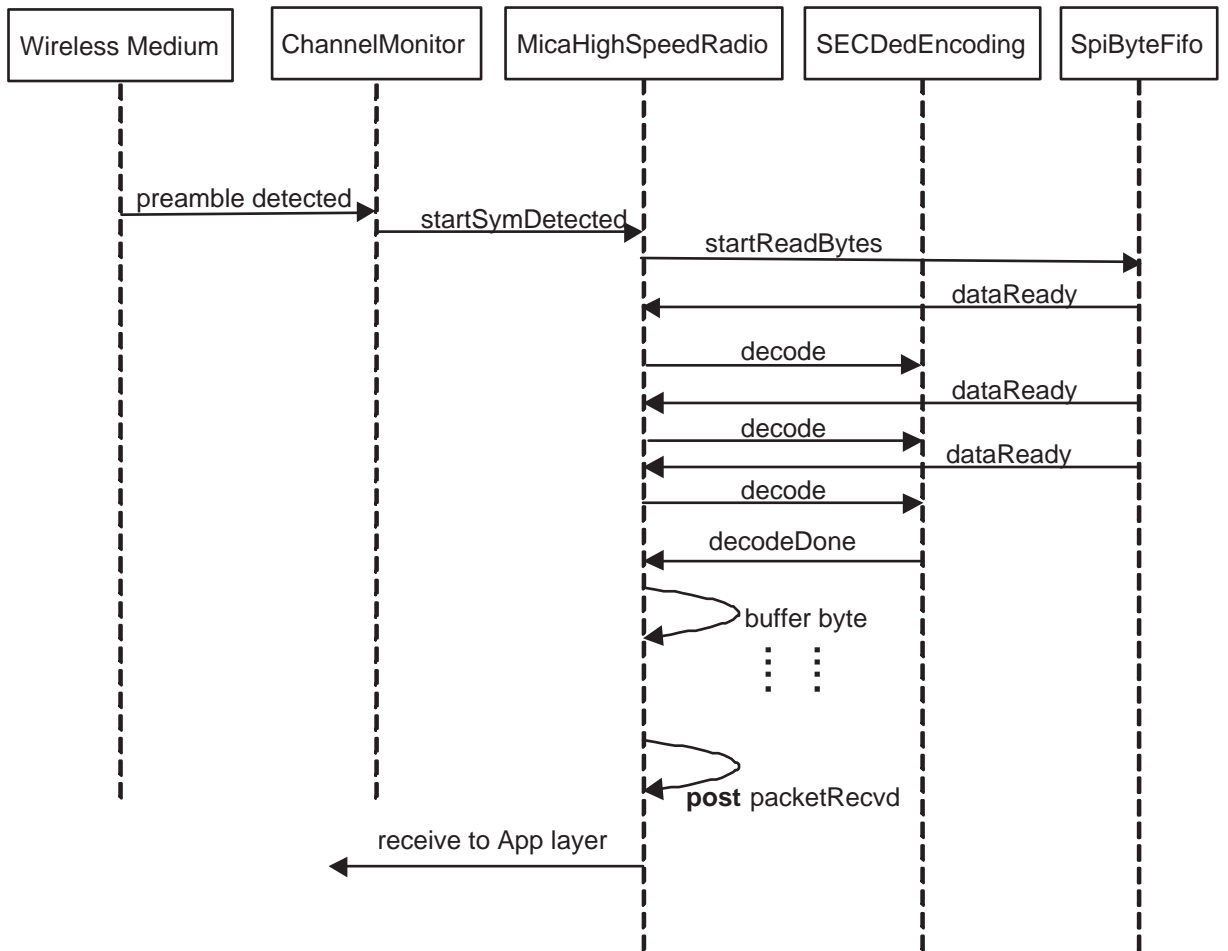


Figure 4: Sequence of Interactions when a packet is received

3 Priority Scheduling in TinyOS

As observed in the previous section, if there are enough local processing tasks, the received packet processing is likely to take longer time which leads to reduced throughput. Such scenarios would be particularly seen in a topology where data is being aggregated, processed and then sent to the base station. We noticed that since the packet was being put at the end of the queue, till the other tasks are completed, the *receivePacket* task would not be executed and till then no other packet can be received. But if the task for packet receive/send is posted at the beginning/head of the queue instead of appending it at the end of the queue, this task would be the next task to be executed leading to a likelihood of better throughput.

3.1 Prioritized Tasks

The existing programming model implicitly supports a FIFO model for posting of tasks. This FIFO model is implicitly enforced by the scheduler in TinyOS which maintains a FIFO queue of posted tasks. The task at the head of the queue will be selected for execution by the TinyOS scheduler. In nesC, a task is posted as follows.

```
task void function_call
{
    ....
}

post function_call ();
```

We propose a new programming model where tasks could be posted with a priority. If unspecified, the priority of a task is 1 by default. For example,

```
task void function_call
{
    ....
}

post function_call () 3;
```

In order to enforce this priority scheme, we modified the TinyOS scheduler to use a priority queue instead of a FIFO queue. The scheduler inserts a new task with a higher priority ahead of already existing tasks with lower priorities. This will eventually result in more important tasks being selected for execution before less important tasks. Moreover, if the task queue is full and the incoming task is of higher

priority compared to the task at the tail of the queue, the task at the tail of the queue will be dropped. This changes the semantics of `post` with regard to the guaranteed execution of a task provided by the original scheduler once a task has been posted. With our enhancement, there is a probability that a lower priority task could get dropped from a full task queue to make room for a incoming higher priority task. Details of how the nesC compiler and TinyOS components were changed, to achieve this, are explained in Appendix A.

3.2 Increased Radio Throughput

As described in Section 2.2, some use-cases require the radio packet processing to have a higher priority than local computations. The priority for the `packetSent` and `packetReceived` explained in Section 2.3 are increased when posting them. These tasks will be enqueued at the head of the queue and gets processed as the next task. Once the task gets executed the application would be able to send the next packet or receive another packet which was being hindered before by the presence of local computation.

When a task is to be inserted, the scheduler starts at the end of the queue and keeps moving till it comes across a task with a priority equal to it. It shifts all the tasks from that point (excluding the one with the same priority) toward the end of the task queue and inserts the current task. Thus the task queue is always sorted in the order of increasing priority.

4 Experimental Evaluation

Figure 5 shows the experimental setup that we tried to use to evaluate the our proposed solution, since this could be a real application. There is a sender which sends a monotonically increasing counter value at 8Hz. There is a router which receives these packets and forwards it to a base station receiver. The router also runs a local task, which encrypts packets. We vary the rate of the encryption task to see the effect on the number of packets forwarded by the router. We could not get accurate results using this setup since the base station would receive the packets sent by the sender also, when it should be only receiving the packets which are forwarded by the router. We tried adjusting the distance between the motes in such a way that the router is within radio range from the sender and the base station but the base station is not within the radio range of the sender. But this setup was very error-prone and very sensitive to the environment and there were lot of variations in the readings that we were getting.

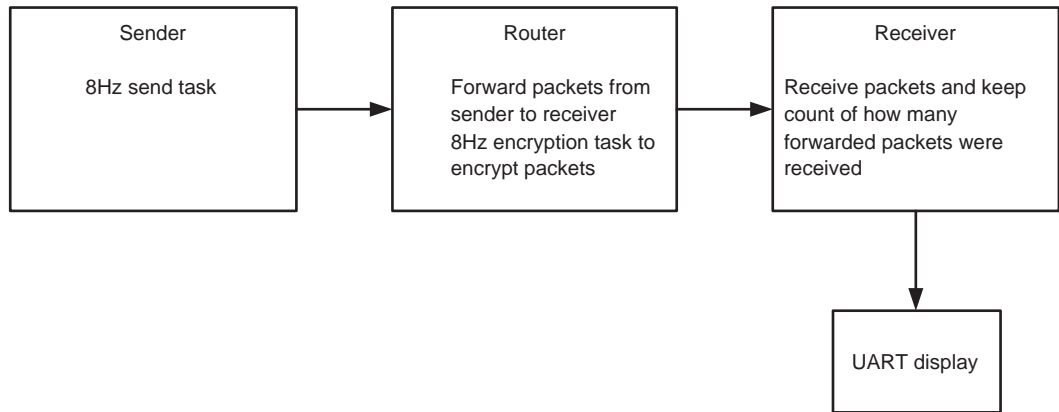


Figure 5: Experimental Setup

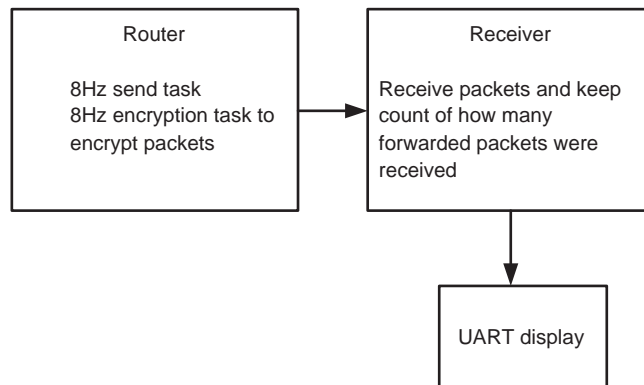


Figure 6: Simplified Experimental Setup

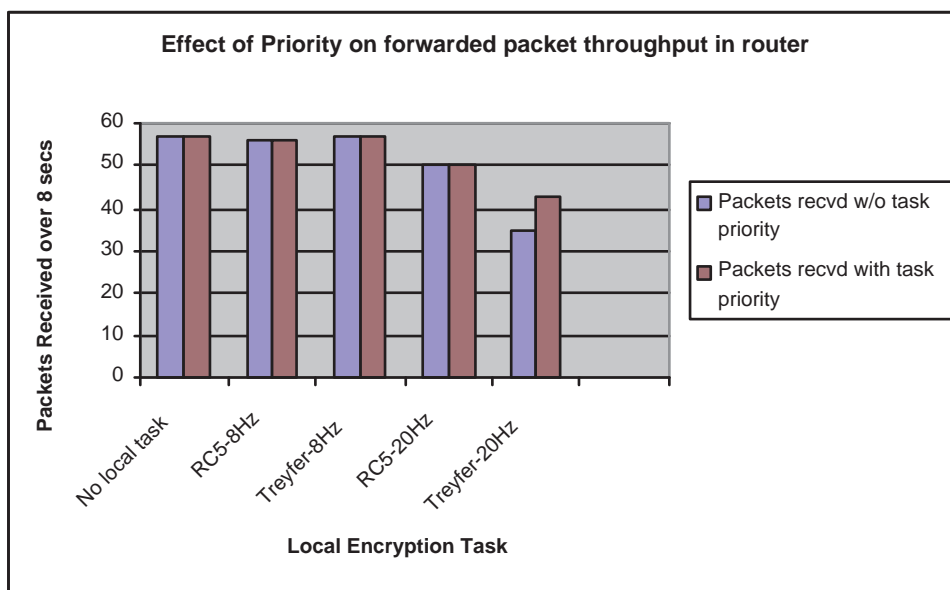


Figure 7: Effect of priority on send rate in presence of local task

In order to circumvent this problem, we merged the router and the sender functionality into the router and performed our measurements from the resulting setup shown in Figure 6.

We used two different encryption algorithms - RC5 [3] and Treyfer [4]. Under our setup, the RC5 encryption takes about 1 ms and the Treyfer algorithm takes about 6 ms execution time. Our results are shown in Figure 7. If there are no local tasks, the number of packets forwarded is about 8 per second. For RC5 it does not make much impact, even with the addition of priority. But for Treyfer, we do see some improvement with the addition of priority.

4.1 Effect of Task Execution Time

This section describes experiments that we performed to find the effect of task execution time on the packet throughput.

4.1.1 Increased Packet Send Throughput

In this experimental setup, there is a sender and a receiver. The sender sends packets at 8Hz (125ms). It also has a local periodic task which also runs at 8Hz. We use



Figure 8: Experimental Setup for Packet Send Throughput

different execution times for the local task to find out the effect on the throughput. Figure 8 illustrates the experimental setup and Figure 9 shows the results.

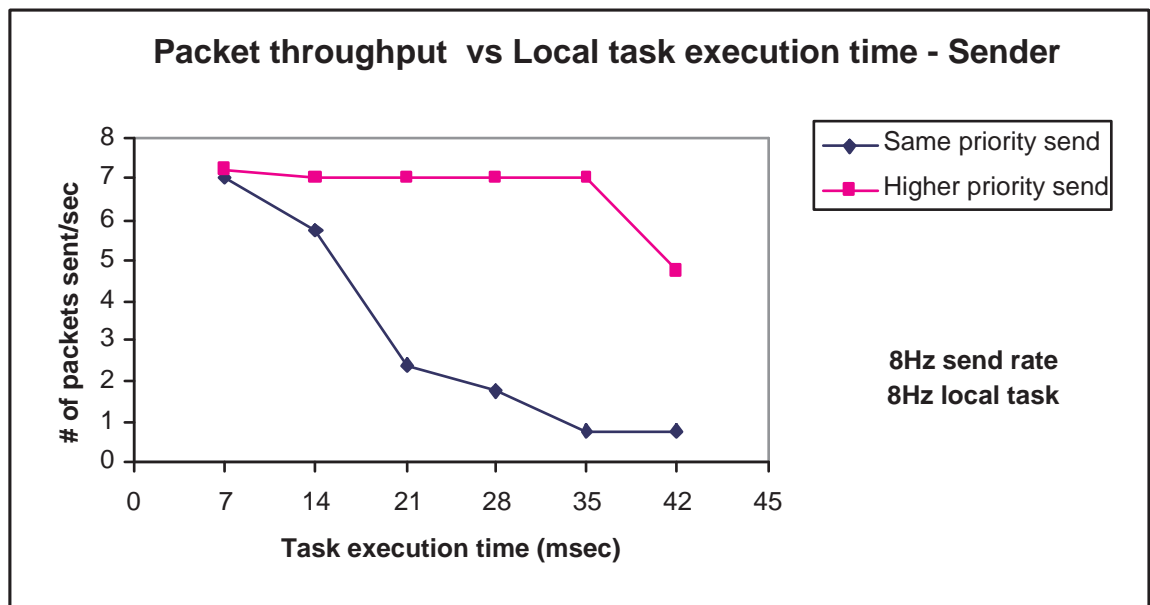


Figure 9: Increased Packet Send Throughput

As seen in the figure, the addition of task priorities improves the throughput by a substantial amount.

4.1.2 Increased Packet Receive Throughput

In this experimental setup, there is a sender and a receiver. The sender sends packets at 8Hz (125ms). The receiver, apart from receiving packets, also has a local periodic

task which runs at 8Hz. We use different execution times for the local task to find out the effect on the throughput. Figure 10 illustrates the experimental setup and Figure 11 shows the results.



Figure 10: Experimental Setup for Packet Receive Throughput

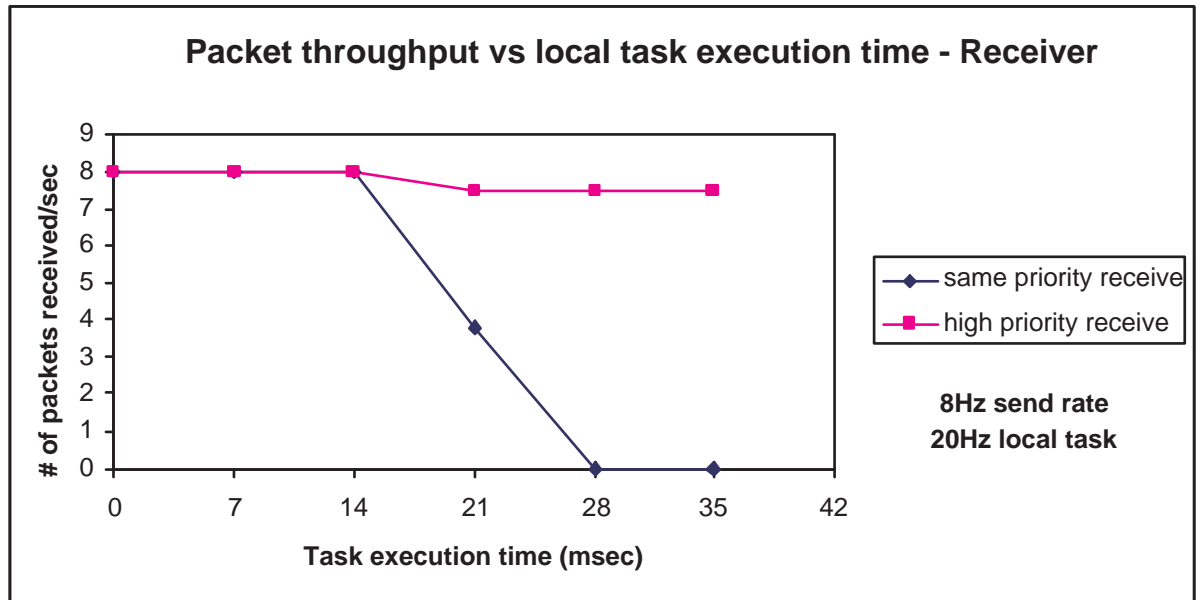


Figure 11: Increased Packet Receive Throughput

As seen in the figure, the addition of task priorities improves the throughput by a substantial amount. In the case with no priority, note that the packets received drops to zero when the task execution time reaches 28 msec. We attribute the reason to an overload situation which happens at this point. From our observations during the experiment, we found that the posting of the *packetReceived* task, as explained in Section 2.3, was failing. Due to this, the channel monitor is never made to listen for

start symbols by *MicaHighSpeedRadio*. This results in no more packets being received. With our improvements, even if the task queue is filled with the local computationally intensive tasks, the *packetReceived* task is given priority and the task at the tail of the queue is removed making room for the *packetReceived* task. This results in the lower layers continuing to receive packets. It should also be noticed that the throughput goes down at a much faster rate in the previous experiment because two tasks are getting posted there - a *sendPacket* task to send a packet and a *sendDone* task to notify that a packet has been sent. In this experiment there is only one task which is the *packetReceived* notification task.

5 Conclusion

TinyOS scheduler uses a FIFO queue for scheduling tasks. All tasks are treated with equal importance. We investigated into certain use-cases where this could pose a problem resulting in situations where packet processing completely stops on a sensor node. We propose a mechanism by which we could specify a priority associated with a task. We also enforce this priority scheme by enhancing the TinyOS scheduler with a priority queue instead of a FIFO queue. From the experimental results, we see an improvement in the packet processing throughput. We also see better behavior when the system gets overloaded.

6 Acknowledgements

We would like to thank Guoliang Xing for helping us with the code for the Treyfer and RC5 encryption algorithms.

A TinyOS Code Enhancements

In order to post a task with a priority, we had to modify the nesC grammar and the parser. In the file *c-parse.y*, a *task_call* token has been added for taking the extra priority parameter as follows,

```
callkind:
    CALL { $$i = command_call; }
    | SIGNAL { $$i = event_signal; }
/*    | POST { $$i = post_task; } */
    ;
```

```

task_call:
    function_call
| function_call CONSTANT
    {
        function_call fc = CAST(function_call, $1);
        fc->args = CAST(expression,$2);
        $$ = $1;
    }

unary_expr:
    primary
| callkind function_call
    {
        function_call fc = CAST(function_call, $2);
        type calltype = fc->arg1->type;

        $$ = $2;
        CAST(function_call, $$)->call_kind = $1.i;
        switch ($1.i)
        {
            case command_call:
                if (!type_command(calltype))
                    error("only commands can be called");
                break;
            case event_signal:
                if (!type_event(calltype))
                    error("only events can be signaled");
                break;
            /* case post_task:
                if (!type_task(calltype))
                    error("only tasks can be posted");
                fc->type = unsigned_char_type;
                break; */
        }
    }
| POST task_call
    {
        function_call fc = CAST(function_call, $2);
        type calltype = fc->arg1->type;
        fc->call_kind = post_task;
    }

```

```

        $$ = $2;
        if (!type_task(calltype))
            error("only tasks can be posted");
        fc->type = unsigned_char_type;
    }

```

The *unparse.c* file is modified to output the extra priority parameter for all the tasks to the generated C code.

```

void prt_function_call(function_call e, int context_priority)
{
    switch (e->call_kind)
    {
        case post_task:
            set_location(e->arg1->location);
            output("TOS_prio_post(");
            prt_expression(e->arg1, P_ASSIGN);
            output(",");

            if (e->args != 0)
                prt_expression(e->args, P_ASSIGN);
            else
                output("1");

            output(")");
            break;
        ...
    }
}

```

Figures 3 and 4 respectively show that a task is posted to indicate that a packet has been sent or received. The priorities for these are incremented in *MicaHighSpeedRadioM.nc*. In *AMStandard.nc*, a send call is implemented in terms of posting a task to do the actual send. The priority of this call is incremented.

The scheduler in the *sched.c* has been modified so that it inserts the new task with higher priority ahead of the already existing tasks with a lower priority. When a task is to be inserted, the scheduler starts at the end of the queue and keeps moving till it comes across a task with a priority equal to it. It shifts all the tasks from that point (excluding the one with the same priority) toward the end of the task queue and inserts the current task. Thus the task queue is always sorted in the order of increasing priority.

Following show the code snippets that required modifications to the scheduler source code.

```

bool TOSH_prio_post(void (*tp) (), int prio) __attribute__((spontaneous)) {
    //TOSH_queue[tmp].prio = prio;
    //TOSH_queue[tmp].tp = tp;
    insert_task(tp, prio);
}

void insert_task(void (*tp) (), int prio)
{
    int i, nexti;
    for(i=prev(TOSH_sched_free);i!=prev(TOSH_sched_full);i=prev(i))
    {
        nexti = next(i);
        if (TOSH_queue[i].prio < prio)
        {
            TOSH_queue[nexti] = TOSH_queue[i];
        }
        else
        {
            break;
        }
    }
    TOSH_queue[next(i)].tp = tp;
    TOSH_queue[next(i)].prio = prio;
}

```

References

- [1] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.
- [2] Philip Buonadonna, Jason Hill, and David Culler. Active message communication for tiny networked sensors. 2001.
- [3] R. L. Rivest. The rc5 encryption algorithm. In *Proceedings of the 1994 Leuven Workshop on Fast Software Encryption*, pages 86–96, 1995.

- [4] G. Yuval. Reinventing the travois: Encryption/mac in 30 rom bytes. *Software Encryption, 4th International Workshop Proceedings*, pages 205–209, 1997.