Achieving Real-Time Operation in TinyOS

Cormac Duffy, John Herbert

Computer Science Dept.
University College Cork, Ireland
{c.duffy|j.herbert}@cs.ucc.ie

Abstract Achieving predictable operation is a complex task in sensor networks as applications intrinsically rely on unstable network links to transmit unpredictable quantities of environmental data. A real-time development infrastructure is needed to provide a greater degree of performance control while still adhering to the development constraints inherent in sensor networks. In this paper we discuss a popular sensor network operating system TinyOS, that has been successful in providing an efficient development environment, but never strived to provide hard real-time operation. The lack of temporal specification and constraints in TinyOS precludes real-time application development. In this paper we propose a real-time model that provides a temporal infrastructure as a foundation for building and analysing real-time applications.

Wireless Sensor Networks, TinyOS, Real-Time.

1 Introduction

Wireless sensor networks are tiny sensor acquisition systems wirelessly tethered to provide cost efficient fine grained monitoring for environments. Often deployed in harsh environments, sensor networks frequently precipitate erratic network communication, as nodes can malfunction, or radio communication links can be disrupted. Forecasting the behaviour of an application is therefore very difficult. Developers can try to determine the response times of a system through extensive testing but this is very costly and does not guarantee timely operation. Thus performance control mechanisms are required in sensor networks to ensure timely execution of processes.

Performance control is an active research topic in sensor networks that endeavours to achieve deterministic operation in each layer of the sensor network architecture. Calculable end-to-end message transmission times rely on a number of real-time network layer technologies such as a real-time MAC layer and a real-time routing protocol. However coordinating such technologies in a predictable application fundamentally requires a real-time operating system, which will be the focus of this paper.

Real-time systems are a common infrastructure within embedded architectures, as such, the term real-time has come to have different meanings in literature. To resolve any semantic ambiguities we provide the following definitions:

- A real-time system is a system in which the correct run-time behaviour depends upon results being delivered within certain temporal constraints.
- The goal of a real-time system is not to provide the fastest possible execution time for all processes, but to provide methodologies for calculating the worst case response time of a process and allow developers to predict the maximum utilization of the system.

In other words developers should be able to predict under what level of stress a real-time operating systems will fail, if failure is possible.

We focus on developing real-time applications for TinyOS[1], a tiny modular operating system designed specifically for sensor network systems. Wireless sensor networks have many unique constraints that are ideally met by the light-weight response mechanisms in TinyOS and should ideally be adopted by a real-time sensor network OS. However many of the TinyOS mechanisms operate unpredictably and are at odds with real-time engineering. The TinyOS component design, for example, provides functional encapsulation at the expense of hiding temporal behaviour. Processes are distributed over a range of component event-handlers, executing asynchronously in response to environment events. Applications can be constructed to execute efficiently and possibly meet real-time requirements, but TinyOS does not provide any support for developers to definitively calculate conditions under which an application will correctly behave.

The contributions of this paper can be summarised as follows,

- We propose to extend TinyOS, with real-time components. By providing more concise temporal specifications to component interfaces, we can facilitate more calculable process timings.
- We also describe how these constraints can be used to calculate the execution times of processes distributed over numerous event handlers, in order to determine task schedulability.

The rest of this paper is organised as follows, in section 2, we provide a more detailed analysis of the TinyOS operating systems and its processing constraints. In section 3, we introduce the idea of implementing real-time components in TinyOS as a basis for constraining processes. In section 4 we expand on this idea and demonstrate how such a system can be used to determine the temporal behaviour of component operations. A brief overview of related work in both real-time component system and TinyOS is detailed in section 5 and finally we conclude in section 6.

2 A Sensor Network Operating System

In realising a real-time sensor network operating system, it is important to consider the concepts and constraints of both architectures before a bipartite solution can be found. In this section we briefly outline the TinyOS process model. We discuss the architectural support for sensor network applications and its negative impact on real-time applications.

2.1 TinyOS

The TinyOS operating system implements an event-based architecture to facilitate sensor network application requirements in an efficient and responsive manner. All IO processes are divided into split-phased operations, which consist of a request operation, e.g. *getData* and a response operation such as *dataReady*[1]. In this way no process has to poll an interrupt or delay execution for any pending processes. Any lengthly operations can be scheduled as a TinyOS task to execute atomically at a later time in order to ensure all pending events are quickly processed.

Sensor network applications require complex concurrent mechanisms that can responsively execute concurrent events. Such development is often complicated by either race-conditions, which can be a common source of annoyance inherent in concurrent development, or by the memory constraints of the target sensor node platform. TinyOS employs a static component design to facilitate race-condition checks and perform dead code elimination, allowing developers to eliminate potential bugs and reduce application memory and code size requirements.

2.2 Real-Time Vulnerabilities

The TinyOS design principles satisfy fundamental sensor network requirements but in turn inhibit traditional methods of performance control. Traditional real-time systems implement a procedure-based architecture, which provides methods for realizing a process as a common entity, a thread[2]. However component based architectures naturally define a concrete boundary between component entities through which neither data nor state are shared. The thread of execution is dispersed among a series of component event handlers, that trigger indeterminately in response to interrupts, obscuring the common thread of execution. While event-based process flows are designed to provide responsive execution, they require an upper bound on the number of times they are triggered, to allow developers to calculate the processor utilization at any point in time.

TinyOS components conceal the temporal behaviour of processes. Components inherently encapsulate functionality and provide coherent interfaces to ensure unambiguous operation. However the component interfaces in TinyOS do not express the temporal requirements or properties that might facilitate predictable execution. For example a message send interface might express a radio transmit operation with a *sendMessage* command and its counterpart event *messageSent*. The interface would adequately describe the component operation but would provide no support for predictable operation. Developers can try to determine the worst case execution times of a system through extensive testing but this is very costly as a single change in component arrangement can drastically change system behaviour making predictable operation impossible.

The TinyOS task scheduler allows developers to have a non-preemptive control over task execution. TinyOS tasks run atomically with respect to each other to avoid race conditions [4], but there can be significant latency in scheduling

high-priority tasks if a lengthly low priority task has already begun execution. This has a significant impact on task schedulability as non-preemptive tasks can only be effectively scheduled if task execution times are known in advance [3]. However, as previously explained, the TinyOS component architecture conceals such information.

3 Real-Time Component Based Software Engineering

In the previous section we highlighted real-time analysis problems associated with component-based architectures such as TinyOS. Determination of process execution times is complicated by the fact that processes are fragmented over a number of different components. In this section we introduce real-time component engineering concepts as a basis for providing a suitable model for predetermining process execution timings.

3.1 The Real-Time Architecture

We propose to extend the TinyOS model to facilitate real-time component configurations. Encapsulating functionality as a real-time component facilitates intuitive control over system operations. Real-time components provide interfaces that express both temporal behaviour and operation control. This enables developers to enforce timing constraints at discrete and important intersections of component functionality.

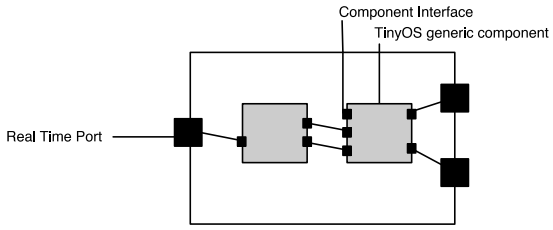

Figure 1. Real-Time Component

Real-time component models facilitate predictable systems, but their design is complicated by the complex temporal relationships components require to meet their collaborative deadlines. This assumption has been thoroughly researched by Wegener & Mueller[3] who claim that the overhead in designing a component is not recovered until its fifth reuse.

To reduce the modeling complexity of a real-time system, our model implements a hybrid approach to developing real-time component functionality. Complex component relationships are decomposed into real-time *black-boxes*, in

which a real-time component encapsulates generic components and bounds their inputs and outputs at the real-time component interface.

Each real-time interface is constrained by a real-time port which validates incoming events based on a minimum period constraint defined in the real-time interface. Any subsequent event that occurs within the minimum period of invocation is rejected by the real-time port. This constraint-based modular engineering style facilitates a more user-friendly real-time model allowing real-time applications to be realized in TinyOS without introducing overly elaborate concepts or multiple changes to the existing TinyOS Architecture (See Figure 1).

4 Defining a Real-Time Interface

In this section we define a hybrid real-time component model that exploits configurative information to provide more transparent task execution times. We start by defining a real-time component as simply a wiring configuration of generic components that are arranged in a way that they can meet the constraints imposed by the real-time specification. Each real-time component will support and possibly require a set of time constrained split-phase operations. Each split-phase interface operation T_i is bounded by e_i, r_i, x_i, in which e_i and r_i are the *partial* worst-case execution and *partial* response times of T_i respectively and x_i is the minimum period of invocation and deadline of T_i. The interface response time includes the time to execute a task and the task delay in waiting for a hardware device to respond. Both times do not take into account delay resulting from concurrent processes and do not include execution time of sub-component functionality.

We also introduce some constraints and assumptions about the execution of the real-time model in order to ensure that real-time analysis is both correct and feasible:

First, it is assumed that the developer will design a real-time component with generic TinyOS components that adhere to the real-time constraints of the component. Second, a real-time split-phase interface may only be invoked once per event occurrence. Encapsulated components cannot be wired to components encapsulated in a different real-time component. All real-time components will execute atomically with respect to each other (this rule is enforced by the real-time port described earlier). Finally, all task periods are equal to their deadlines.

In TinyOS the workload of task τ, is distributed over a hierarchy of interface functions as per Figure 2. As such, if task τ invokes the interface function T_1, then its worst case execution time E_τ, must include the worst-case execution time of T_1, E_1. However T_1 subsequently invokes T_2 and T_3 which in turn invoke other interface functions. Therefore to calculate E_τ, we must first calculate E_i where I_τ is the set of interface functions that are *directly* invoked by τ and $T_i \in I_\tau$. If each interface is only invoked once per event, we can calculate the worst cause execution time of τ, E_τ and the worst case response time R_τ with the following:

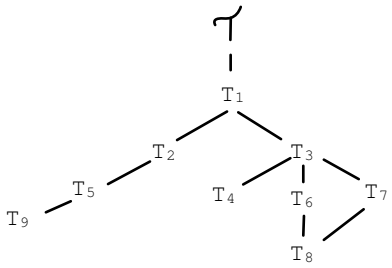

Figure 2. Task Graph

Definition 1. $E_\tau = \sum_{j \in I_\tau} E_j$,$R_\tau = \sum_{j \in I_\tau} R_j$

Subsequently, E_i,R_i will depend on the partial execution and response times e_i,r_i and the worst case execution and response times E_j, R_j such that $T_j \in I_i$. We can therefore calculate E_i and R_i with the following:

Definition 2. $\forall_{T_i}, E_i = e_i + \sum_{j \in I_i} E_j,\ R_i = r_i + \sum_{j \in I_i} R_j$

Having provided a means for determining an interface function execution time, we now turn our attention to the minimum period of interface T_i, x_i. In practise this value will rarely be equal to the minimum period of invocation required by the system P_i, however the constraint is necessary to provide an upper bound on the number of interface invocations. Calculating the required period P_i is complicated by the variable cardinality a component relationship can facilitate. If an interface T_i is used by a number of components, than the interface must be at least able to handle the combined number of invocations of all the connected components. We define Q_i as the set of real-time interface functions that rely on T_i, and endeavour to determine P_i such that we can determine the required period of sub-components I_i and determine if the relationship can be supported, in other words if $x_i \leq P_i$.

Definition 3. $\forall_i$,$P_i = \frac{\min\{P_j | T_j \in Q_i\}}{|Q_i|}$

4.1 Example use of model

We provide a simple configuration of real-time components, in order to demonstrate how our model could be applied to a real-time application. Figure 3 shows a simple outline of two tasks using a set of real time components for a sensing application.

We determine E_i and R_i from Definition 2 above. The task period times P_i are calculated using Definition 3 and we present the results in Table 1. In order to determine if each interface is overloaded, we check if the assigned P_i in Table 1 is greater then the port period constraint in Figure 3. From the table it is

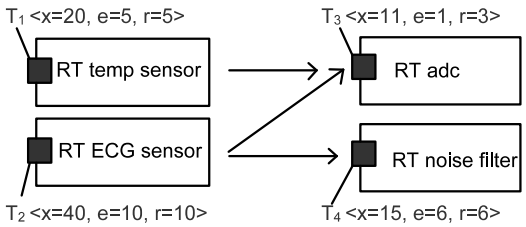

Figure 3. Scenario of Real-Time Components

evident that the relation $T_2 \longrightarrow T_4$ is feasible, but $(T_2, T_1) \longrightarrow T_3$ is not as $x_3 \nleq P_3$.

In this section we have provided a means for developers to check the feasibility of a component relationship and a means to determine execution time, deadline and period requirements necessary for conventional task schedulability algorithms (such as a non-preemptive EDF algorithm [5]). However due to space constraints we can not show detailed results of this model in this paper.

T_i	Q_i	I_i	E_i	R_i	P_i
T_1	$\emptyset$	$\{T_3\}$	$5+1$	$5+3$	20
T_2	$\emptyset$	$\{T_3, T_4\}$	$10+6+1$	$10+6+3$	40
T_3	$\{T_1, T_2\}$	$\emptyset$	1	3	10
T_4	$\{T_1\}$	$\emptyset$	6	6	40

Table 1. Real-Time Interface Calculations

5 Related Work

Improving predictability and real-time performance has been lightly researched by the sensor network community. AmbientRT was one of the first real-time operating systems designed specifically for sensor networks, using a preemptive task based EDF scheduler [6]. Regehr et al. tried to resolve the scheduling latency issues with TinyOS by facilitating task preemption with a multi-threaded task queue as part of a hierarchical concurrency analysis research [7].

Many studies in real-time component systems have been carried out to investigate the problem of constraining components such that they can be independently developed and operate predictably in a component system. Wang et al. proposed a system of component contracts in which each component interface would specify the maximum number of operations it would support and provide a worst-case response time, depending on interface utilization [8]. Shin provided a component specification that facilitated a composition framework

analysis of components [9]. There has been sufficient interest in both real-time sensor networks and real-time component systems to provide a solid foundation for research into real-time component operation in TinyOS.

6 Conclusion

In this paper we have highlighted the problems with implementing a real-time application in TinyOS. Component interfaces do not specify temporal requirements, while event-handlers execute in an unconstrained manner. While TinyOS is uniquely developed for sensor networks, its components can be supplemented with real-time components proposed in this paper, to ensure a more strict and transparent component behaviour. Further more we have shown how a system of component functions can be realized by a common task with defined execution times, in order to determine task schedulability and ensure predictable operation.

References

1. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. System Architecture Directions for Networked Sensors, ASPLOS 2000, Cambridge, Nov 2000.
2. H. C. Lauer, R. M. Needham. On the Duality of Operating System Structures. In Proc. Second Inter-Operating Systems Review, 13, 2, April 1979, pp. 3-19.
3. J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. Real-Time Systems, 21, 3, Nov 2001, pp. 241-268.
4. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. Proceedings of Programming Language Design and Implementation (PLDI) 2003, Jun 2003.
5. K. Jeffay, D. F. Stanat, U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In Proc. Twelfth IEEE Real-Time Systems Symposium, SanAntonio, Texas, Dec 1991, IEEE Computer Society Press, pp. 129-139.
6. T. J. Hofmeijer, S. O. Dulman, P.G. Jansen, P. J. M. Havinga. AmbientRT - real time system software support fordata centric sensor networks. 2nd Int. Conf. on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), published by IEEE Computer Society Press, Los Alamitos, California, held in Melbourne, Australia, Dec 2004, pp 61-66.
7. J. Regehr, A. Reid, K. Webb, M. Parker, J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS2003), Cancun, Mexico, Dec 2003, pp 25-36.
8. S. Wang, S. Rho, R. Bettati, W. Zhao, Real-Time Component-based Systems. In Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Mar 2005.
9. I. Shin and I. Lee, Component-based Design for Real-Time Embedded Systems. Tech. Report, Dept. of Computer and Information Science, University of Pennsylvania, 2005.