University of Thessaly

Master Thesis

Specification and runtime checking of timing constraints in distributed event-based applications

Προσδιορισμός και έλεγχος κατά τη διάρκεια της εκτέλεσης χρονικών περιορισμών σε κατανεμημένες εφαρμογές με γεγονότα

Author:
Nasos Grigoropoulos

Supervisors:
Spyros Lalis
Thanasis Korakis
Petros Lampsas

A thesis submitted in fulfilment of the requirements for the degree of Master in the

Department of Electrical and Computer Engineering
University of Thessaly

Volos, July 15, 2015

ABSTRACT

Distributed monitoring and control applications that rely on wireless sensor and actuator networks need to be developed in a structured and flexible way. Since such applications may have strict timing constraints, static analysis methods and tools can be used to achieve their correct-by-construction design in terms of timing behavior. In addition, dynamic analysis techniques can test their behavior for a number of different scenarios. However, since at run-time violations of the timing constraints can still occur due to a variety of reasons, such as an overly high processing load at some node, or the typically unpredictable and unreliable nature of the wireless medium, it is equally important to monitor the systems performance and detect violations of timing specifications. To this end, we present an event-oriented component model and runtime system that aims on the one hand to simplify the development of distributed event-based applications, and on the other hand monitor their timing behavior in order to detect timing errors. In particular, the user writes the code of the component's event handlers and specifies the desirable timing characteristics, while the runtime system records and logs the time that elapses between the generation of an event and its delivery time, as well as the time it takes for the corresponding handler to process it. In case the respective timing constraint is not met, a violation is reported, while the user can inspect the local logs to determine the cause of failure. Notably, the proposed approach can be used in conjunction with other testing tools during development to assist in the correct specification of these timing constraints.

ΠΕΡΙΛΗΨΗ

Η ανάπτυξη κατανεμημένων εφαρμογών παρακολούθησης και ελέγχου που βασίζονται σε ασύρματα δίκτυα αισθητήρων και ενεργοποιητών πρέπει να γίνεται με ένα δομημένο αλλά και ευέλικτο τρόπο. Στην περίπτωση που αυτές οι εφαρμογές έχουν αυστηρούς χρονικούς περιορισμούς, μέθοδοι και εργαλεία στατικής ανάλυσης μπορούν να χρησιμοποιηθούν για να επιτευχθεί ο σχεδιασμός τους με όσο το δυνατόν προβλέψιμη χρονική συμπεριφορά, ενώ με τη χρήση δυναμικών τεχνικών ανάλυσης καθίσταται δυνατός ο έλεγχος της συμπεριφοράς τους για ένα πλήθος διαφορετικών σεναρίων ενδιαφέροντος. Ωστόσο, δεδομένου ότι στο πραγματικό περιβάλλον εγκατάστασης παραβιάσεις των χρονικών περιορισμών μπορούν να πραγματοποιηθούν για διάφορους λόγους, όπως είναι η υπερβολική αύξηση του φόρτου επεξεργασίας σε κάποιο κόμβο, ή το συνήθως απρόβλεπτο και αναξιόπιστο από τη φύση του ασύρματο μέσο, η παρακολούθηση της απόδοσής τους και ο εντοπισμός παραβιάσεων των χρονικών προδιαγραφών τους κατά το χρόνο εκτέλεσης είναι εξίσου σημαντικά. Προς αυτή την κατεύθυνση, παρουσιάζουμε ένα μοντέλο προγραμματισμού βασισμένο στα γεγονότα και ένα περιβάλλον εκτέλεσης που αποσκοπεί αφενός, στην απλοποίηση της ανάπτυξης κατανεμημένων εφαρμογών με γεγονότα και αφετέρου, στην διευκόλυνση της παρακολούθησης της χρονικής συμπεριφοράς. Συγκεκριμένα, ο χρήστης γράφει τον κώδικα τον χειριστών γεγονότων της εφαρμογής και καθορίζει τα επιθυμητά χαρακτηριστικά χρονισμού τους, ενώ το περιβάλλον εκτέλεσης καταγράφει και αποθηκεύει το χρόνο που μεσολαβεί μεταξύ της παραγωγής ενός γεγονότος και της παράδοσής του, καθώς και το χρόνο που απαιτείται από τον αντίστοιχο χειριστή για την επεξεργασία του. Σε περίπτωση παραβίασης του αντίστοιχου χρονικού περιορισμού, ο χρήστης ειδοποιείται, και ακολούθως μπορεί να επιθεωρήσει την τοπικά αποθηκευμένη πληροφορία για να προσδιορίσει την αιτία της αποτυχίας. Επιπλέον, η προτεινόμενη προσέγγιση μπορεί να χρησιμοποιηθεί στο στάδιο ανάπτυξης των εφαρμογών σε συνδυασμό με άλλα εργαλεία ελέγχου ώστε να συνδράμει στον προσδιορισμό αυτών των χρονικών προδιαγραφών.

ACKNOWLEDGEMENTS

I first and foremost thank my adviser Prof. Spyros Lalis for his useful comments and suggestions throughout this work. During my studies he introduced me to several interesting topics of the computer science while over the past few years his guidance and support have been invaluable. I am also grateful to Prof. Thanasis Korakis and Petros Lampsas, the members of the thesis examination committee, for approving my work.

Many thanks go to my colleague and PhD candidate Manos Koutsoubelias for being a stimulating discussion partner and contributing to a pleasant working environment. Also, I wish to thank my friends, and Eva that not only made life outside of studies really interesting but also helped me evolve personally.

Last but not least, my deepest gratitude goes to my family for the moral support and encouragement throughout the years.

The work in this thesis was partially supported by the SYNERGASIA program of the GSRT, project MariBrain, grant 11SYN-6-288.

MariBrain: "Ship' s Health Condition, Operational Status and Performance Remote Monitoring based on wireless sensor network and technical experience management system".

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE

1

INTRODUCTION

Wireless Sensor Networks (WSNs) are widely used in the monitoring of physical and environmental conditions and are considered appropriate for a wide range of applications, such as intrusion detection, precision agriculture and fire detection, as they simplify deployment and reduce costs. While early WSN deployments were focused in *sensing* the environment and were organized in a centralized way where the gathered data were being reported to a sink node, through the years, and with the inclusion of nodes with *acting* capability, they evolved to more decentralized architectures where application components running on different nodes cooperate in order to achieve a common goal.

In such distributed scenarios, direct interaction between the participating nodes without the mediation of a centralized sink (or a back-end infrastructure) can offer many advantages concerning the introduced latency and the resource utilization of the already resource-constrained nodes. However, since the application's logic is embodied in the network there is a strong need for proper programming abstractions that will allow the programmer to focus on the high-level interactions instead of the implementation details. The event-driven software architecture pattern offers this kind of abstractions by promoting the production and consumption of events among the application components, and hiding the event distribution technicalities from the developer.

Performance-wise, during the application development various techniques can be used to obtain upper bound estimates of the execution time of the applications tasks. Static analysis techniques usually combine the task code with a more or less abstract model of the system while dynamic techniques are based on execution measurements on a given hardware or simulator for a sufficient set of test inputs. In addition to the aforementioned methods, that are primarily focused in the node-level execution of tasks, for applications exhibiting the distributed characteristics described earlier the developers can utilize network-level simulators, while testbeds, large testing infrastructures, can provide valuable experimental measurements over true wireless topologies.

However, even if extensively tested during development, in the real deployment the application may not meet its timing requirements due to a variety of reasons. For instance, the deployment environment can affect the wireless communications causing significant delays, while in overload situations the limited in processing power nodes do not behave so well. In such cases, it is desirable to monitor the timing behavior of the distributed system and detect violations of timing specifications in order to identify possible problematic situations.

Having all of the above in mind, our objective is to create a framework that would give to the application developer the ability to *a*) write distributed monitoring and control applications in a simple way; *b*) define the timing constraints of such applications; *c*) refine them through testing; and *d*) monitor them at run-time.

Our approach addresses (a) and (b) through a lightweight runtime that supports the development of distributed event-based applications together with an API for the specification of their timing constraints, while (c) and (d) are tackled through a decentralized software monitoring system. In a nutshell, an application is composed by software components that communicate in a loosely coupled way through application-defined events. Each component consists of a number of time- and event-triggered operations that have user-defined timing constraints. During development the monitoring system assists in the correct timing characterization by logging the validation results. In the real-world deployment violations are reported directly to the back-end from where the validation logs can be inspected in order to derive information concerning the cause of violation and act accordingly. Therefore, we consider the provided functionality a valuable add-on to both application developers and system administrators.

The rest of this thesis is structured as follows. Chapter 2 points out the key characteristics of our approach and presents the API that can be used to develop such applications. Chapter 3 discusses the application development process with a complete application example. Chapter 4 introduces the implementation platform, in terms of both hardware and software while Chapter 5 discusses the main aspects of our prototype implementation. Chapter 6 demonstrates our framework for a violation scenario and evaluates our implementation in terms of its memory and performance overhead. Chapter 7 discusses related work. Finally, Chapter 8 concludes the thesis and provides directions for future work.

2

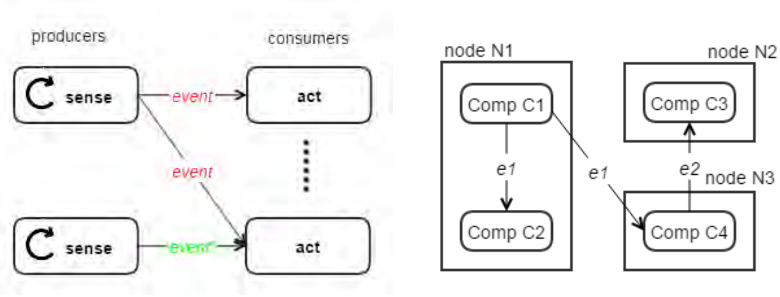
BASIC CONCEPTS AND API

2.1 KEY CHARACTERISTICS

The main goal of our model is to promote a structured design of monitoring and control applications, which is also sufficiently flexible so as to enable the seamless distribution of the application logic in the WSAN. Ideally, the developer should program the application and specify the respective timing requirements without having to worry about its actual physical placement in the WSAN. Of course, at deployment time, the different parts of the application must be placed onto the "proper" nodes, the ones that feature the required sensors and/or actuators. The key aspects of both our application composition and timing specification model are described in the following.

Application composition

The application is structured as a collection of different software *components* that communicate via *events*, which are defined by an application-specific event ontology. A component can publish (produce) and subscribe for (consume) specific events. In the typical scenario, producers are responsible for detecting certain conditions, e.g., by sensing the environment, and informing the consumers who in turn are responsible for applying an appropriate reaction, as shown in Figure 2-1a.



(a) Interaction logic (b) Composition example

Figure 2-1: Key characteristics of the component model

A key feature of the components interaction is that components need not to know about one another in order to exchange events, they merely need to be producers and consumers of matching events. It is the responsibility of the underlying runtime system to deliver the produced events to the respective consumers. Also, a producer is not aware of the consumer's reaction which may not be self-contained, e.g., an event handling operation can involve processing/filtering and may lead to the production of another event. This *loose coupling* between the application's components enables the flexible deployment of the application in the WSAN, while also allows the easy extensibility/ adjustment of an already deployed application, with the addition, removal, replication and replacement of components.

This concept is illustrated in Figure 2-1b. The application example includes four components *C1*, *C2*, *C3* and *C4*. Component *C1* produces events of type *e1* which are consumed by components *C2* and *C4*. In addition, *C4* produces events of type *e2* consumed by *C3*. While in the depicted deployment scenario *C1* and *C2* are co-located on node *N1* and the rest are placed on different nodes, in principal, all the possible combinations would be feasible, without the need to change the application code.

Component execution

Each component is a collection of *event handlers*, which are up-called by the underlying runtime system whenever an event is delivered to the component, as shown in Figure 2-2. Inside a handler the component can read a sensor, drive an actuator, do some processing and publish one or more events. Note, that event handlers should be non-blocking, hence they are not allowed to contain any endless loops. In the same spirit, the event publication operation, which occurs via down-calls through the runtime API, is non-blocking and returns immediately.

Aside from the handlers for application-level events, a component may have special handlers which are invoked at a user-specified period. We refer to those handlers as *tasks*, and their main usage is to poll sensors.

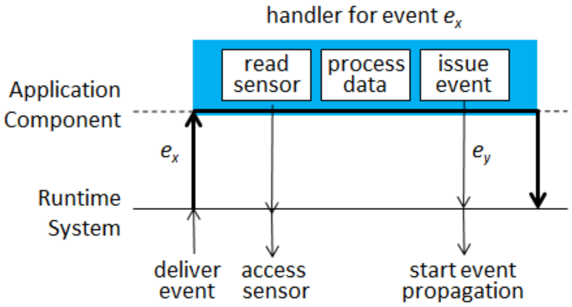


Figure 2-2: Component execution.

Timing specification model

Our timing specification model is focused in the responsiveness, which is considered an important performance metric for such sense-and-react systems. More specifically, the programmer can specify the desirable response time, T_{rsp}, of handlers which is defined as the time elapsed from the moment the event is issued until the moment the event handler completes its execution. As shown in Figure 2-3, this interval can be decomposed into $T_{delivery}$, the time it took for the event to be delivered to the component, and $T_{process}$, the time it took the handler to process the event.

Subsequently, $T_{delivery}$ can be further decomposed into the system-internal delays at the source node (T_{src}) and the destination node (T_{dst}), as well as the event transfer time ($T_{transfer}$). T_{src} and T_{dst} capture the queuing/scheduling delays at the respective nodes, while $T_{transfer}$ captures the transmission time and the network stack delays. We currently treat the network as a "black box" and do not further decompose $T_{transfer}$. As a consequence, it includes re-transmissions and routing overhead (in multi-hop networks).

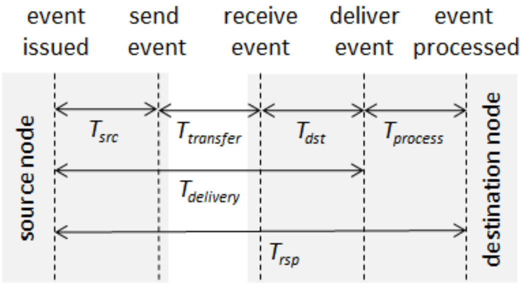


Figure 2-3: Illustration of timing constraints.

Obviously, for events that are delivered locally $T_{transfer} = T_{dst} = 0$, and the delivery delay equals the time the event remains in the queue before it is delivered to the application, i.e., $T_{delivery} = T_{src}$. Since timeouts can be viewed as a special type of local events, for tasks also stands that $T_{delivery} = T_{src}$.

2.2 API

The programming constructs that can be used to build such distributed applications are introduced as macros on top of the C programming language, and are listed in Table 2-2, placed for reference at the end of the section. In the following, these constructs are presented through a simple component that is part of a ping-pong application, depicted in Figure 2-4. More specifically, the component in Listing 2-1 periodically increments a counter and publishes it using a `PING_EVENT` event while also handles `PONG_EVENT` events.

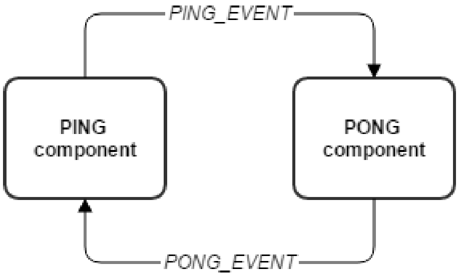


Figure 2-4: Ping-pong application interactions.

The interaction capability of the component, i.e., the events it produces and the ones it handles, are explicitly declared using the **PRODUCES** and **CONSUMES** macros that take as parameter a comma separated list of the respective event ids (line 2-3), while its name (`ping`) and its textual description (`"Ping sender"`) are specified using the **COMPONENT** macro.

A component's functionality is implemented in a number of tasks and event handlers (lines 7-18), that can share global data (e.g., line 5). Besides application-level events, there are also system-level ones automatically generated by our run-time that have to do with the life cycle of the component. The user can implement handlers for them in order to initialize the global variables and allocate/free resources.

The association of a handler with an event, and the assignment of an invocation period to a task is done using the macros listed in the second part of Table 2-2. More specifically, **ON_INIT** and **ON_FIN** specify the functions invoked at the initialization and the finalization of the component, and can be `NULL` if not needed (e.g., line 23). Tasks are declared with the **TASK** macro that assigns a unique numerical id to them and sets their invocation period, as well as their response time constraint (line 21). Finally, the **HANDLER** macro associates a function with an application-level event and also sets its maximum response time (line 22).

At this point there are two things that need to be noted: *a*) system-level event handlers do not have user-defined timing constraints as they are expected to be executed just once; and *b*) the functions have different prototypes depending of their functionality as shown in Table 2-1. In particular, application-level event handlers take as parameters a pointer to the event-specific data and the data length (line 16), while tasks and system-level event handlers are parameterless (line 7, 11).

According to its logic, a component can interact with others (passing data or just notifying) by calling the non-blocking **PUBLISH** function that takes as parameters the event type, a pointer to the event payload and the payload's length. For instance, in Listing 2-1, line 10, a `PING_EVENT` event containing a counter is published.

Listing 2-1: A simple component

```
COMPONENT(ping, "Ping sender");
PRODUCES(PING_EVENT);
CONSUMES(PONG_EVENT);

uint8_t counter;

void init(void){
    counter = 1;
}

void send_ping(void){
    counter++;
    PUBLISH(PING_EV, &counter, sizeof(uint8_t));
}

void handle_pong(void *data, uint8_t len){
    BLINK_LEDS(RED);
}

ON_INIT(init)
TASK(send_ping, 0, SECS(5), MILLIS(50))
HANDLER(handle_pong, PONG_EVENT, MILLIS(150))
ON_FIN(NULL)
```

Table 2-1: Function signatures

Function prototype	**Description**
void funcName(void);	Used by system-level event handlers and tasks.
void funcName(void *data, uint8_t len);	Used by application-level event handlers. *Data* is a pointer to a byte array containing the event-specific payload and *len* the payload's length.

Note that the ping component does not direct the published event to a specific component; it is programmer's responsibility to define the appropriate interaction capability to the pong component. Also, ping is not aware of the event consumer's location; its implementation remains the same either they are installed on the same or on remote nodes. Nonetheless, the physical placement of the components affects the response time of the PING_EVENT handler.

Table 2-2: Basic primitives

Statement	Description
COMPONENT(*name, nameStr*)	Declares a component named *name* with *nameStr* human readable description.
PRODUCES(*eventIDs*)	Declares the types of events that can be produced by the component.
CONSUMES(*eventIDs*)	Declares the types of events that can be consumed by the component.
ON_INIT(*funcName*)	Specifies the function invoked (once) upon the loading/instantiation of the component.
ON_FIN(*funcName*)	Specifies the function invoked (once) upon the unloading/termination of the component.
TASK(*funcName, taskID, prd, rspT*)	Specifies a function scheduled periodically every *prd* time units, under the identifier *taskID*, with a desired response time *rspT*.
HANDLER(*funcName, eventID, rspT*)	Specifies a function invoked upon the reception of the application-level *eventID* event, with a desired response time *rspT*.
PUBLISH(*eventID, data, len*)	Publish an application-level event, with identifier *eventID*, payload *void *data* and payload size *uint8_t len.*

3

APPLICATION DEVELOPMENT

3.1 WORKFLOW

The application development process that we envision has four stages, as illustrated in Figure 3-1.

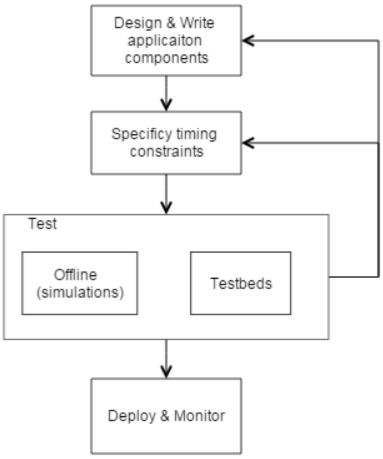


Figure 3-1: Application development process.

As a first step, the developer has to design the system components, define the event ontology of the application and implement them using the API described in section 2.2.

Then, for each task and event handler of the implemented components, he has to specify the desired timing characteristics. This is not an one-off process and the developer is expected to return several times to make adjustments.

The testing step is crucial for the validation of the application's logic, as well as for the refinement of the timing constraints before the actual deployment, and a number of tools[1] can be used for this purpose. In a first phase, using simulators and emulators the developer can verify the interactions between the application components and also get good estimates of the application's timing behavior. In addition, the application can be deployed in testbeds to observe its behavior in real

[1] For more details on pre-deployment tools see section 7.2 of Related Work Chapter.

nodes. In each testing phase, the developer can experiment with different topologies and component placements, and based on the gathered information he can go back to one of the previous steps to revise the timing constraints or in the extreme case alter the application by merging/splitting components.

Finally, once the application has been sufficiently tested and has reached a stable state regarding both its structure and its timing specifications, it can be deployed in the field by installing the components on the nodes of interest. In this stage, the timing constraints are being monitored consistently and the system administrator will get alerted in case a violation occurs in order to act accordingly. At the same time, by inspecting the logs of the monitoring system on each node, he can have an overview of the application's performance and deduce possible overloading situations.

3.2 CASE STUDY

Consider the case of a fire alarm system. This is a typical example of a sense-and-react system, where a number of devices cooperate to *detect* the presence of fire and *notify* people through visual and audio means. Using Wireless Sensor Network technology such a system can be built utilizing nodes with sensing and acting capability. In the following we describe how a simple fire alarm application can be developed using our programming model and following the proposed work-flow. Note however that the application's functionality is indicative.

Application design - Component implementation

During the decomposition of the application's functionality, the required components can be classified to *fire detectors* and *fire notifiers*. The former monitor different parameters of the environment and when specific conditions are met they inform the latter in order to act accordingly.

In order to simplify the implementation, we do not distinguish the fire detection conditions and the means of notification is a single event indicating the possibility of fire. Therefore, we let the fire notifying components act proactively if just one of the conditions is met.

The application components considered in our example and their functionality are as follows:

- Fire detectors
  - temperature monitor, reads periodically the temperature sensor and checks if a significant temperature increase occurs or a predefined threshold is reached
  - gas monitor, reads periodically the gas sensor and checks if the carbon monoxide concentration exceeds the safe limit

- Fire notifiers
  - visual alarm, issues a visual alert by blinking a red led
  - audio alarm, issues an audio alert

Their classification and high-level interactions are shown in Figure 3-2 while their actual implementation is presented in Listings 3-1- 3-4. Note that since the detection logic is implemented entirely in the fire detection components, the event they produce does not include any payload. Also, the timing constraints in the components implementation are left blank as their specification process is discussed in the sequel.

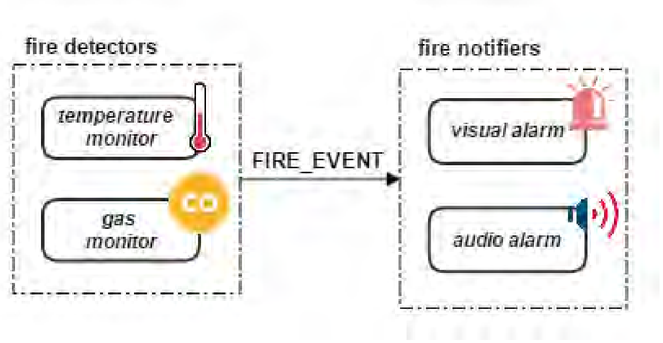


Figure 3-2: High-level structure of the fire alarm application.

Listing 3-1: Temperature monitor component

```
/* Threshold in Celcius degrees */
#define TEMP_THRESHOLD      60
/* Rate of rise */
#define TEMP_INCR           7
COMPONENT(temp_monitor, "Temperature monitor");
PRODUCES(FIRE_EVENT);
CONSUMES(NONE);

int32_t prev_temp; /* previous reading */

void read_temp(void){
    int32_t temp = get_temp();
    if ((temp - prev_temp) > TEMP_INCR) ||
                    (temp > TEMP_THRESHOLD)){
        PUBLISH(FIRE_EVENT, null, 0);
    }
    prev_temp = temp;
}

ON_INIT(NULL)
TASK(read_temp, 0, SECS(10), -)
ON_FIN(NULL)
```

Listing 3-2: Gas monitor component

```
/* Threshold in ppm */
#define CO_THRESHOLD      65
COMPONENT(gas_monitor, "Gas monitor");
PRODUCES(FIRE_EVENT);
CONSUMES(NONE);

void check_gas(void){
    int32_t concentration = get_gas();
    if (concentration > CO_THRESHOLD)) {
        PUBLISH(FIRE_EVENT, null, 0);
    }
}

ON_INIT(NULL)
TASK(check_gas, 0, SECS(5), -)
ON_FIN(NULL)
```

Listing 3-3: Audio alarm component

```
COMPONENT(audio_alarm, "Audio fire alarm");
PRODUCES(NONE);
CONSUMES(FIRE_EVENT);

void handle_fire(void *data, uint8_t len){
    beep();
}

ON_INIT(NULL)
HANDLER(handle_temp, FIRE_EVENT, -)
ON_FIN(NULL)
```

Listing 3-4: Visual alarm component

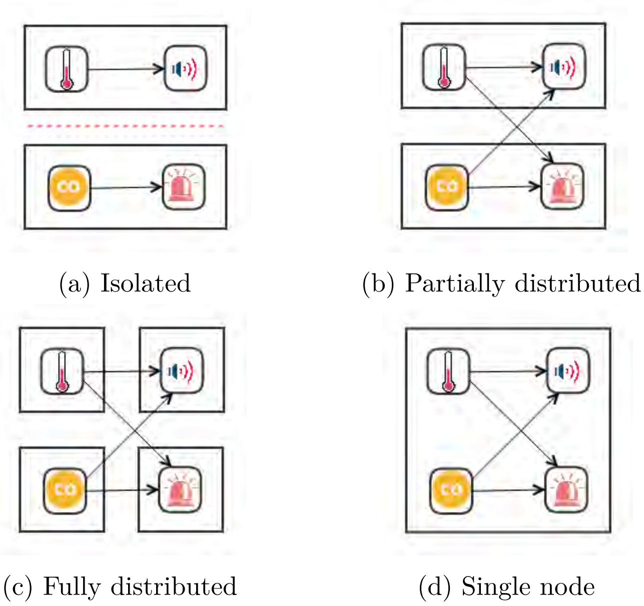
```
COMPONENT(visual_alarm, "Visual fire alarm");
PRODUCES(NONE);
CONSUMES(FIRE_EVENT);

void init(void){
    leds_off(LEDS_ALL);
}

void handle_fire(void *data, uint8_t len){
    leds_toggle(LEDS_RED);
}

void fin(void){
    leds_off(LEDS_ALL);
}

ON_INIT(init)
HANDLER(handle_temp, FIRE_EVENT, -)
ON_FIN(fin)
```



(a) Isolated

(b) Partially distributed

(c) Fully distributed

(d) Single node

Figure 3-3: Indicative deployment scenarios of the fire alarm application.

Specification of timing constraints - Testing

Since the developer can not be aware of the timing constraints, at first they can be rough estimates. However, through testing he can try out several setups, as shown in Figure 3-3, and observe the application's timing behavior. For instance, using one of the "isolated" deployments illustrated in Figure 3-3a, where some of the components are excluded and the remaining are installed on the same node, he can obtain the lower bounds of the response time for the handlers of each fire notifier. Then, by deploying more distributed scenarios , like the ones depicted in Figures 3-3b and 3-3c, and through experimentation with different radio interference models he can observe the handlers response time with the network latency taken into account. In the fully distributed case, in addition, the ideal response time for the tasks of the fire detectors can be obtained since the local load is minimized. On the contrary, the maximum local load is created in the setup of Figure 3-3d where all the components are installed on a single node. Intuitively, this will lead to the observation of larger response times for the tasks of the fire detectors.

While some of these component placements may not be realized in the actual deployment, e.g., the last one requires multi-sensor nodes, all of them can provide insight concerning the timing bounds of the application. Thus, it is generally a good practice to try as many configurations as possible in the simulation environment. Also, recall that component interactions are based on producer/consumer relationships and are not related to their physical placement. Therefore, in all of the aforementioned cases no changes are required in the components code; a fact that can simplify and significantly accelerate the testing process.

More realistic measurements can be obtained once the testing procedure moves to a testbed. Although it is difficult to reproduce the fire alarm conditions on the testbed nodes, since they report real sensor measurements, the developer can employ a mechanism like the virtual onboard sensors [1] in order to let the sensors produce the desired artificial data. This way, even on real nodes the application's code can remain untouched.

Deployment

Once the developer feels confident about the application's state, he can proceed to the real-world installation and observe at run-time its timing behavior. It is important to stress the fact that the monitoring mechanism used in the field is the same one used during testing. Hence, the monitoring operation's overhead has been already taken into account during the process of specifying the timing constraints.

4

IMPLEMENTATION PLATFORM

In this chapter we present some relevant aspects of the implementation environment, in terms of both software and hardware, upon which our prototype was built, namely Contiki OS and Tmote Sky.

4.1 CONTIKI OS

Contiki [2] is an open source operating system for low-power, memory-constrained, networked embedded systems, such as WSNs, written in the C language. It was created by Adam Dunkels at the Swedish Institute of Computer Science (SICS) in 2002 and has been further developed by a world-wide team of developers, from both academia and industry, constituting a large, active community. Contiki is built around an event-driven kernel and inter-process communication is implemented using message passing via events. Among its main features are the protothreads, a low-overhead mechanism for multitasking, the dynamic loading and replacement of code at run-time, and its particular focus on the Internet of Things (IoT).

Architecture

Unlike other embedded OS's, like TinyOS, where the whole executable is statically linked at compile time, Contiki's architecture is modular allowing the dynamic loading of code at run-time. A Contiki system is partitioned into the core and the loaded programs as shown in Figure 4-1. The core, which typically consists of the kernel, the program loader, libraries, the device drivers and a communication service is compiled into a single binary and generally once installed, is not modified. In contrast, the loaded programs can be distributed independently and can be application processes and services. Their binaries are obtained using a wireless or wired communication interface and the program loader is responsible for their integration in the running system.

The kernel is a lightweight scheduler that removes events from the event queue and dispatches them to the running processes. The processes are implemented as general purpose event handlers whose execution is not interrupted by the kernel and therefore always run to completion. Communication between them is realized using events that go

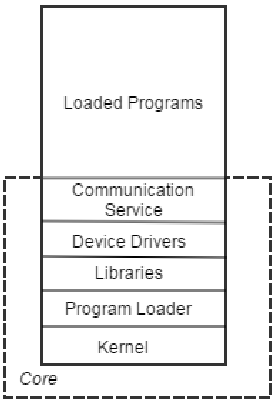
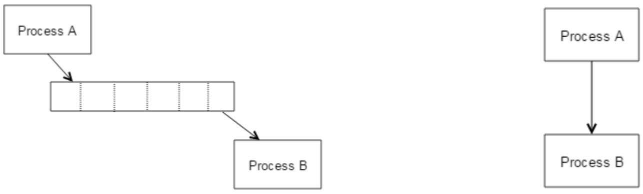


Figure 4-1: Partitioning of a Contiki system.

through the kernel. The events can be synchronous or asynchronous; the former are put in the event queue and delivered some time later while the latter are delivered directly being functionally equivalent to a function call (see Figure 4-2).



(a) Asynchronous event (b) Synchronous event

Figure 4-2: Event delivery in Contiki

Programming model

Contiki processes are implemented using the programming abstraction of protothreads [3] that provides a conditional blocking wait statement. This allows the developers to contain the high-level logic of their program in a single process which has sequential structure, in contrast to traditional event-driven systems where the control flow has to be expressed using a state machine. Protothreads are lightweight, since they run on the same stack, and context switching is done by stack rewinding.

Dynamic loading

To facilitate reprogramming and code swapping, Contiki supports the loading of code modules at runtime [4]. The loadable modules must be in ELF format and may contain a standalone application or a service that is intended to replace one already running. Once the ELF binary is received by a Contiki system, it is typically stored in an external storage unit and the dynamic loader is responsible for its linking, relocation and loading into the system's image.

Supported communication protocols

Contiki supports three networking mechanisms that can be used depending on the application's requirements. More specifically, two implementations of the TCP/IP stack for 8-bit/16-bit microcontrollers exist, namely uIP [5] and uIPv6 [6], offering IPv4 and IPv6 connectivity respectively. For cases where the IP connectivity is not necessary, Rime [7], a lightweight and multi-layered protocol stack offering best-effort and reliable communication primitives, can be used instead.

File system

Contiki provides a minimalistic, yet powerfull, flat file system for flash memories named Coffee [4]. Beyond the API for basic file operations (open/close, read/write, seek, remove) Coffee offers the ability for more efficient memory usage through the micro-logs structure it introduced while additionally it features a garbage collection mechanism for managing the memory's wear levelling.

Simulation support

COOJA [8], Contiki's network simulator, offers the ability to run simulation scenarios with three types of nodes: nodes implemented in Java using the COOJA API, nodes that run the Contiki OS compiled for the host computer and nodes that run Contiki OS on an emulated MSP430 platform. The latest feature is offered through the integrated MSPSim [9] instruction-level emulator.

4.2 TMOTE SKY

Tmote Sky [10] is a wireless sensor platform widely used in both industry and research. It is built upon the Texas Instruments MSP430F1611 microcontroller that features 10 KB of RAM, 48KB of flash (ROM) and a number of peripherals such as ADC and DAC modules, timers and USART modules. It is equipped with leds, on-board humidity, temperature and light sensors, an 1 MB external flash storage unit and the 802.15.4-compatible CC2420 radio chip for the wireless communication. Sky was among the first hardware platforms fully supporting Contiki while using MSPSim in COOJA simulations it is possible to emulate the complete platform behavior including its peripherals.

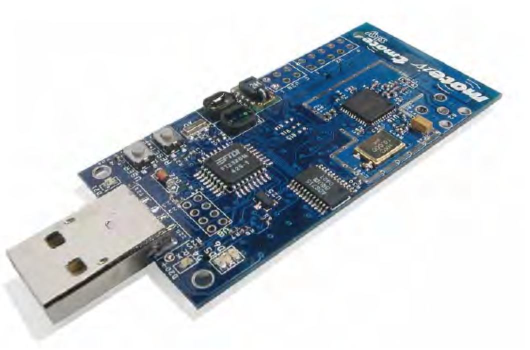


Figure 4-3: The TMote Sky platform.

4.3 development environment

The development of our work was conducted in the environment provided by the Contiki community, namely the Instant Contiki virtual machine, using the MSPGCC toolchain (ver. 4.7.0). All software developed is based on the version 2.7 of Contiki while the Contiki kernel was left unmodified.

IMPLEMENTATION

In this Chapter we present the key aspects of our design and prototype implementation. In the first section we outline the architecture and the main parts of a run-time system that supports the development of distributed event-based applications, as well as the monitoring of their timing behavior, while in the following we describe in detail their interactions/operation. Lastly, we provide a concrete description of how the run-time support is implemented over the Contiki OS.

5.1 MIDDLEWARE ARCHITECTURE

The runtime system is composed of three core services, namely the *Event Manager*, the *Network Manager* and the *Timing Monitor*. These elements and their interactions are depicted in Figure 5-1.

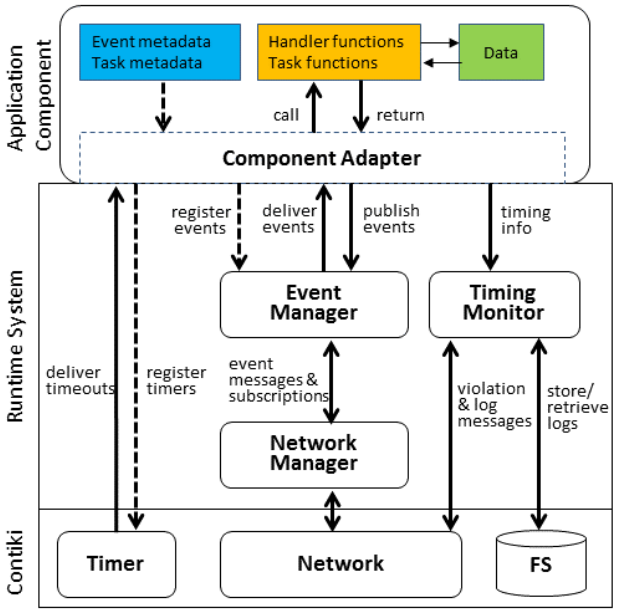


Figure 5-1: Middleware overview.

The *Event Manager* provides the event distribution mechanism to the application components, which essentially is a distributed publish/subscribe service. The *Network Manager* is responsible for connecting a node with the network. It receives events from the *Event Manager*, constructs the respective packets and forwards them to the underlying network stack in order to be disseminated to the network and vice versa. The *Timing Monitor* is responsible for checking the timing behavior of the locally installed components, logging the validation results and reporting violations. Finally, the *Component Adapter* is a special glue layer that connects the application components with the runtime system by exposing to them the public runtime API, while in addition intercepts their execution and collects the respective timing information for checking their constraints.

5.2 COMPONENT OPERATION

Initialization

When a component is loaded in the system, the *Component Adapter* retrieves its produced and consumed events and registers them to the *Event Manager* who keeps track of this information for all the locally installed components. The *Event Manager* periodically broadcasts the ”aggregated” event subscriptions of the node via advertisement messages through the *Network Manager*. Upon the reception of such a message a check for matches between the events contained in the message and the events from the local publishing list takes place. For each matched event, an entry in a forwarding table associating the event with the message sender node is created. In addition, at component loading the *Component Adapter* retrieves the information about its tasks and registers the respective timers to the underlying OS timer module. The respective interactions are depicted in Figure 5-2.

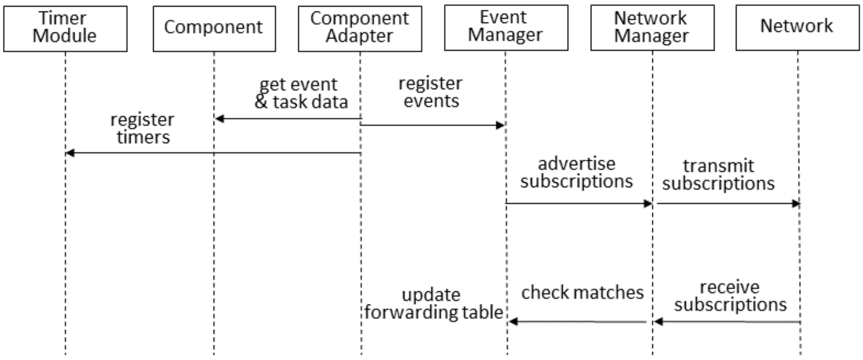


Figure 5-2: Component initialization sequence diagram.

Interaction

When a component publishes an event this is dispatched asynchronously to the *Event Manager*. The *Event Manager* firstly checks if there are any local components subscribed for this event type and in case this is true the event is delivered synchronously to each one of them, in a serial manner. Subsequently, it checks the forwarding table for remote nodes associated with the specific event and in case there are any it invokes the *Network's Manager* functions that take care of creating and transmitting the respective event messages.

Upon the reception of an event message, the *Network Manager* dispatches it to the *Event Manager* who in turn delivers it to the locally subscribed components. Finally, when a component receives an event the appropriate handler is invoked. The described event publishing and handling procedures are depicted in Figure 5-3.

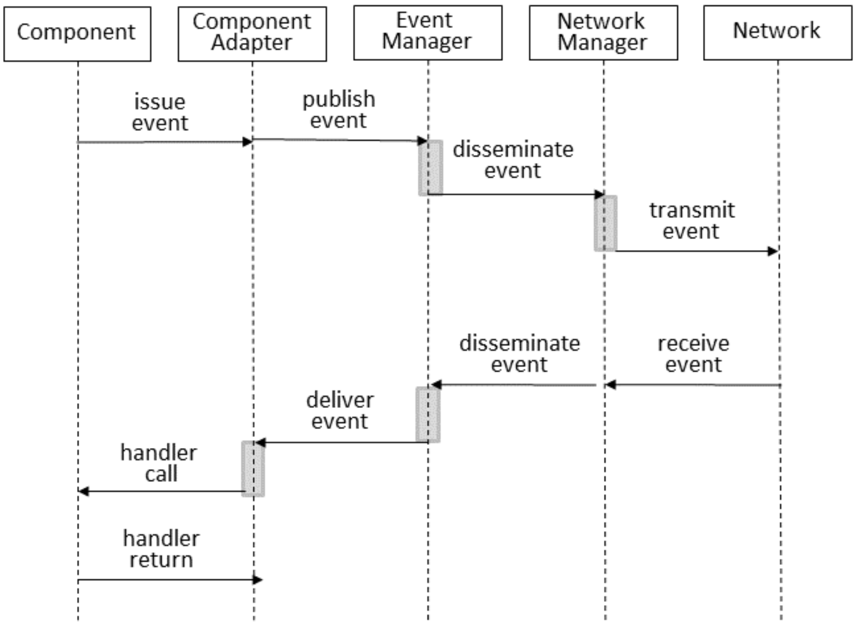


Figure 5-3: Event publishing and handling sequence diagram.

Our prototype implementation deals with one-hop networks but the provided functionality can be easily extended to multi-hop networks by employing a network-wide broadcast mechanism and an ad-hoc routing protocol like AODV [11]. Alternatively, one can use mature ad-hoc wireless networking technologies, such as ZigBee [12], that come with built-in routing support.

5.3 MONITORING OF TIMING CONSTRAINTS

In order to monitor the response time constraint of tasks/handlers at run-time knowledge of their release time, i.e., the issue time of the event that causes their execution, and their end of execution time is required. For tasks the issue time is retrieved through the system's timer interface while for application-level event handlers it is encapsulated in the event as meta-data. Furthermore, in accordance with the timing model presented in section 2.1, and in order to provide better monitoring functionality, additional timestamps are included during the event flow from the producer to the consumer as shown in Figure 5-4.

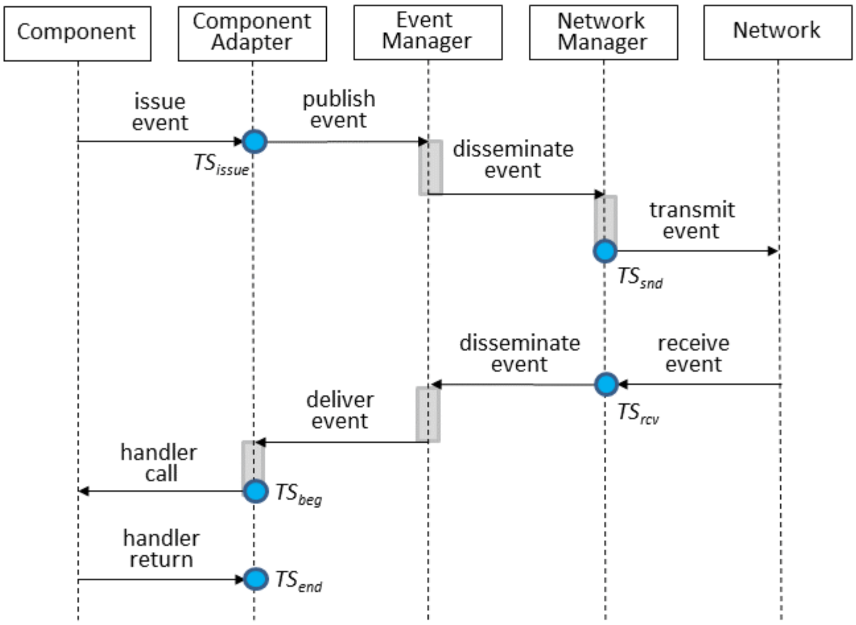


Figure 5-4: Timestamping points in the event flow.

Since a locally generated event may be handled in a remote node, time synchronization is required. To achieve this, the nodes must have their clocks synchronized or alternatively have knowledge of their clocks offset in respect to the other nodes. For our prototype implementation the second approach was chosen and upon the reception of a remote event, its occurrence time stamp is adjusted accordingly.

The message exchanges of the synchronization protocol are depicted in Figure 5-5 and are similar to the Cristian's algorithm [13]. More specifically, each node periodically broadcasts a `SYNC_REQ` message to the network and waits for a timeout. When a `SYNC_REQ` is received by a node, it responds to the originator via a unicast `SYNC_RESP` message that includes its current timestamp (T_{remote}). When the originator receives a `SYNC_RESP` message within the timeout, it calculates the phase offset of the remote node. The network delay, assuming that is equal in both directions, is estimated by measuring in the originator's side the time at which the request is sent (T_{snd}) and the time at which a response is received (T_{rcv}) and is equal to $T_{net} = (T_{rcv} - T_{snd})/2$. The phase offset is then calculated according to the following formula:

$$T_{offset} = |T_{rcv} - (T_{remote} + T_{net})|)$$

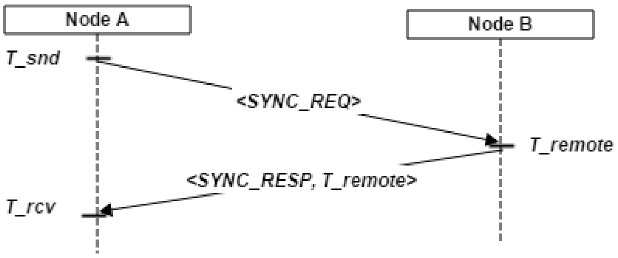


Figure 5-5: Synchronization protocol.

Whenever a task/handler is executed, the *Component Adapter* collects the timestamps, retrieves the response time constraint along its identification data and report them to the *Timing Monitor* who evaluates them and logs the results.

Each log consists of the component's name, the task's/handler's identifier, the response time constraint and the response time breakdown. In addition, information about the event's source (local/remote) and the local time stamp are saved.

In case a violation occurs, a violation report, composed by the task's/ handler's identification data and the local time stamp, is delivered to the end user. The user in turn has the ability to inspect/query/retrieve the local logs of a node through a request-reply protocol. Additionally, utilizing the information of a violation report he can optionally specify the component of interest whose logs are to be returned. The respective message exchanges are depicted in Figure 5-6.

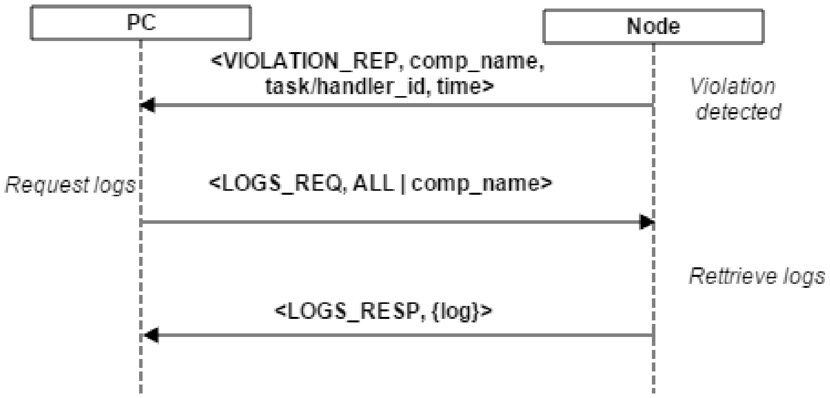

PC
Node
<VIOLATION_REP, comp_name, task/handler_id, time>
Violation detected
Request logs
<LOGS_REQ, ALL | comp_name>
Rettrieve logs
<LOGS_RESP, {log}>

Figure 5-6: Violation reports and log retrieval sequence diagram.

Violation reports and log requests/responses can be delivered through a local serial connection or through a wireless interface to a distinguished gateway node that has the responsibility of notifying the end user. The first option is feasible for testbed environments while the latter is more suitable for real deployments. The address of this distinguished gateway node, as well as the memory type used for the logs history storage and its reserved size are specified via a configuration file.

5.4 RUNTIME SUPPORT

Since our runtime is built on top of the Contiki operating system [2] it relies on the C programming language. It provides the constructs described in Section 2.2 by making extensive use of C macros and moving most of the added complexity at compilation time. Also, a minimal amount of pre-processing is required. This is done using a dedicated pre-processor that takes as input a component's source file and creates the Contiki-valid code. The code transformation process, demonstrated in Listings 5-1 and 5-2 for the example `ping` component of section 2.2, is as follows:

- the `COMPONENT(<name>,<stringName>)` statement is split in a `#define COMPONENT_NAME <name>` placed in the start of the file (line 1) and a `COMPONENT(<stringName>)` macro placed after the declaration of the publishable and subscribed events (line 4).
- global variables are declared as static (line 6)
- for each task, a timer is declared (line 7) and a reference to it is added in the respective `TASK` macro (line 16).

Listing 5-1: Pre-processed code

```
 1  COMPONENT(ping, "Ping sender");
 2  PRODUCES(PING_EVENT);
 3  CONSUMES(PONG_EVENT);
 4
 5
 6  uint8_t counter;
 7
 8
 9  /* Fuction definitions */
10  void init(void){...}
11  void send_ping(void){...}
12  void handle_pong(void* data,
13                  uint8_t len){...}
14
15  ON_INIT(init)
16  TASK(send_ping, 0, 5000, 50)
17  HANDLER(handle_pong, PONG_EV, 150)
18  ON_FIN(NULL)
```

Listing 5-2: Contiki-valid code

```
#define COMPONENT_NAME ping
PRODUCES(PING_EVENT);
CONSUMES(PONG_EVENT);
COMPONENT("Ping sender");

static uint8_t counter;
static struct etimer et;

/* Fuction definitions unchanged */
...//




ON_INIT(init)
TASK(send_ping, 0, &et, 5000, 50)
HANDLER(handle_pong, PONG_EV, 150)
ON_FIN(NULL)
```

Components are mapped to Contiki processes and their internal representation, called the component control block (CCB), is a struct encapsulating a Contiki process control block with some extra fields (see Listing 5-3). The first field, `next`, points to the next component control block in the linked list of active components. Name is a pointer to the component's textual name while id is a numerical identifier which is assigned at runtime. Produces/consumes point to the lists of events declared through the PRODUCE and CONSUME macros and finally state holds the component's current state.

Listing 5-3: The component control block

```
struct component {
    struct component *next;    /* pointer to the next component */
    struct process *process;   /* pointer to the process control block */
    void *name;                /* pointer to the component/process name */
    uint8_t id;                /* component id, assigned by the runtime */
    void *produces;            /* pointer to the publishable events */
    void *consumes;            /* pointer to the subscribed events */
    uint8_t state;             /* component state */
};
```

The component control block is not declared or defined directly, but through the COMPONENT() macro, which takes as parameter the textual name of the component, while its variable name is defined indirectly through the COMPONENT_NAME define.

For structuring the component's code instead of using the protothreads abstraction [3], a simpler event-driven approach is followed. More specifically, the ON_INIT, TASK, HANDLER and ON_FIN macros form the component's functions look-up table. The latter additionally defines the respective Contiki process thread whose body includes a single call to a dedicated function of the Component Adapter, responsible for invoking the appropriate function (task/handler).

The Component Adapter also controls the components life cycle. When the runtime is initialized, this module reads the lists of publishable/subscribed events of all the components included in an ”autostart” list, in the spirit of Contiki's respective list, and registers them to the Event Manager. After the advertisement phase is finished it posts synchronously the standard Contiki process initialization event to all of them serially, starts their timers and sets their state to ACTIVE. In this state, a component can receive/publish events and execute its periodic operations. Moreover, since Contiki does not support periodic timers, whenever a task is invoked this module is also responsible for resetting the associated timer. A component can be deactivated only through an explicit request from the back-end which is handled by the Component Adapter.

All the core runtime services, i.e., Event Manager, Network Manager and Timing Monitor are implemented as Contiki processes that receive and send custom Contiki events. More specifically, when the Network Manager receives a packet containing an application event it posts REMOTE_PUBLISHING event to the Event Manager. The latter also handles LOCAL_PUBLISHING events posted by the components through the Component Adapter's PUBLISH statement. While these events are delivered asynchronously through Contiki's general event queue, the dispatching of application-level events from the Event Manager to the components is done via synchronous events. Finally, the Timing Monitor receives asynchronous CONSTRAINTS_CHECK events from the Component Adapter whenever a handler completes its execution, logs the checking results and in case of a violation transmits a violation report message to the back-end. In case the external flash is selected as the storage medium of the logs, the Coffee File System [4] is being used.

6

EVALUATION

In this Chapter we evaluate our prototype implementation in terms of its memory and performance overhead, as well as its timing violation detection functionality.

6.1 MEMORY FOOTPRINT

The memory requirements of our runtime system are analyzed in Table 6-1. The binary code accounts for less than 4.5 Kbytes in total, which is roughly a 15% increase over the basic Contiki OS including the Rime stack layers for one-hop communication and the Coffee file system. Apart from a small fixed overhead, RAM consumption depends on several parameters, whose (maximum) values can be set when building the system. Specifically, the additional RAM required by the *Event Manager* is $X = 24 \times e + s + 2 \times (p + c) + 8 \times f$, where e is the size of the application-level event queue, p is the total event payload size, p and c are the number of locally produced and consumed events, and f is the number of entries in the event forwarding table. For the *Network Manager*, the extra RAM usage is $Y = 6 \times o$, where o is the number of entries in the clock offset table. Finally, for the *Timing Monitor* the additional RAM required is $Z = 42 \times l$, where l is the size of the log history ($l = 0$ if logs are kept on the external flash). As an example, if $p = c = f = e = 5$, $s = 100$, $o = 1$ and $l = 0$, the total amount of RAM used is 378 B. For this configuration, the combined image of Contiki and our runtime system leaves over 32% of ROM and 69% of RAM free for application development on the TMote Sky.

Table 6-1: Memory consumption of the runtime system

Module	**Code (B)**	**RAM usage (B)**
Component Adapter	1198	0
Event Manager	1116	$34 + X$
Network Manager	1806	$40 + Y$
Timing Monitor	232	$18 + Z$
Total	4352	$92 + X + Y + Z$

Table 6-2: Execution overhead of the runtime system

Configuration	**Response time (ms)**	**Overhead (ms)**
Native Contiki application	0.098	-
Runtime with basic event delivery	0.386	0.288
+ timing violation detection	0.671	0.573
+ logging in RAM	0.750	0.652
+ logging in FLASH	3.042	2.944

6.2 PERFORMANCE OVERHEAD

To assess the time penalty introduced by our runtime system, we have measured the event response time for a test application comprised of two components that interact in a ping-pong event loop. The application does not perform any actual processing of the events. Both components reside on the same TMote Sky node running at 3.9 MHz, and the results are averaged over 1000 iterations. Table 6-2 gives the end-to-end response time and overhead for different versions of the runtime system where more functionality is added in an incremental way. As a reference, we use a native version of the application that runs directly on top of Contiki (first row).

The overhead of the basic version of the runtime system (event delivery only) is due to memory copying (for the event payload) and the look-up in the local event subscription table. Then, timing monitoring introduces additional overhead mainly due to event timestamping and violation checks. Finally, more time is needed to store the timing information produced in the log – if the log is kept on external flash storage this becomes the most important cost component (by an order of magnitude compared to everything else). Although the overhead is substantial (even with logging in RAM) for a "null" application, it is quite small in absolute terms and its relative importance will drop significantly for applications that actually perform some processing.

6.3 DETECTION OF TIMING VIOLATIONS

To verify that our runtime system can indeed detect timing constraint violations, we use an application composed of a *Temperature Monitor* component that polls the temperature sensor every 1 second and issues *TEMP_EVENT* events containing the measured value, and an *Alarm Detector* component that handles them. The timing constraint of the respective event handler is set to 50 ms. Each component resides on a different TMote Sky node. In addition, a third node is used as an interferer which transmits dummy packets at a high rate. The *Network Manager* sends the event messages using Contiki's Rime stack reliable unicast primitive with four retransmissions in 250 ms intervals. The interferer uses the plain (unreliable) unicast primitive, with the dummy packets being addressed to a non-existent node.

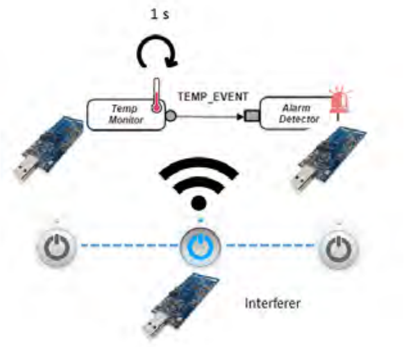


Figure 6-1: Experiment setup

The test scenario, depicted in Figure 6-1, has three phases: (i) the two nodes that host the application's components boot up in isolation for 50 seconds; (ii) the interferer node is turned on and injects noise for 50 seconds; (iii) the interferer is turned off, and the other two nodes continue their execution for another 50 seconds. Figure 6-2 plots the recorded response time for the $TEMP\_EVENT$ handler of the *Alarm Detector* component on the second node. As can be seen, timing violations are detected/reported when the interferer is turned on. By inspecting the logs we confirmed that during the test run both T_{src} and T_{dst} were around 0.7 ms and at no point exceeded 0.9 ms, while $T_{process}$ was constant at 0.06 ms. What actually varies and results in the sharp rise of the end-to-end response time is $T_{transfer}$, due to packet loss and the increased number of retransmissions required to deal with it at the network layer.

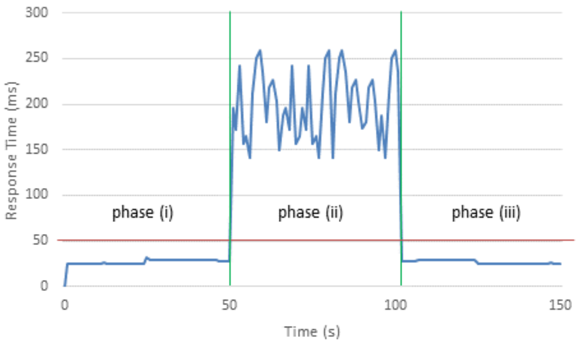


Figure 6-2: Application response time during the test run.

7

RELATED WORK

In this Chapter we briefly discuss indicative application composition models and techniques for the specification and monitoring of timing constraints.

7.1 APPLICATION COMPOSITION MECHANISMS

Component-based programming: In component-based programming, an application is composed by a number of components *wired* together through well-defined interfaces. Each component can be seen as a black-box that has a set of *provided* interfaces, representing the functionality it offers, and *required* interfaces specifying dependencies on functionality offered by others. In the area of WSNs a lot of work has been done towards this direction, and the proposed models span from operating system libraries to middleware solutions.

Typical examples of operating system libraries are nesC [14] and OpenCom [15]. NesC, which supports the TinyOS [16] operating system, provides a static component model where at compile-time an application composition is optimized and compiled into a monolithic image. On the contrary, OpenCom, which supports the Lorien [17] WSN OS, is a dynamic model where each component remains independent throughout the application lifecycle. Components can be instantiated/destroyed and connected/disconnected at run-time. Furthermore, components can be completely and independently unloaded from the system image at runtime, or other components loaded and integrated into the running system image.

On the other hand, Remora [18] and RUNES [19] are two well known middleware solutions focused in the run-time reconfiguration. In Remora, components and application compositions are described in XML. For each component except from the provided and required interfaces the produced and consumed events and properties are specified, with the latter being used for the parametrization of components at run-time. While Remora targets only on application-level programming, RUNES, which is a branch of OpenCom, provides a middleware kernel API that can be used to build compositions of both application- and middleware-level components.

Although component models provide a powerful abstraction for composing applications from generic and reusable building blocks, in general they focus in the node-level programming and do not support distributed relationships between components. In case such relationships must exist it is developer's responsibility to implement the right distribution mechanisms. For instance, in RUNES associations between remote components can be realized through Component Frameworks (CFs) providing interaction services. Therefore, our work is orthogonal to these models, as we target distributed application scenarios.

Coordination programming: Coordination is a programming paradigm whose goal is to separate the definition of the individual behavior of the application components from the mechanics of their interaction. This is usually achieved using message passing or data sharing as a model of interaction. Publish/subscribe is an example of the message passing model where a subscriber has the ability to express its interest in an event, or a pattern of events, in order to be notified subsequently and coordination occurs only through the exchange of messages (events) among publishers and subscribers. On the other hand, tuple spaces are an example of data sharing where communication takes place through the insertion/removal of tuples, into/from the tuple space which is an abstract of shared memory.

Mires [20], ESCAPE [21] and TinyCOPS [22] are publish/subscribe WSN middlewares built on top of the static component model provided by TinyOS, while the Loosely-Coupled Component Infrastructure (LooCI) [23] offers a dynamic component model. Mires follows a centralized approach where the subscriptions are driven by and notification are forwarded to the end-user through the sink node whereas the other solutions let remote components interact directly with each other. In addition, Mires and LooCI use a simple topic-based naming scheme where the publishers transmit all data referred to the subscribed topics while the other two allow more expressiveness through content-based filtering. In TinyCOPS this is achieved by augmenting the subscriptions API with constraints over the content of the published data while ESCAPE supports it by means of policies which are separated by the components implementation.

Hood [24] and TeenyLime [25] on the other hand, are middlewares that revolve around the notion of data sharing within the scope of a local neighborhood in WSNs. In Hood the elements that a node can share are defined through its attributes and local updates are reflected to the neighbors through periodic broadcasting and filtering on the receiver's side, while in TeenyLime, which is based on the tuple spaces paradigm, the tuple spaces are distributed among the devices and data are transiently shared with the one-hop neighbors.

Both message passing and data sharing paradigms allow high decoupling in space and time among event source nodes and event handling nodes that fits well the inherent event-driven communication model of WSNs. With respect to the aforementioned solutions, our design approach and programming model has similarities mainly with LooCI which, however, targets more resource-rich wireless devices. In LooCI associations between components are specified before deployment or established at run-time through explicit requests from the back-end. Differently, in our runtime associations between components are not programmer's nor network administrator's concern, but are formed and refreshed dynamically through periodic advertisements of the local subscriptions.

7.2 TIMING CONSTRAINTS SPECIFICATION AND RUN-TIME MONITORING

The specification and validation of timing constraints is the process of describing the timing requirements of specific operations in an application and checking them *before* or *after* the application deployment.

Pre-deployment methods: During development *static* and *dynamic* timing analysis techniques can be used to obtain estimates of the execution time and build correct-by-design systems.

Static methods are typically performed by combining some version of the code (source or object) with a model of the hardware architecture. More specifically, through program path analysis they derive feasible execution paths while through the system modelling they get the corresponding instruction timing information. Subsequently, the worst-case execution time (WCET) is estimated as the program path with the maximum cost. Such a method is described in [26] while a popular static analysis tool for embedded systems is Chronos [27].

Dynamic methods are based on measurements retrieved by executing the code on a given hardware or simulator for some set of test inputs. MSPSim [9] and WSim [28] are cycle-accurate WSN simulators that provide profiling and monitoring tools and can emulate complete MSP430-based platforms. Respectively, Avrora [29] and ATEMU [30] provide this functionality for the AVR microcontrollers. On the other hand, measurements on the hardware can be retrieved in the node-level using in-circuit programmers, emulators and debuggers, while in the network-level testbeds, large testing infrastructures, offer the ability to experiment over true wireless topologies. Notable WSN testbed are SmartSantader [31] and Wisebed [32], which offers the ability to combine real nodes with simulated and emulated ones.

In addition, real-time operating systems (RTOS), where the correctness of the system depends not only on the logical results of computation but also on the time at which these results are produced, advanced scheduling algorithms such as rate-monotonic and fixed-priority preemptive scheduling are employed in order to guarantee that deadlines are met. Example RTOS for WSNs are uC/OS-II [33], FreeRTOS [34] and Nano-RK [35] which adapts the Resource Kernel paradigm [36] in sensor networks and provides guarantees through static resource reservations based on offline estimates of CPU time, packet rates, and sensor sampling intervals used by the application tasks.

Our work is complementary to all of these techniques that help in the timing characterization of the applications. In fact, it is intended to be used in conjunction with them in order to specify the timing properties taking into account the overhead of the run-time monitoring.

Run-time monitoring: Monitoring of timing constraints has been widely studied and several approaches have been proposed to detect event occurrences on the target system and gather the respective timing information, either for post-processing or for run-time verification.

Hardware and hybrid monitoring approaches [37] [38] [39], utilize dedicated hardware devices that snoop the target system bus to detect event occurrences and gather the respective timing information. While they offer non-intrusiveness to the target system, they are costly to implement and rather inflexible as the dedicated hardware depends on the target system.

On the contrary, software monitoring approaches offer more flexibility and portability at the cost of intrusiveness (prode effects). The ART Real-Time Monitor [40] is a software monitoring tool focused on the visualization of the timing behavior of the system processes in distributed real-time systems. To this end, in the target system the corresponding state-changing events are recorded and subsequently reported to a remote host for further processing. A different approach is followed by Jahanian et al. in [41] where they present a method that utilizes a constraint graph based algorithm for detecting violations of user-defined timing assertions at the earliest possible time in distributed real-time systems.

In the area of WSNs there are several works focused in the post-deployment monitoring of the network. SNMS [42] is a network management system which on the one hand provides continuous user-driven monitoring of (local) application attributes through a query system and on the other hand offers program-driven notification of one-time events through an event logging system. PDA [43] allows the user to insert distributed assertions in the form of Boolean expressions over local and remote node attributes in the source code. In each node whenever a local attribute changes value or an assertion is executed respective messages tagged with a global time stamp are published and passively collected for evaluation in the back-end. PD2 [44] intends to help users identify/locate the sources of performance degradation in distributed WSN applications. To achieve this, it models the data flows generated by the application as causal paths in a graph of software components and relates poor application performance to significant data losses or latencies of some data flows as they traverse the software components on individual nodes and through the network.

With SNMS and PDA developers and system administrators can debug and monitor a WSN at run-time but none of them provide mechanisms for the detection of timing constraints violations. PD2 could provide such information but its monitoring operation is triggered only in case poor performance is exhibited. Thus, it is not appropriate for the validation of timing constraints, which should be a continuous process.

More close to our work is Breadcrumbs [45] which offers similar functionality for detecting violations of timing constraints at runtime. In Breadcrumbs events refer to state changes in the system, and the constraints specification is related to event flows, defined by an initial and a final event. Before deployment all possible paths of event flows are analyzed and validation checking code is inserted. At run-time, when a violation is detected, in the final event's handling method, the user gets alerted and can query the network to retrieve the actual path of the event flow along with the time consumption of each participating software module. However, if the earliest possible detection of a violation is desirable, the event flow should be divided to smaller pieces. Compared to our work, it is less flexible since the modification of a single timing constraint requires recompilation the application programs; a fact that could significantly slowdown the testing process in large-sized programs.

8

CONCLUSION

In this thesis we presented a programming model and runtime system for (a) structuring a distributed WSAN application in terms of components that interact via events, (b) specifying constraints regarding the respective event response times, and (c) checking these constraints during execution. By inspecting the information produced, it is possible to see not only whether there are any timing violations but also their breakdown along the entire end-to-end path.

Several enhancements could greatly augment the current solution's functionality. First of all, we are considering to prioritize the event forwarding/delivery operations based on the timing specifications. This way we could support, to some extent, QoS to the applications. We also think of including report analysis functionality to the back-end making the debugging procedure more automated.

Finally, since our solution targets both application developers and system administrators, we wish to evaluate our system and get feedback from real users. Towards this direction we have already ported Contiki, along our runtime, to a commercial-strength WSN platform [46] and we plan to make extensive evaluations in the field.

BIBLIOGRAPHY

[1] M. Koutsoubelias, S. Lalis, N. Grigoropoulos, P. Lampsas, S. Katsikas, and D. Dimas. “Scriptable Virtual Onboard Sensors for Conducting Post- Deployment Drills in Wireless Sensor Networks”. In: *IEEE Sensors Applications Symposium, 2015.* IEEE, 2015.

[2] A. Dunkels, B. Gronvall, and T. Voigt. “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”. In: *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*. Tampa, Florida, USA, Nov. 2004.

[3] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. “Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems”. In: *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*. Boulder, Colorado, USA, Nov. 2006.

[4] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. “Enabling Large-Scale Storage in Sensor Networks with the Coffee File System”. In: *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*. San Francisco, USA, Apr. 2009.

[5] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller. “Connecting Wireless Sensornets with TCP/IP Networks”. In: *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*. Frankfurt (Oder), Germany, Feb. 2004.

[6] M. Durvy, J. Abeillé, P. Wetterwald, C. O’Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. “Making Sensor Networks IPv6 Ready”. In: *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session.* Raleigh, North Carolina, USA, Nov. 2008.

[7] A. Dunkels, F. Osterlind, and Z. He. “An Adaptive Communication Architecture for Wireless Sensor Networks”. In: *Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*. Sydney, Australia, Nov. 2007.

[8] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. “Cross-Level Sensor Network Simulation with COOJA”. In: *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*. Tampa, Florida, USA, Nov. 2006.

[9] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, T. Voigt, and N. Tsiftes. “Demo abstract: MSPsim - an extensible simulator for MSP430-equipped sensor boards”. In: *Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN 2008).* Bologna, Italy, Jan. 2008.

[10] J. Polastre, R. Szewczyk, and D. Culler. “Telos: enabling ultra-low power wireless research”. In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*

[11] C. Perkins and E. Royer. “Ad-hoc on-demand distance vector routing”. In: *Proceedings WMCSA ’99. Second IEEE Workshop on Mobile Computing Systems and Applications.* IEEE, 1999.

[12] *ZigBee Alliance.* `www.zigbee.org`. Accessed: 2015-07-14.

[13] F. Cristian. “Probabilistic clock synchronization”. In: *Distrib Comput* 3.3 (Sept. 1989), pp. 146–158.

[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. “The nesC language”. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI ’03.* ACM Press, 2003.

[15] G. Coulson, G. S. Blair, P. Grace, F. Taani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. “A generic component model for building systems software”. In: *ACM Trans. Comput. Syst.* 26.1 (2008).

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. “System architecture directions for networked sensors”. In: *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems - ASPLOS-IX.* ACM Press, 2000.

[17] B. Porter and G. Coulson. “Lorien: a pure dynamic component-based operating system for wireless sensor networks”. In: *Proceedings of the Fourth International Workshop on Middleware for Sensor Networks, MidSens 2009, December 1, 2009, Champaign, IL, USA, Co-located with Middleware 2009.* 2009, pp. 7–12.

[18] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. L. Trung, and F. Eliassen. “Programming Sensor Networks Using Remora Component Model”. In: *Distributed Computing in Sensor Systems, 6th IEEE International Conference, DCOSS 2010, Santa Barbara, CA, USA, June 21-23, 2010. Proceedings.* 2010, pp. 45–62.

[19] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. “The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario”. In: *Fifth Annual IEEE International Conference on Pervasive Computing*

and Communications (PerCom 2007), 19-23 March 2007, White Plains, New York, USA. 2007, pp. 69–78.

[20] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. S. Rosa, C. A. G. Ferraz, and J. Kelner. “Mires: a publish/subscribe middleware for sensor networks”. In: *Personal and Ubiquitous Computing* 10.1 (2006), pp. 37–44.

[21] G. Russello, L. Mostarda, and N. Dulay. “A policy-based publish/subscribe middleware for sense-and-react applications”. In: *Journal of Systems and Software* 84.4 (Apr. 2011), pp. 638–654.

[22] J.-H. Hauer, V. Handziski, A. Köpke, A. Willig, and A. Wolisz. “A Component Framework for Content-Based Publish/Subscribe in Sensor Networks”. In: *Wireless Sensor Networks*. Springer Science + Business Media, 2008, pp. 369–385.

[23] D. Hughes, K. Thoelen, J. Maerien, N. Matthys, W. Horré, J. Cid, C. Huygens, S. Michiels, and W. Joosen. “LooCI: The Loosely-coupled Component Infrastructure”. In: *11th IEEE International Symposium on Network Computing and Applications, NCA 2012, Cambridge, MA, USA, August 23-25, 2012*. 2012, pp. 236–243.

[24] K. Whitehouse, C. Sharp, D. E. Culler, and E. A. Brewer. “Hood: A Neighborhood Abstraction for Sensor Networks”. In: *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services, MobiSys 2004, Hyatt Harborside, Boston, Massachusetts, USA, June 6-9, 2004*. 2004.

[25] P. Costa, L. Mottola, A. Murphy, and G. Picco. “TeenyLIME: transiently shared tuple space middleware for wireless sensor networks”. In: *Proceedings of the First International Workshop on Middleware for Sensor Networks, MidSens 2006, November 28, 2006, Melbourne, Australia, Co-located with Middleware 2006*. 2006, pp. 43–48.

[26] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm. “Static Timing Analysis for Hard Real-Time Systems”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer Science + Business Media, 2010, pp. 3–22.

[27] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. “Chronos: A timing analyzer for embedded software”. In: *Science of Computer Programming* 69.1-3 (Dec. 2007), pp. 56–67.

[28] A. Fraboulet, G. Chelius, and E. Fleury. “Worldsens: Development and Prototyping Tools for Application Specific Wireless Sensors Networks”. In: *2007 6th International Symposium on Information Processing in Sensor Networks*. IEEE, Apr. 2007.

[29] B. Titzer, D. Lee, and J. Palsberg. “Avrora: scalable sensor network simulation with precise timing”. In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IEEE, 2005.

[30] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. Baras, and M. Karir. “ATEMU: a fine-grained sensor network simulator”. In: *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.* IEEE, 2004.

[31] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and D. Pfisterer. “SmartSantander: IoT experimentation over a smart city testbed”. In: *Computer Networks* 61 (Mar. 2014), pp. 217–238.

[32] G. Coulson et al. “Flexible experimentation in wireless sensor networks”. In: *Commun. ACM* 55.1 (Jan. 2012), p. 82.

[33] J. Labrosse. *MicroC/OS-II : the real-time kernel.* Lawrence, Kan: CMP Books, 2002. ISBN: 1578201039.

[34] *FreeRTOS.org project.* http://www.freertos.org. Accessed: 2015-06-01.

[35] A. Eswaran, A. Rowe, and R. Rajkumar. “Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks”. In: *26th IEEE International Real-Time Systems Symposium (RTSS ’05).* IEEE, 2005.

[36] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. “Resource kernels: A resource-centric approach to real-time and multimedia systems”. In: *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking.* 1998, pp. 150–164.

[37] J. Tsai, K.-Y. Fang, and H.-Y. Chen. “A noninvasive architecture to monitor real-time distributed systems”. In: *Computer* 23.3 (Mar. 1990), pp. 11–23.

[38] D. Haban and D. Wybranietz. “A hybrid monitor for behavior and performance analysis of distributed systems”. In: *IIEEE Trans. Software Eng.* 16.2 (1990), pp. 197–211.

[39] S. Ricardo and J. de Almeida. “Run-time monitoring for dependable systems: an approach and a case study”. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* IEEE, 2004.

[40] H. Tokuda, M. Kotera, and C. E. Mercer. “A real-time monitor for a distributed real-time operating system”. In: *ACM SIGPLAN Notices* 24.1 (Jan. 1989), pp. 68–77.

[41] F. Jahanian, R. Rajkumar, and S. C. V. Raju. “Runtime monitoring of timing constraints in distributed real-time systems”. In: *Real-time Systems* 7.3 (Nov. 1994), pp. 247–273.

[42] G. Tolle and D. Culler. “Design of an application-cooperative management system for wireless sensor networks”. In: *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.* IEEE, 2005.

[43] K. Römer and M. Ringwald. “Increasing the visibility of sensor networks with passive distributed assertions”. In: *Proceedings of the workshop on Real-world wireless sensor networks - REAL-WSN ’08.* ACM Press, 2008.

[44] Z. Chen and K. G. Shin. “Post-Deployment Performance Debugging in Wireless Sensor Networks”. In: *2009 30th IEEE Real-Time Systems Symposium.* IEEE, Dec. 2009.

[45] H. Kim, S. Yi, W. Jung, and H. Cha. “A Decentralized Approach for Monitoring Timing Constraints of Event Flows”. In: *2010 31st IEEE Real-Time Systems Symposium.* IEEE, Nov. 2010.

[46] *Prisma Sense RMS (Remote Monitoring Systems).* `http://www.prismaelectronics.eu/site/index.php/en/solutions/prismasense`. Accessed: 2015-07-14.