CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Lancaster E-Prints

An Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems

Cormac Duffy, John Herbert, Cormac Sreenan
Computer Science Department
University College Cork, Ireland
{c.duffy|j.herbert|c.sreenan}@cs.ucc.ie

Utz Roedig
InfoLab21
Lancaster University, UK
u.roedig@lancaster.ac.uk

Abstract

Two different operating system types are currently considered for sensor networks: event driven and multi-threaded. This paper compares the two well -known operating systems TinyOS (event driven) and MANTIS (multi-threaded) regarding their memory usage, power consumption and processing capabilities. TinyOS and MANTIS are both ported to the DSYS25 sensor platform. Both operating systems are used to execute the same sensor network application and the aforementioned parameters of interest are measured. The results presented in this paper show for which set of applications each operating system is preferable.

1. Introduction

Currently, operating systems for sensor nodes follow either one of two different design concepts, *event-driven* and *multi-threaded*. In event-driven systems every action an operating system has to perform is triggered by an event (e.g. a timer, an interrupt indicating new sensor readings or an incoming radio packet). The tasks associated with each event are processed sequentially until the operating system is idle and can be sent into an energy-efficient sleep state. As events are processed in order, expensive context switching between tasks is not necessary. An example of such an operating system is TinyOS [1]. The second approach follows the *multi-threaded* operating system concept. The operating system multiplexes execution time between the different tasks, implemented as threads. While switching from one thread to another, the current context has to be saved and the new context must be restored. This consumes costly resources in the constrained sensor node. An example of such an operating system for sensor nodes is MANTIS [2].

It is generally assumed that an event-driven operating system is very suitable for sensor networks because few resources are needed, resulting in an energy-efficient system [1]. However, the exact figures are unknown and therefore this paper quantifies precisely the resource usage. It is also claimed that a multi-threaded operating system has comparatively better event processing capabilities in terms of meeting processing deadlines [2]. Again, an in-depth analysis is currently missing and is therefore conducted. For the comparisons, the event-based system *TinyOS* and the multi-threaded system *MANTIS both* execute the same sensor network applications on the *DSYS25* [3] sensor platform. Memory requirements, energy consumption patterns and the event processing capabilities of the two operating systems are investigated in this paper. The results presented can be used to decide which type of operating system should be used for a specific sensor network application. The results also show that for a number of application areas a thread-based sensor network operating system is actually feasible and even preferable.

Historically, there has been much debate on whether an event-based or multi-threaded architecture is more efficient. However, none of these discussions consider the sensor network domain which dictates very specific constraints. Existing work targets only a subset of aspects investigated in this paper. For example papers analyzing or describing one specific operating system (e.g. [1, 2, 4]), or publications comparing only one aspect (e.g. memory usage in [5]). As each single existing analysis is based on different assumptions and experimental setups, it is not possible to extract an objective comparison. For an objective comparison of the operating systems, a complete study presented as in this paper, is required. Due to space restrictions, existing related work is not discussed in more detail.

The rest of the paper is organized as follows. Section 2 gives an overview of the operating systems TinyOS and MANTIS. Section 3 describes the test application implement on our sensor nodes, for the comparative study. Section 5 presents the experimental comparison of the operating systems. Section 7 concludes the paper.

Algorithm 1 TinyOS structure

```
 1: component_A
 2:   task do(){...}
 3:   command X(){...}
 4:   event Y(){...}

 5: int_A
 6:   ...
 7:   post_task(A)

 8: TOSH_run_task()
 9:   while(TOSH_run_next_task())
10:   TOSH_sleep()
```

Algorithm 2 MANTIS structure

```
 1: thread_A
 2:   while(running)
 3:      ...;mos_semaphore_wait(A1);...

 4: int_A
 5:   ...;mos_semaphore_post(A1);...

 6:dispatch_thread()
 7:   PUSH_THREAD_STACK()
 8:   CURRENT_THREAD = readyQ.getThread()
 9:   CURRENT_THREAD.state=RUNNING
10:   POP_THREAD_STACK()
```

2. Sensor Node Operating Systems

In order to compare the event driven and multi-threaded operating system concepts, a well known and widely used implementation of each is selected, namely *TinyOS* and *MANTIS*.

TinyOS The operating system and specialized applications are written in the programming language nesC and are organized in self-contained components. A simplified view of this component structure is shown in Alg. 1. Components consist of interfaces in the form of *command* and *event* functions. Components are assembled together, connecting interfaces used by components to interfaces provided by others, forming a customized sensor application. The resulting component architecture facilitates event-based processing by implementing event-handlers and TinyOS tasks. TinyOS tasks are deferred function calls and are placed in a simple FIFO task-queue for execution (see Alg. 1, line 8). TinyOS tasks are taken sequentially from the queue and are run to completion. Once running, the TinyOS task can not be interrupted (preempted) by another TinyOS task. Event-handlers are triggered in response to a hardware interrupt and are able to preempt the execution of a currently running TinyOS task (see Alg. 1, line 5). Event-handlers perform the minimum amount of processing to service the event. Further non time-critical processing is performed within a TinyOS task that is created by the event handler. After all TinyOS tasks in the task queue are executed, the TinyOS system enters a sleep state to conserve energy (see Alg. 1, line 10). The sleep state is terminated if an interrupt occurs.

MANTIS Each task the operating system must support can be implemented - using standard C - as a separate MANTIS thread. A simplified view of this thread structure is shown in Alg. 2. A new thread is initialized and thread processing is started (line 1). Processing might be halted using the function *mos_semaphore_wait* when a thread has to wait for a resource to become available (line 3). An interrupt handler (line 4) using the function *mos_semaphore_post* (line 5) is used to signal the waiting thread that the resource is now available and thread processing is resumed. While a thread is waiting on a resource to become available, other threads might be activated or, if no other processing is required, a power saving mode is entered. Power saving is handled by a thread called idle-task which is scheduled when no other threads are active. Thread scheduling is performed within the kernel function *dispatch_thread* shown in Alg. 2, line 6. This function searches a data structure called *readyQ* for the highest prioritized thread and activates it. When the *dispatch_thread* function is called, the current active thread is suspended calling *PUSH_THREAD_STACK* (line 7) which saves CPU register information. The highest priority thread is then selected from the *readyQ* (line 8) and its register values are restored by the *POP_THREAD_STACK* function (line 10). Before the *dispatch_thread* function is called, the *readyQ* structure is updated. Threads that are currently sleeping or that are waiting on a semaphore (resource) are excluded from the *readyQ*. The scheduling through the *dispatch_thread* function can be initiated by two different means. *Dispatch_thread*is called when a semaphore operation is called (e.g. to let the current thread wait on a resource). *Dispatch_thread*is also called periodically by a time slice timer to ensure processing of all threads according to their priority.

3. Evaluation Setup

For the evaluation, TinyOS and MANTIS are ported to the DSYS25 [3] platform and measurement facilities are integrated in both operating systems (see [6], for more detailed information on the evaluation setup) . To actually perform the comparative evaluation, an abstract application scenario is defined. Depending on a sensor node's role within this scenario (leaf node vs. forwarding node) and the configuration of the scenario itself (high sensing task vs. small sensing task), a node is stressed differently. The performance of a single node, exposed to the different stress situations is measured while using the two different operating systems. In the following paragraphs, the abstract application scenario is motivated and described.

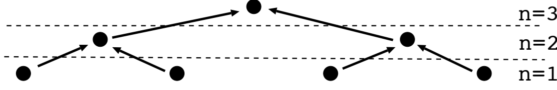


Figure 1. Binary Tree

OS	Program Size (KB)	Required RAM (B)
TinyOS	9	283
MANTIS	13.1	287

Table 1. Memory Usage

Application Scenario In many cases, a sensor network is used to collect periodically obtained measurement data at a central point (sink or base-station) for further analysis. The sensor nodes in such a network execute two major tasks. First a sensor nodes perform a sensing operation and second the node must forward the gathered data hop-by-hop to the sink. The execution time of the sensing task will depend on the nature of the physical phenomenon monitored and the complexity of the algorithm used to analyze it. Therefore, the position of the node in such a network and the complexity of the sensing task define the operating system load of the sensor node.

The complexity of the sensing operation depends on the phenomenon monitored, the sensor device used and the data preprocessing required. As a result, the operating system can be stressed very differently. If, for example, an ATMEGA128 CPU with a processing speed of $4Mhz$ is considered, a simple temperature sensing task processed through the Analog to Digital Converter can be performed in less than a millisecond. If the same device is used in conjunction with a camera, image processing might take up to $100ms$ [7] before a decision is made. Note that a long sensing task can be split-up into several sub-tasks but in practice this is not always possible[7].

The complexity of a packet forwarding operation depends on the transceiver type, the MAC-layer and routing protocols used. On the DSYS25 platform with a Nordic transceiver approximately 4000 clock cycles are necessary to read a packet from the transceiver, perform routing and re-send the packet over the transceiver. The amount of packet forwarding tasks depends obviously on the node under consideration and the current network topology.

Topology It is assumed that a binary tree topology is formed in the network (see Fig. 1). Depending on the position n in the tree, a sensor node might process varying amounts of packets. In the experiments, the behavior of a single node at all possible positions n is emulated and measured by applying the sensing pattern and network traffic as described next.

Sensing Pattern A homogeneous activity in the sensor field is assumed for the abstract application scenario. Each sensor gathers data with a fixed frequency f_s. Thus, every $t_s = 1/f_s$ a sensing task of the duration l_s has to be processed. The duration l_s is variable between $l_s = 4000$ and $l_s = 400000$ clock cycles depending on the type of sensing task under consideration (Which corresponds to $1ms$/$100ms$ on a $4MHz$ CPU).

Traffic Pattern Depending on the position n of a node in the tree, varying amounts of forwarding tasks have to be performed. It is assumed that no time synchronization among the sensors in the network exists. Thus, even if each sensor produces data with a fixed frequency, data forwarding tasks are not created at fixed points in time. The arrival rate λ_n of packets at a node at tree-level n is modeled as a Poisson process. As the packet forwarding activity is related to the sensing activity in the field, λ_n is given by: $\lambda_n = (2^n - 1) \cdot f_s$. It is assumed that the duration (complexity) l_p of the packet-processing task, is $l_p = 4000$ clock cycles.

4. Memory Usage

The memory footprint of the operating system has to be as small as possible. The more complex an application is (with respect to memory requirements), the more likely a more capable Memory/CPU chip will be required to host the application.

In order to determine the memory usage of each operating system, we use the GNU project binary utility avr-size. Avr-size is a flash image reader that outputs the program size and static memory (global variables) required by each operating system. However, the compilation procedure for both operating systems is somewhat different. A specialized custom compiler (nesC) provided with the TinyOS framework, exploits the component-based architecture to include only components required by the application's wiring schema in the compiled program image. Furthermore, the nesC compiler can deduce and remove any unused component functions within the application. Thus, to provide a fair memory comparison of both operating systems, the MANTIS application and operating system is stripped before compiling of all functionality that is not used for the abstract application .

Results The results in Table 1 show that the MANTIS operating system takes 30% extra programmable memory space compared to the TinyOS operating system. It has to be noted that both operating systems require additional flash memory to cater for the stack which is not shown in Table 1.

Furthermore, the MANTIS scheduler dynamically allocates a memory pool to store the stack and processor registers for each thread.

The results show that both operating systems have very similar memory requirements. Thus, conventional microprocessors combining CPU and memory can normally hold either of the investigated operating systems.

5. Event Processing

It is assumed that the packet-processing task within the nodes has priority so that deadlines regarding packet forwarding can be met. Thus, in the MANTIS implementation, the packet-processing task has a higher priority than the sensing task. In the TinyOS implementation, no prioritization is implemented as this feature is not provided by the operating system.

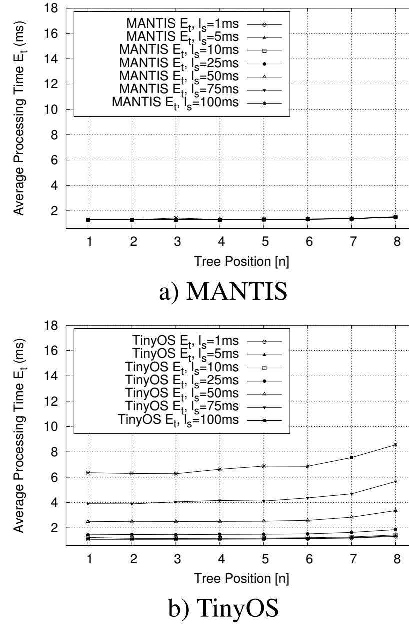
To characterize processing performance of the operating system, the average task execution time E_t of the packet forwarding task, is measured. During the experiment, J packet-processing times e_j are recorded. To do so, the task start time e_{start} and the task completion time e_{stop} are measured and the packet-processing time is recorded as $e = e_{stop} - e_{start}$. The average task execution time E_t is calculated at the end of the experiment as: $E_t = \sum e_j / J$. For each tree position n, the experiment is run until $J = 25000$ packet-processing events are recorded.

Results In the experiment, the average task execution time E_t is determined for TinyOS and MANTIS supporting the abstract application scenario (see Fig. 2).

Where MANTIS is used, it can be observed that the average packet-processing time is independent of the sensing task execution time. Furthermore, E_t is also independent from the position n of the node in the tree. The average processing time increases slightly, under a heavy load. This is due to the fact that under heavy load packet forwarding tasks have to be queued (see Fig. 2 a)).

Where TinyOS is used, the average processing time for the packet forwarding task E_t depends on the length of the sensing l_s of the sensing task. In addition, under heavy load the queuing effects of the packet forwarding tasks also contribute somewhat to the average processing time (see Fig. 2 b)).

The variance in the packet-processing time E_t is also recorded but not shown due to space restrictions. It has to be noted that this variance is significantly smaller in MANTIS than in TinyOS (e.g. with $n = 8$ and $l_s = 75ms$, there is a $8.3ms$ variation of packet processing time in TinyOS compared with a $0.4ms$ variation in MANTIS). Thus, MANTIS is better able to support scenarios which require predictable processing behavior.



a) MANTIS

b) TinyOS

Figure 2. Average packet-processing time E_t

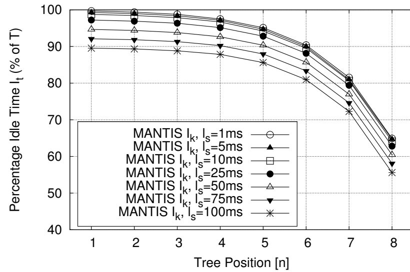
The thread prioritization capability of MANTIS is clearly visible in the experimental results. Packet processing times are independent of the concurrently executed and lower priority sensing task. In TinyOS, sensing and packet forwarding task delays are coupled, and the influence of the sensing activity on the packet forwarding activity is clearly visible.

6. Energy Consumption

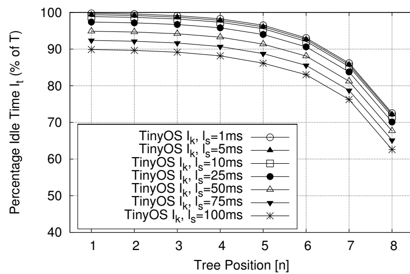
To evaluate power-efficiency, This study investigates the available idle time in which low-power operations can be scheduled. Thus the comparative effectiveness of specific power management policies can be gauged on the amount of potential low-power (idle) time available.

In the experiment, the abstract application scenario is executed by the sensor node running TinyOS or MANTIS. The duration of the experiment T and the duration i_k of K idle time periods during the experiment is recorded. i is defined as $i = i_{stop} - i_{start}$. All idle periods i_k are summarized and the percentage idle time, I_t, the percent of experiment time, in which the processor is idle, which is calculated as follows: $I_t = (\sum i_k / T) \cdot 100$. Again, for each tree position n, the experiment is run until $J = 25000$ packet-processing events are recorded.

Results In the first experiment, the percentage idle time I_t is determined for TinyOS and MANTIS supporting the abstract application scenario. (see Fig. 3).



a) MANTIS



b) TinyOS

Figure 3. Percentage idle time I_t

The time spent in idle mode drops exponentially for both operating systems with the increasing node position in the tree described by the parameter n. This behavior is expected as the number of packet tasks increases accordingly. Less obvious is the fact that the available idle time drops faster in MANTIS than in TinyOS. The fast drop in idle time is caused by the context switches in the MANTIS operating system. The more packet forwarding tasks are created, the more likely it is that a sensing task is currently running when a packet interrupt occurs. Subsequently, a context switch to the higher prioritized forwarding task is needed.

It is clearly visible that TinyOS is more energy efficient than MANTIS; especially under a high system load.

7. Conclusion

Both operating systems fit on standard microprocessors combining CPU and memory. However MANTIS uses 30% more space, but both systems are well within reasonable bounds for today's microprocessors. The experimental results show that MANTIS is more predictable than TinyOS. Specifically, the packet forwarding task execution time in MANTIS has a low variation and is independent of other activity such as the sensing task. Thus, MANTIS would be preferable in situations that need deterministic and timely processing. However, as the experiments show, the MANTIS system is not as power-efficient as TinyOS. Thus, TinyOS would seem preferable if energy consumption is deemed to be of primary importance.

In general, the experiments confirm what one would expect. However, an interesting and not obvious fact is highlighted by the experiments. If the system is not loaded (leaf node with $n = 1$ and a sensing task with the size of $l_s = 1ms$) a difference of only 0.1% in idle time is measured (compared to a difference of 6.9% under heavy load with $n = 8$ and $l_s = 100ms$). Thus, MANTIS would be a good choice in cases where the sensor network is idle for long periods and suddenly high activity is encountered that requires timely processing of sensor information. For these kinds of applications, MANTIS combines both important sensor network design goals, i.e. energy efficiency and predictive behavior.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *ACM SIGOPS Operating Systems Review*, vol. 34, pp. 93–104, December 2000.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgenson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *ACM kluwer Mobile Networks & Applications Journal, special Issue on Wireless Sensor Networks*, August 2005.

[3] A. Barroso, J. Benson, T. Murphy, U. Roedig, C. Sreenan, J. Barton, S. Bellis, B. O'Flynn, and K. Delaney, "Demo abstract: The DSYS25 sensor platform," in *2nd international conference on Embedded networked sensor systems*, pp. 314–314, November 2004.

[4] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, November 2004.

[5] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *3rd International Conference on Mobile Systems, Applications, and Services*, pp. 117–124, June 2005.

[6] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan, "A performance analysis of TinyOS and MANTIS," Tech. Rep. CS-2006-27-11, University College Cork, Ireland, November 2006.

[7] M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava, "Cyclops: In situ image sensing and interpretation in wireless sensor networks," in *In proc. 3rd international conference on Embedded Networked Sensor Systems,*, pp. 192–204, November 2005.