Adaptation of On-line Scheduling Strategies for Sensor Network Platforms

Christian Decker, Till Riedel, Michael Beigl
Telecooperation Office (TecO),
University of Karlsruhe
Karlsruhe, Germany
{cdecker,riedel,beigl}@teco.edu

***Abstract*— Current sensor network platforms perform multiple processes including sensor sampling, communication, and various computational tasks. When deployed in unpredictable environments, complex schedules of those processes may arise. Typical sensor network qualities like periodic sampling of sensors, avoidance of starvation of single processes and automatic energy management are crucial and required to be maintained in such situations. We propose a scheduling framework for senor nodes consisting of a scheduler, a dispatcher and a controller for an adaptation of process execution during the runtime. The key components of our framework are a controller and an enhanced dispatcher which both implement various strategies to maintain crucial qualities of process execution on sensor nodes deployed in unpredictable environments. Further, the framework is aware of the energy consumption of sensors. We show that our controlled scheduling framework performs significantly better than a non-controlled single scheduler in unpredictable environments. Our proposed measures are efficient to implement. Results are underpinned by extensive simulations and a first implementation on our Particle Computer platform.**

Keywords: Scheduling, Sensor Network, Energy Management, Particle Computer

I. INTRODUCTION

Sensor node platforms are proposed for various monitoring and tracking tasks which are hard to accomplished by other technologies. Thereby, sensor nodes are deployed in unknown and therefore in unpredictable environment. As a consequence, the nodes should implement mechanism to adapt their tasks to the environment. Microcontrollers on current sensor nodes platforms have to handle a large diversity of processes. Among them are sensor sampling tasks, communication tasks and computational task. Additionally, there are background processes for maintenance. Within each of those groups further dimensions are revealed. For instance, sensor sampling tasks may have different properties regarding sampling interval and sampling time. Recently, lots of operating systems for sensor nodes have been proposed as an underlying runtime system handling various tasks. A prominent example is TinyOS[6], the operating system for the Berkeley Motes. In this paper we want to approach three goals, which we identified as crucial for process execution on sensor nodes in unpredictable environments. Our first goal is to achieve a *low jitter in periodic sensor sampling processes*. Sampling in fixed intervals is reasonable as many algorithms rely on this property. But, with many competing sampling processes in an unpredictable environment high load and overload situations may occur. As a consequence, the runtime system has to adapt the task properties to fulfill such requests.

Our second goal is the *avoidance of task starvation*. The nodes are deployed in areas where no external administration or debugging is possible. Therefore, the system should take measures, if a task is constantly left out. In particular, such a situation may occur, if other tasks are permanently considered to be more important because of an event detection. Our third goal is the *automatic energy management* of sensors on the platforms. Sensors have start-up times, shutdown times or sample delays. This may lead to complex dependencies which need to be resolved by the runtime system.

These goals can be approached by an appropriate scheduling strategy. However, in unpredictable environment, the task properties are unpredictable, too. For instance, computation times of tasks are varying, overload situations may occur due to event detection and processing or computational tasks may delay the entire system because their runtime depends strongly on their input data. This unpredictability of the environment makes it necessary to adapt the scheduler online while the system is running. Furthermore, the restricted resources of the sensor node platforms make it rather hard to implement a complex strategy. Therefore, we propose an adaptive scheduling framework utilizing compact priority-based schedulers which are further supported by an enhanced dispatcher and a controller.

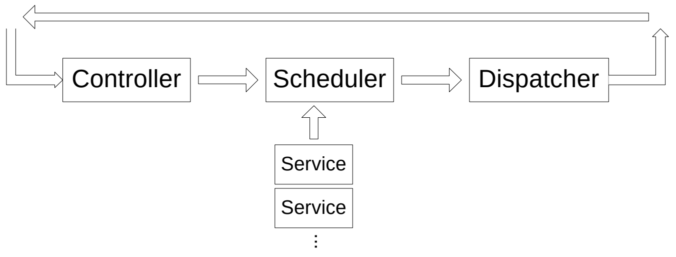

Figure 1. The adaptive scheduling framework

The key component of our framework is the controller and an enhanced dispatcher implementing a set of strategies to handle the dynamic effects of the system. Tasks are encapsulated in services representing stand-alone functional entities. The dispatcher monitors the performance of the execution of the services and triggers the controller for adaptation if necessary. Our contribution in this paper is the quantitative performance evaluation of online scheduling adaptation strategies which are suited for resource constrained sensor network platforms applied in unpredictable environments. Our approach is based on permanent feedback from the current process execution. Remarkable for this approach is that it results in a very good performance with a minimum of previously defined and uncertain knowledge.

The remainder is structured as follows: In section II we will review scheduling strategies on sensor nodes. As a result, we will derive important requirements for our scheduling framework. The framework works on functional entities called services, which will be introduced in section III. Section IV then presents the details of adaptive scheduling framework and the adaptation strategies. For the evaluation of the framework we will describe a representative set of services in section

V and the results of comparisons with non-adaptive strategies are discussed on section VI. Section VII shows an implementation of the adaptive framework on our Particle Computer platform. We conclude the paper in section VIII.

II. Scheduling Strategies on Sensor Nodes

In this section we will review important scheduling algorithms, which are currently implemented on sensor nodes. All algorithms utilize the same task model. A task i is defined as a tuple $task_i = \{P_i, C_i, T_i, d_i, r_i\}$, whereas P_i is the function which is executed, C_i is the computation time, T_i is the period of the execution, d_i is the deadline and r_i identifies the resources required by the task. The recurred execution of a task i is said to be a set of jobs $J_i = \{j_{i0}, j_{i1}, ...\}$. A job j_{i0} is scheduled to start its execution at the time $a(j_{i0}) = a_{i0}$, which is called the arrival time. An ordered set of jobs according to some policy is called a schedule. If a job gets interrupted during its execution, the schedule is called preemptive. If all jobs runs to completion before the next one is started, the schedule is called non-preemptive.

A. First-In-First-Out (FIFO)

A FIFO scheduler executes tasks in the order of their arrival. As a consequence, each job has a priority proportional to its arrival time and the resulting schedule is a totally ordered list of those priorities. A FIFO strategy runs non-preemptively. As a result, the implementation only requires an array of jobs and does not need any a-priori knowledge about the jobs. These qualities constitute the FIFO strategy as a preferred approach for resource constrained devices. TinyOS 1.x [6], the operating system of the Berkeley Motes, implements a FIFO scheduler. However, the scheduler cannot assert time guarantees of periodic jobs nor execution guarantees. These are serious drawbacks of this approach. If the job queue is exceeded, then newly arrived jobs must be canceled. As a result, events may get lost and tasks run into starvation since constantly upcoming events may avoid the queuing of the starving task.

B. Rate Monotonic Scheduling (RMS)

RMS is a fix priority strategy where once a priority is assigned to a task depending on its period P_i. Tasks with shorter periods are assigned higher priorities than tasks with longer periods. In [9] the authors showed that RMS is optimal, i.e. that no other fixed-priority algorithm can schedule a task set that cannot be scheduled by RMS. The authors also derived a least upper utilization bound for a set of n periodic tasks in a preemptive schedule. If the utilization

$$U = \sum_i \frac{C_i}{T_i} \le n(2^{1/2} - 1), \qquad (1)$$

then the RMS guarantees that all deadlines will hold. For a non-preemptive schedule this guarantee can only be given, if the following condition holds:

$$\sum_i C_i \le \min T_i$$

As a result, RMS executes the jobs repeatable in the same order and achieves even very low jitter during the periodic execution of the tasks. Fixed-priority strategies are attractive to be implemented on sensor nodes, since they need to compute the schedule only once before the runtime. They can even guarantee periodic behavior of tasks and the implementation just requires a single array where the jobs rotate through. The major drawback is the inflexibility of RMS. The strategy does not consider varying execution times and cannot adapt its runtime behavior. Further, an admission based on (1) under-utilizes the system constantly and can result in starvation of tasks, which do not satisfy the criteria.

C. Earliest Deadline First (EDF)

In [9] Lui and Layland investigated EDF. The next job is selected according to its deadline during the runtime of the system. This behavior can handle system dynamics where new jobs arrive unpredictably in the system. This property of EDF guarantees real-time behavior even in the case of unpredictable event occurrence on sensor nodes. The EDF strategy is preferred for real-time systems, because it utilizes the processor at best. Further, EDF is optimal in the sense, that if there is a schedule for which all deadlines will hold, then EDF will find it. A task set of n periodic tasks is guaranteed to be schedulable and therefore during execution all deadlines will hold, iff the utilization U is

$$U = \sum_i \frac{C_i}{T_i} \le 1 \qquad (2)$$

However, (2) holds only for preemptive schedules. In the case of non-preemptive schedules, a complete search according to Bartley's algorithm [2] has to be undertaken to find a feasible schedule. A variation is the Spring algorithm [11] which utilizes additional heuristics.

AmbientRT, the operating system for the µNode platform [7], implements an EDF variation called Earliest Deadline First with Deadline Inheritance (EDFI). This is a preemptive strategy using a priority ceiling mechanism applied on resource usage in order to solve conflicts when resources are shared between different tasks. In order to run correctly, EDFI requires exact computation times and resource allocation of each task beforehand. Instead of the exact computation time, a Worst-Case-Execution-Time (WCET) can be specified. WCET is an upper bound, which will never be exceeded. The properties specifications are left to the developer of an application on µNodes and therefore a potential source of errors when wrongly estimated. The EDF strategy is fragile to a domino effect, which may results in a continuously missed deadlines, once a single deadline was missed. In particular, computational tasks underlie a large variations as their computation times are dominated by the input data. Nevertheless, if these parameters are correctly chosen, then EDFI guarantees deadlock-free execution and real-time behavior. But, even if the task set is static and the admission has verified the schedulability, i.e. all deadlines will hold, EDF may not guarantee fix intervals between two jobs of the same task. As a consequence, low jitter during periodic sampling cannot be guaranteed.

D. Cooperative Scheduling

In cooperative scheduling, a task yields its execution and the control is given to a dispatcher for selecting the next one. As a consequence, the process order is implicitly encoded in the application program. Operating system like SOS[5] for Motes and the highly-portable operating system Contiki [4], which is also available for sensor node devices, implement cooperative scheduling. In particular, latter system supports this strategy through a very lightweight implementation called protothreads. The overhead is comparable to the FIFO scheduling strategy because scheduling decisions are primarily specified by the application or the developer. There is basically additional effort through a specific scheduler. Cooperative scheduling burdens the effort on the developer. An implementation of periodic execution of tasks must be implicitly encoded in the application logic. For achieving fix periodic intervals the developer need to constantly monitor the execution behavior of the entire application. In case of a modification of the application, the measures to achieve a certain processing behavior also have to be modified. These might be distributed across all tasks of an application. If applications get more complex the burden on the developer increases. As a result, it arises the risk, that decisions where to place yields within a task and which task to select next, may lead to task starvation. Since the task switches are distributed across an application, the possibilities of different execution orders increase dramatically. A developer has to carefully study the entire application in order to remain an intended behavior.

E. Résumé

From the previous analysis we derive now important properties of our scheduling framework for sensor node platforms. Preemptive real-time schedulers guarantee deadlines for job execution and utilize the processor at best. However, additional effort is required to resolve conflicts due to resource sharing when a job accesses a resource, while a previously started job is preempted. Preemption is difficult to implement safely on microcontroller systems with no protection between the tasks. Usually, a timer interrupt is used to preempt a running task. But the authors of [10] showed, it impossible to guarantee that a task mustn't switch off this interrupt. The memory consumption of the switching between jobs when they are preempted is not negligible. A job has to store the current process control block consisting of actual processor registers and a stack pointer in order to return to its own block of local variables on the stack. AmbientRT reports an additional overhead of 10 bytes per task while 72 tasks are allowed at maximum. Although this is a low overhead per task, it scales up with the number of tasks and allocates more memory. Since the memory layout is often organized statically and fixed during compile time of an application, the memory is permanently reserved for the scheduler. Schedulers supporting real-time guarantees require an accurate specification of task properties, especially of the computation time. These are hard to find, especially, when they depend on the input data of a task. WCET approximations on the other side result in underutilization of the system.

FIFO strategies and fixed-priority strategies such as RMS are very inflexible. Although attractive for resource constrained sensor nodes due to their low effort, periodic processes with low jitter and avoidance of task starvation are hard to support reliably. Automatic managing of the energy consumption of sensors, i.e. switching them on and off with respect to their specific start-up times is not considered at all.

Due to the restricted resources and the additional overhead required for preemptive tasks, our framework will implement non-preemptive tasks. As a result, resource conflicts cannot occur and this avoids any measures for synchronization. Unpredictable event processing may disturb real-time properties. Our framework omits real-time guarantees, but rather prefers low jitter during periodic executions of tasks. Periods of tasks are adaptable in order to achieve this requirement. Cooperative scheduling is certainly best suited for specific parts of an application on sensor nodes, but it has no view on the overall execution performance. Furthermore, it is complex and burdens a developer due to the tight coupling of the application and the scheduling decisions applied on the tasks. Finally, our framework allows flexible annotations. For instance, sensor tasks are annotated by their start-up times, in order to make the system aware of the sensors' energy management.

III. SERVICES

As a consequence of our analysis in section II, we introduce the concept of services, which reflects the results from in section II.E. Services represent a uniform abstraction of all recurring processes on a sensor node. A schematic view on a service is depicted in Figure 2. Central in this abstraction is the service function, which implements the functionality, e.g. sensor sampling. Services run non-preemptively and are independent from each other, i.e. there is no service, which calls another one. The start of a service execution is always driven by the underlying runtime system. A service execution is always periodic, non-preemptive and an execution cycle finishes with a result stored in a result buffer. The result set is allowed to be empty. As an abstraction of recurring processes, a service provides rich capabilities to be configured. Among them are regular task properties like period, deadline and computation time, but also additional parameters like starvation level, and start-up time for sensor sampling services. The runtime system utilizes these parameters for its online decisions.

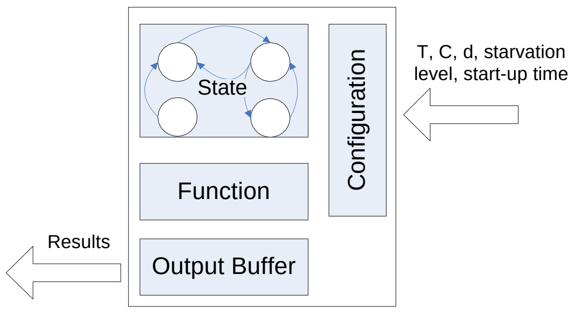

Figure 2. A schematic view on a service – an encapsulated independent functional entity for sensor nodes

During the periodic execution, a service transits through several states which are depicted in detailed in Figure 3. During the boot-up of the system, all services are initialized and to the sleeping state. Since all services arrive at the same time in the runtime system, they are now turned on one after the other, i.e. according to their start-up parameter, their new arrival time of each service is computed and the service is placed in the waiting state. If the arrival time is reached for a waiting service it changes its state to ready. All ready services are then executed by the dispatcher and they automatically transit to the waiting state. A separate execution state is not necessary because services run non-preemptively. If a period is greater than the start-up time of a service, the service may want to go to the sleeping state in order to save energy. This is a crucial feature for sensor sampling services which can power-down energy-consuming sensors until the next usage.

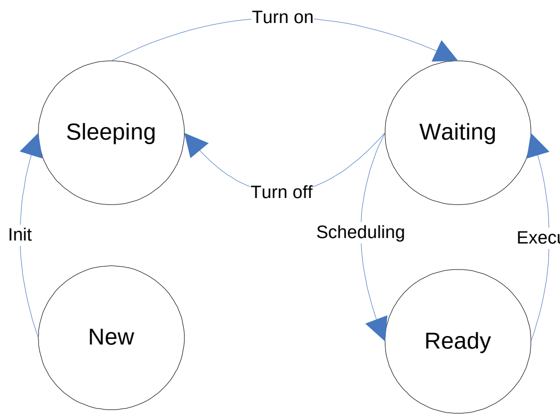

Figure 3. State transition diagram of a service

The service abstraction does not prohibit event processing. Events occur sporadically. In [8] the authors suggest a method to model sporadic processes with periodic processes. Thereby, the minimum time difference between two sporadic events can be considered as the period of a periodic process. As a consequence, the introduction of services can handle sporadic event processing.

Formally, services are similar to the notion of tasks as presented in section II. We describe the service S_i as a tuple $S_i = \{P_i, C_i, T_i\}$, where P_i is the service function which is executed, C_i is the computation time, and T_i is the period of the execution. The resource description can be omitted because non-preemptive services do not interfere with each other on resources when accessing them. However, conflicts occurring as a result of sharing resources between different service invocations, are determined by the application logic and must be handled by utilizing internal states of resources. In the service model, the computation time C_i is given (i.e. specified by the developer), but for a concrete execution instance it is unknown. In particular, C_i of an instance is highly dynamically and may strongly vary according to the data which are processed. This behavior is the reason for omitting the deadline. Services are brought to execution as jobs at a given arrival time. The arrival time of the n-th job representing the n-th invocation of service S_i is defined as $a_{i,n} = (n-1)T_i$.

IV. Scheduling Framework and Runtime System

The scheduling framework for sensor node platforms is responsible for executing the services. It serves as an underlying runtime system for all services on a sensor node platform. The design goal was to plug-in multiple schedulers and adaptation strategies. As a consequence, the framework is divided in the three components: dispatcher, scheduler and controller. The framework operates on the notion of jobs, which are the runtime representation of services. It utilizes the jobs' arrival time and a reference from a job to its originating service. Jobs are organized in two queues, one for all ready jobs, which need to be executed now, and one queue for waiting jobs, which have their arrival time in the future. When designing the framework we followed a strict separation between the components' actions on the jobs in order to form a clear concept and to avoid side-effects between the components. An overview of the framework is depicted in Figure 4.

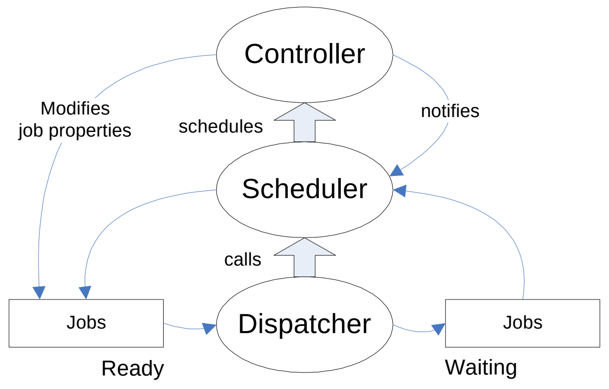

Figure 4. Adaptive scheduling framework

The core component is the dispatcher. It executes the service function associated with a job. The dispatcher is the only component which is allowed to remove jobs from the ready queue and to insert jobs in the waiting queue. If jobs achieve their arrival time, the scheduler moves them from the waiting queue to the ready queue in order to be executed by the dispatcher. The waiting queue is sorted according to the next arrival time of a job, while the ready queue is sorted according to priorities computed by the scheduler. The scheduler is the only component which inserts jobs in the ready queue and changes their ordering therein. The planning of the ordering is based on the service parameters, especially on the computation time and period. The priority based interface via the ready queue allows us to plug-in priority based schedulers, such as FIFO, RMS and EDF. Since schedulers describe a ordered execution plan, this interface is naturally generic to handle other schedulers as well. On top of the scheduler we introduce a controller. This component is responsible for schedule adaptation. In order to avoid faulty interference with the scheduler, the controller is only allowed to modify job properties. If it changes the priority of a job, it must notify the scheduler. In the next subsections we will explain the design of the framework components in more detail.

A. *Dispatcher*

The dispatcher is executed for each job. It has access to two queues – ready and waiting – which match the states of a service. Services in the sleeping and waiting state are both contained in the waiting queue. Jobs in the ready queue are ordered by their priority with the highest priority on the first position. The dispatcher retrieves the first job from the ready queue, executes it and inserts it in the waiting queue. Latter is organized ascending according to the next arrival times of the jobs. The figure below illustrates the separate actions of the dispatcher.

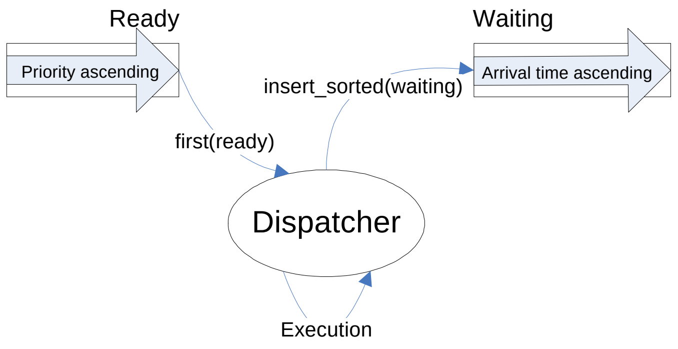

Figure 5. A slim dispatcher

The dispatcher has only the knowledge about the just occurred execution of a job. For computation of the job's next arrival time the dispatcher calls an appropriate function from the service because the dispatcher does not know the service's period. The result from this function is used to insert the job at the right position within the waiting queue. Since the ready queue is already sorted it takes a constant effort to retrieve the highest priority job. The insertion in the waiting queue uses a insertion sort algorithm with a complexity of O(n). Based on just two queues and a very low knowledge about each job we achieved a *slim design of the dispatcher*, but still maintaining flexibility for adaptation.

B. *Dispatcher adaptation strategies*

In the following subsections we present two adaptation strategies – automatic sensor energy management and jitter correction – which are directly implemented in the dispatcher.

1) *Energy management*

Our slim dispatcher design is enhanced for *automatic energy management*. The goal is to switch off sensor after the sampling job, but switch them on before the next sampling. This has to take the start-up time of the specific sensors into account. Here, the service plays an active role. The dispatcher is state neutral, i.e. the state is held within the service and modified by the service itself when the job is accessed by the dispatcher. As a result the service itself can schedule its on/off behavior by returning the appropriate arrival time depending on its state when asked by the dispatcher.

In Figure 6 the sensor sampling is just over when the dispatcher requests a new arrival time for the job. The associate service returns instead of its period T, the start-up difference $T - w$; W hereby denotes the configured start-up time specific for the sensor sampling service. After the job is executed at $T - w$ the service will then return W as the next arrival time. It is assumed that the job's computation time for switching the sensor on at $T - w$ is negligible.

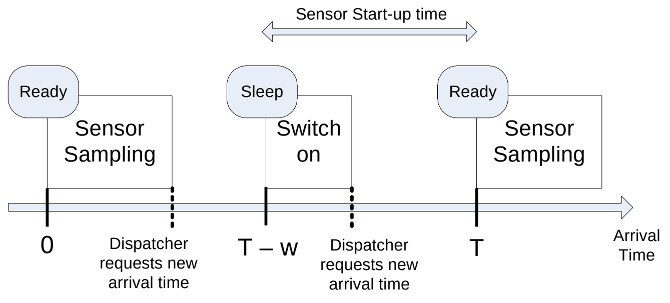

Figure 6. Automatic energy management

2) *Jitter correction*

The dispatcher is further enhanced to support the goal of low jitter in successive sampling processes. The jitter J is the delay between the arrival time and the time the job starts its execution. We call this start time dispatch time. Jitter occurs as a result of a job's unknown computation time. The scheduler has planned the execution order, but a job exceeds the planned computation time. As a consequence, it causes a jitter for the next job. The *jitter correction* works as follows: Instead of setting the next arrival time of a job on the period interval, the period is added to the dispatch time of the job. The new arrival time is then recursively defined as $a_{i,n} = a_{i,n-1} + J_i + T_i$. The advantage of this definition is that jobs with common arrival times are differently shifted. As a result of this adaptation, the next dispatch time is set closer to the real behavior of the service. Jitter affects the period of jobs. Addressing our goal of low jitter, the jitter correction guarantees that the period deviation is within $\pm J$. We show this in the following theorem.

Theorem 1: The jitter correction applied on the arrival times of jobs – $a_{i,n} = a_{i,n-1} + J_i + T_i$ – guarantees a worst-case period deviation within $\pm \max\{J_i\}$.

Proof: Firstly, we compute the period of a job using the difference of the dispatch times between the $(n-2)$th execution instance and the $(n-1)$th instance. We further assume, that the $(n-1)$th instance was delayed by a jitter J_i. Clearly, the difference is $a_{i,n-1} + J_i - a_{i,n-2} = T_i + J_i$. In the second step, we compute the difference between the nth execution instance and the $(n-1)$th instance. The $(n-1)$th instance started at $a_{i,n-1} = a_{i,n-1} + J_i$, because it was delayed, and the nth instance starts at $a_{i,n} = a_{i,n-1} + J_i + T_i$, because of the jitter correction. However, $a_{i,n}$ may experience an additional jitter $J_{i,2}$. The difference is $a_{i,n} - a_{i,n-1} = T_i + J_{i,2}$. The period deviation ΔT is now the difference of two consecutive periods of jobs: $\Delta T = J_i - J_{i,2} \le \max\{J_i\}$. The sequence of the jitter occurrence can be also the other way around, therefore the deviation is $\Delta T = \pm \max\{J_i\}$.

In the following example we illustrate the effect of jitter correction. We simulated jobs with period $T = 3$. The execution of jobs is randomly interfered with jitter between 0 and 1. One set of jobs was simulated without jitter correction and the other set has utilized the jitter correction as defined above. The figure Figure 7 plots the histogram of the resulting period deviations. As proofed above, the deviation stays between ± 1 for the set utilizing the correction. Furthermore, the deviation is primarily 0. For the set where the jitter is not corrected, the deviation stays between ± 2. Although the deviation stays primarily around 0, the deviation is broader distributed than in the case where correction is applied. Addressing our goal of low jitter, the jitter correction outperforms the non-corrected case by factor 1.4091 in the case for 0-deviation.

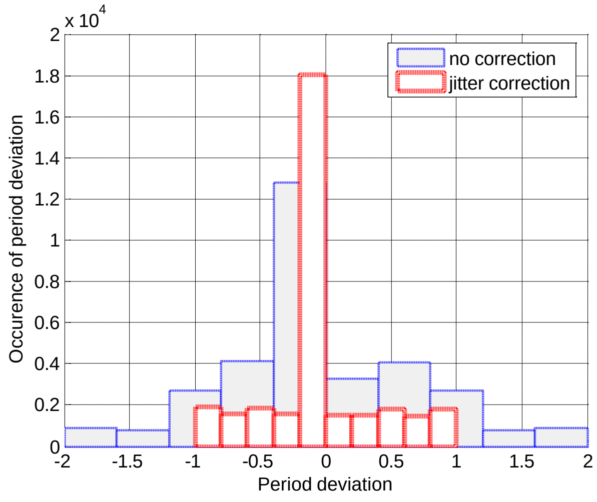

Figure 7. Period deviation for a maximum jitter = 1 and a period T=3

Although the jitter will be reduced, the period increases. The shift always moves the arrival time forward in time, which may result in less execution per time frame.

The dispatcher design is also able to *handle the domino effect by job omission*. If computation times of jobs vary strongly, they may cause an execution delay of a job which causes further jobs to delay their execution. This may add-up to a delay chain and violate constantly the periodicity of jobs. Before the execution of a job, the dispatcher may ask the associate service whether the job i already exceeded its next arrival time. If at the time t with $t > a_{i,n+1}$ the n-th execution of job i has not occurred, then the dispatcher can omit the execution. As a result, the delay chain is broken and the domino effect is stopped.

C. *Scheduler*

Our scheduling framework allows a clear way to plug-in different schedulers. For all jobs in the waiting queue which have achieved their arrival time, the scheduler computes an order of execution. This order is expressed as a priority used to order those jobs in the ready queue. Due to the cooperation between the scheduler and the dispatcher, we achieved that no service is represented by multiple jobs. From this perspective the framework does not impose additional overhead. As a result, the queues can be safely bounded to the maximum number of services the system designer wants to support. For our system tests we implemented the previously discussed strategies EDF, RMS and FIFO. For the reasons analyzed in section II.E we intentionally left out a cooperative scheduling strategy. None of the schedulers in the framework should perform an admission control. Our second central goal was to avoid process starvation and rather handle such situations online. Therefore, the schedulers should always accept all jobs.

D. *Controller*

A serious problem occurs, if jobs face starvation. This might be a result of a fix priority scheduling strategy or constant job omissions in order to stop a domino effect. Such problems are handled by controllers. Controllers are services as well, but the dispatcher directly calls them. This is required to guarantee their execution. By design they are required to only modify job properties. For handling the starvation problem we now introduce the starvation controller. This service observes the job queues and the past runtime behavior of the other services in order to bound the maximum number of job omission. Each service is annotated with a starvation level specifying this bound. The starvation level is set by the developer in advance and reflects his requirements on a minimum response time for a service. If the controller is executed it compares this value with the number of omissions, which occurred in series for this job. Such analyses are most complex compared to the effort of the scheduler and the dispatcher because for all jobs j_i in both queues the controller has to compute the number of omissions given by

$$omission_i = \left\lfloor \frac{t - t_{i,last}}{T_i} \right\rfloor,$$

where t denotes the current time, $t_{i,last}$ is the last time, that the job was successfully executed and T_i is the period of j_i. The starvation controller then follows a straight rule:

$$\forall i : \text{if } \left(l(s_i) - omission_i \le 0\right) \text{then } prio(j_i) = prio_{FIFO},$$

where $l(s_i)$ denotes the configured starvation level, i.e. the number of allowed omissions, $prio(j_i)$ is the priority of the job and $prio_{FIFO}$ is a reserved priority which causes the scheduler to place this job as the first one before all others in the ready queue. If there is more than one job, then they will be placed in FIFO order. Since FIFO is starvation free as long as the computation times are finite, the previously starved job will be guaranteed executed by the dispatcher. After execution the properties priority and starvation level are reset. The controller requires support from the dispatcher. Due to the complex computations it will run seldom. As a consequence, the dispatcher has to record the last successful execution for each job. Usually, the runtime system will utilize only one starvation controller for all jobs. The system performance can be improved, if the period of this controller can be determined in advanced. However, an appropriate controller period depends on the starvation level and the period of the service. For instance, the controller can run more seldom, if the starvation level is high, i.e. the developer allows a high number of omissions. The optimal controller period in order to hold all starvation bounds is the greatest common divisor (gcd) of all service periods multiplied by the smallest starvation level. However, as soon as one period is prime, the gcd is 1, which runs the controller as fast as the most frequent service. Checking all jobs in both queues as fast as a periodic sensor sampling service imposes too much load on the system. As a consequence, we focus here on the *starvation controller for services with the same period*. Although equal in their period, different services may have different starvation levels. For reasons of comparison, we define the omission ratio as the number of job omissions of a job i per time step. We now determine the least upper borderline of the controller period, where the omission ratio for at least one of the jobs differs from all the others. If the controller period exceeds this borderline, the omission ratio will be the same for all jobs. From the controller rule, we can derive the period guaranteeing the starvation bound. From $l(s_i) - omission_i \le 0$, we derive $T_{control} = l(s_i) \cdot T_i \le t - t_{i,last}$. For different starvation levels the upper bound is now found by $\max\{T_i \cdot l(s_i)\}$. Beyond this period, the controller always evaluates its control rule to true and notifies the scheduler to schedule all jobs in FIFO order. The Figure 8 illustrates this behavior for two services with period $T = 3$ and starvation level $l(s_1) = 1$ and $l(s_2) = 10$. The upper bound for the controller period is $\max\{T \cdot l(s_i)\} = 30$.

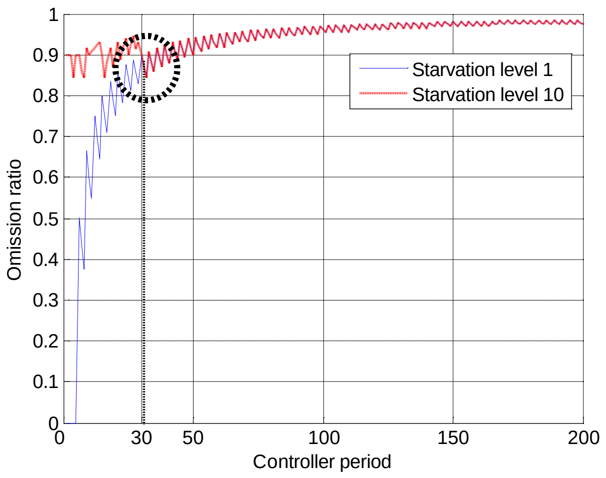

Figure 8. Controller period bound at $T_{control} = 30$ for two services with period T=3 and different starvation levels

In terms of system load and guarantees for starvation bounds, no optimal control period can be given. An alternative is to delegate the need of control to the dispatcher. However, the dispatcher then has to count the number of job omissions in order to call controller immediately. In contrast to our system design, the dispatcher then has to know every detail of a service and additionally it has to implement almost all of the controller logic. This stands in contrast to our design principles from section IV.

V. CASE STUDY

Our evaluation setting is the AwareOffice [1], an office space with multiple moveable and mobile objects such as chairs, tables, windows, doors and office supplies like pens, paper, projectors, whiteboards and flipcharts. Office workers interact with those objects regularly. We attached our Particle Computer [3] sensor nodes to those objects in order to derive actions and complex situations, e.g. meeting or coffee break. Thereby, the recognition of situations is distributed among the sensor nodes. The goal is to support office workers in planning meetings, and control of environmental and office infrastructure components. In the AwareOffice, the Particle sensor nodes have to perform a set of sensing, communication and computation tasks. The sensor nodes detect multiple events in the environment through a rich phalanx of sensors and forward them. On the other hand, the nodes allocate some of their resources on communication and computation for collecting data from multiple sensing nodes and recognition of complex situations. We implemented the tasks as the following services with the properties listed below. Varying computation times are indicated through their relative occurrence in parentheses right after the computation times.

Service	Computation Time C_i (ms)	Period T_i (ms)
Voltage sampling	0.6	100
Audio sampling	3.4	10
Light sampling	1.4	100
Acceleration sampling	2.4	10
Force sampling	0.6	20
Temperature sampling	1.4	50
Volume computation	0.3	50
Shock detection (acceleration)	0.02 (99%) / 200 (1%)	20
Shock detection (force)	0.02 (99%) / 200 (1%)	50
Communication	0.5(50%) / 6 (45%) / 22.5 (5%)	13

Table 1. Service evaluation set for Particle sensor nodes which focus on sensing

The set is not static, but rather contains a dynamic changes in its properties. In particular, communication services and computation services are subject to strong variations of their computation time. The sensor services have short periods, which lead to a high system utilization. As a consequence, we consider the sets as well-chosen representatives for the evaluation. Our evaluation method is to benchmark the scheduling framework first without the extension of the dispatcher and without the use of the starvation controller. We will test the FIFO, RM and EDF scheduling strategies on the service set. In particular, our metrics in these benchmarks are the jitter of each job and the number of omissions. Latter indicates the potential level of starvation. Furthermore, we are interested in the adaptation of the period throughout the jobs. These result provide the baseline which our approach utilizing jitter correction and starvation controller will improve. At the end we will benchmark the sensor energy savings which are achieved by the scheduling.

VI. Scheduling in Unpredictable Environments

For deeper investigations on our approach, the service set from Table 1 was implemented in a scheduling simulator. The simulation covered a time span of 1.1 seconds. We ensured that all possible computation times occurred several times in that period. The service set was scheduled without the adaptation algorithms and afterwards with the adaptation algorithm. We ensured that the behavior of the varying computation time was the same in the comparison. The jitter correction was done immediately by the dispatcher. The starvation controller was called once after the dispatching of 30 jobs. During one simulation run it was called 13 times. The overall system utilization was always between 91% and 93%. For all simulations we considered the average jitter fraction and the average period over all jobs for a service. The jitter fraction expresses the delay between two successive executions of a job normalized to its period. It is computed as follows; i denotes the service, *now* and *last* are denote the current and last dispatch time

$$JitterFrac_i = \frac{t_{i,now} - t_{i,last} - T_i}{T_i}.$$

For the evaluation of the service execution, we average the jitter fraction over all jobs of a single service. This metric is abbreviated as $\overline{Jitter_i}$. The comparison between two simulation runs, one run without adaptation and the second one with adaptation delivers the difference in the jitter fraction, which is abbreviated as $\overline{\Delta Jitter_i}$. The advantage of this metric is that it includes information about job omission. If $\overline{\Delta Jitter_i} > 1$, then the jitter exceeded the period and therefore the job will be canceled by the dispatcher. As a result of the dynamic of the service set, we are interested in the change of the period of each service. After each simulation run we computed the average periodic interval of service which is defined as $\overline{I_i} = t_{simulation} / c$, where c denotes the number of jobs executed for the service i. Like for the jitter fraction these intervals were compared for each service in the two cases with and without the adaptation. The result is represented by $\overline{\Delta I_i}$. We evaluated our scheduling framework using the three scheduling strategies RMS; FIFO and EDF. The results are summarized in Table 2. Note that negative values represent an improvement when using the adaptation. Positive values indicate that the scheduling without adaptation performed better.

	RMS		FIFO		EDF	
Service	$\overline{\Delta I_i}$ (ms)	$\overline{\Delta Jitter_i}$ (%)	$\overline{\Delta I_i}$ (ms)	$\overline{\Delta Jitter_i}$ (%)	$\overline{\Delta I_i}$ (ms)	$\overline{\Delta Jitter_i}$ (%)
Voltage sampling	0.27	-4.4	12.21	2.8	21.32	23.7
Audio sampling	-1.31	-26.0	0.49	-13.6	-0.69	-27.7
Light sampling	0.27	-6.1	12.21	2.6	21.32	24.9
Acceleration sampling	2.04	-4.7	0.51	-22.7	1.37	-5.6
Force sampling	5.11	1.4	5.31	7.7	5.03	7.1
Temperature sampling	6.98	-4.1	6.81	8.1	12.19	4.4
Volume computation	2.23	-7.0	0.97	-25.3	0.67	-37.9
Shock detection (acceleration)	-6.44	13.4	-4.65	21.1	-24.01	-117.2
Shock detection (force)	-48.80	-109.0	-45.82	-99.9	-55.47	-113.3
Communication	-0.29	-3.6	-0.02	-0.3	-0.73	-2.8

Table 2. Results from our scheduling simulations; negative numbers indicate an improvement achieved by the adaptation compared to the non-adaptive scheduling

Interesting in the table above are the results indicating an improvement of more than 100%. A closer look on the data revealed that these services suffered under permanent starvation in the case where no adaptation was active. Reasons were a low priority in case of RMS, a very long deadline in the case of EDF and a period violation in the case of FIFO. The jitter fraction for the jobs of these services were constantly high above 1. As a result the averaged value over all jobs of those services exceeded the starvation border. In the simulation runs those situations could be handled through the starvation controller. The jitter fraction decreased dramatically below the 1-limit and as a result we obtained a difference of more than 100% between the simulation runs. Intuitively, this difference is explainable, if one is aware that the situation for this service changed completely from permanent starvation to actual execution.

A. *Utilization analysis*

In order to compare our results from Table 2 with our selection from Table 2 we analyzed the correspondence between the utilization of each single service and the performance of our scheduling framework. The utilization is computed according to formula (2) but for each service separately. The first result shows the correspondence between the utilization and the jitter fraction.

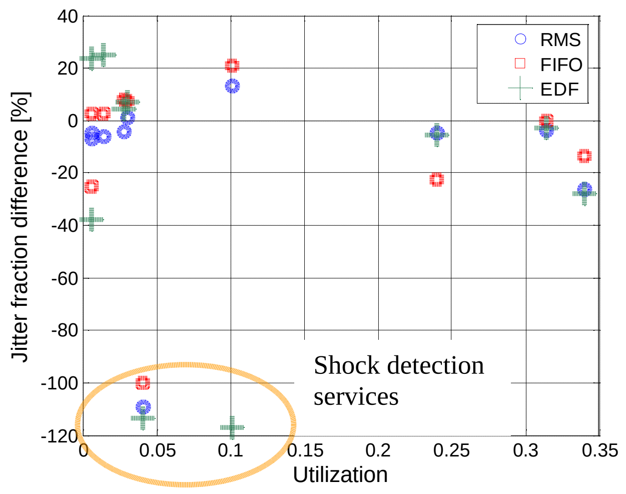

Figure 9. Correspondence between the utilization and the improvements regarding the jitter fraction

The figure above indicates that the scheduling framework works better on services with higher utilization. These are services where the computation time consumes a large fraction of the period. Services with a very short period share the same property. As an example consider the audio sampling service from Table 1. For services with a low system utilization the improvement is not clear. The jitter fraction is negative, this indicates an improvement, as well as positive – this is a weakening. Remarkable are the shock detection services with the large variation in their computation times. The achieved improvement is very high, since the starvation controller resolved the permanent starvation. However, the utilization is rather low, due to the low probability of their long computation time. For EDF the improvement through adaptation is exceptional high because of the very long deadline in 99% of the runtime. As a second result of our investigation we present the correspondence between utilization and the periodic interval difference in Figure 10.

The jitter correction will lead to a longer periods due to the fact that the next arrival time is computed on basis of the last invocation. For services with low utilization, this can be seen in Figure 10. However, the adaptation has a rather small effect on the period for services with higher utilization. Due to their shorter periods, they are already scheduled quite accurate. Nevertheless, the jitter correction can improve this behavior. Remarkable again are the changes for the shock detection services. Due to their long starvation phase the periodic interval is huge. The improvement through the starvation controller is then very significant.

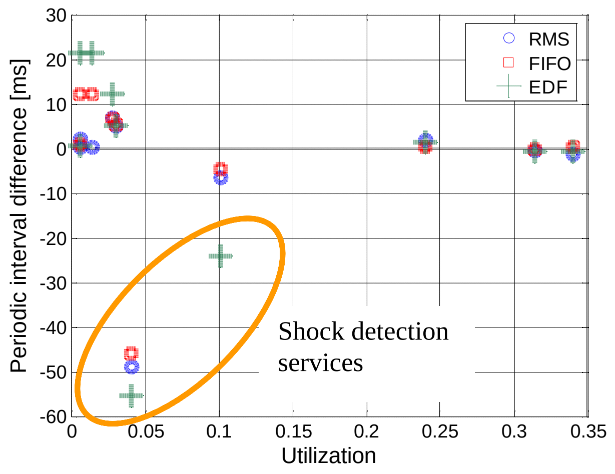

Figure 10. Correspondence between the utilization and the improvements regarding the periodic interval

B. *Automatic energy management*

Our enhanced dispatcher is able to support the power up and down of sensors. Thereby, the dispatcher queries the service for a new arrival time, but let it additionally know the current time. As an result, the service sets the arrival time for its next job before the period, leaving enough time for the start-up. The developer configures this start-up time. We evaluated the energy management using the acceleration sensor ADXL210 of our Particle Computer platform. The sensor needs 316 us for start-up and draws 1 mA. Figure 11 plots the energy consumption over a time frame of 1.1 seconds. The results were gathered using the service set from Table 1 under RMS and all adaptation mechanisms were enabled.

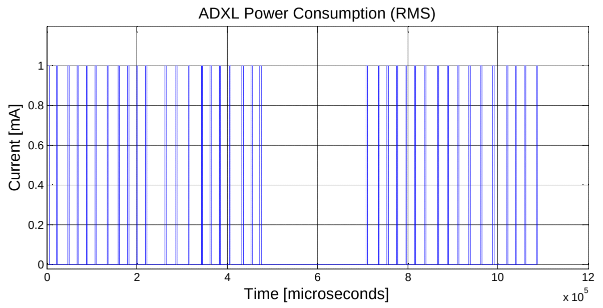

Figure 11. Power consumption ADXL acceleration sensor

Between 475ms and 708ms within the plot, the ADXL sensor-sampling job was skipped due to the long runtime of a shock detection job. The adaptive scheduling framework achieved for the overall time span a duty cycle of 9.5%. No extra code had to be placed in the application for switching the sensor on and off. Simultaneously, the adaptation mechanisms in the framework improved the jitter fraction. Automatic energy management becomes crucial for sensor node platforms incorporating chemical sensors like gas sensors with long start-up times.

VII. Implementation

The implementation of the scheduling framework was done on the Particle Computer platform (Figure 12). It bases on a Microchip PIC18F6720 microcontroller. This low power MCU has an instruction cycle of 0,2 µs and includes only 4K RAM and 128K ROM.

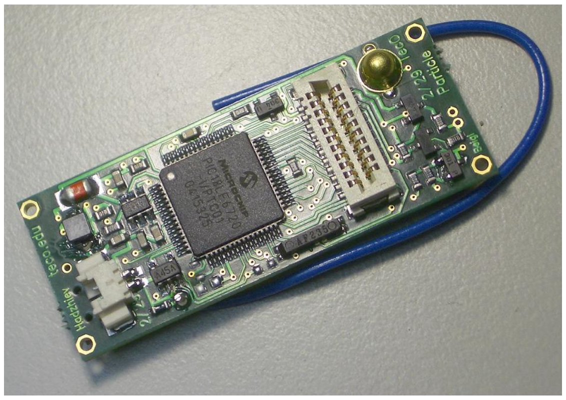

Figure 12. The Particle sensor node

Due to the resource constraints the implementation especially stresses on low scheduling overhead by the means of memory consumption and computation time. A 16bit timer at instruction cycle resolution can be used to trigger dispatching events regaining control from an interruptible background application. Because of the non-preemptive nature of services, no thread control mechanisms are needed. The service execution state is kept on top of the runtime stack of the background application. Services are stored within a bounded array as a set of service control blocks. Each block contains generic parameters such as its period and can be extended at compile time by optional parameters, e.g. the starvation level. The service control block represents an indefinitely long sequence of recurring jobs. Only the arrival time for the earliest job in time that has neither been executed nor skipped is stored with each service element. The service run state is implicitly encoded by the membership in either the ready or waiting queue. The default implementation supports 16 active services in the system. Each service control block consumes 5 Byte of RAM. Additionally 1 byte is needed to point to the queue heads. Structure splitting is used to store static parameters such as the pointers to the service itself. The pointers to the service procedure are kept in parallel jump table in code memory consuming no additional RAM. Overall this setup consumes only 81 byte of statically allocated RAM for the scheduler. Priorities of the jobs are reflected by the sorting order of the ready queue supporting both dynamic and fixed priority schemes. The waiting queue is strictly sorted by arrival time of the next waiting job of each service. The scheduler interacts with the dispatcher by inserting formerly waiting jobs into the ready queue and is called on each arrival of a new job. Arbitrary priority schemes can be implemented. The FIFO scheduler as an example simply retains the order of the waiting queue on the transition from waiting to ready. RMS uses the period length as sort criterion. The overhead of a single scheduler call is thus limited to the overhead of an insertion into a sorted list. Additionally another insertion is needed when moving a serving from ready to waiting virtually creating the next instance of a job. At a maximum that means n_w+n_r per job comparison operation; n_w and n_r being the number of jobs in each queue. By design n_w+n_r equals the total number of services n currently running.

Concrete timing measurements based on the implementation on a Particle 2/29 are depicted in Table 3. We use the algorithm from Figure 13 to implement the dispatcher. Insertion into the waiting queue is only done once per dispatch event. Insertion into the ready queue is on the average also done once per dispatch event. But, it can happen as often as n times if all services become ready at the same time. The concrete timings are strongly influenced by the linked list implementation. We are using forward link lists that are only accessible via the queue head. This explains the fairly high overhead the FIFO scheduler. In this case a more optimized implementation using just a single queue would have been better in terms of performance by avoiding the need to iterate the whole ready queue.

State transition	**Instructions-cycles**	**PIC18F6720**
Ready to running	15 cycles	3µs
Running to waiting		
create next instance	138 cycles	25.6µs
sort into waiting	32-752 cycles	6.5µs-150µs
Waiting to Ready		
FIFO Scheduler	32-480 cycles	6.5µs-92µs
RMS Scheduler	32-752 cycles	6.5µs-150µs
EDF Scheduler	32-752 cycles	6.5µs-150µs

Table 3. Timing Measurement of Service Transitions

On the average the iteration overhead for RMS and EDF is even less than FIFO since sorting terminates the iteration before the end is reached. RMS, EDF and the waiting queue insertion do not differ in the implementation in terms of sorting. Only the sorting key changes from the period in RMS to the deadline in EDF. This explains the equal measurements regarding those algorithms.

```
void dispatcher() interrupt timer1{

for(;;){
   time start=now();
   service current=pop(ready); //dispatch first
   dispatch(current); //run service uninterruptible
   interval C=time_diff(now(),start);

   service_calc_next(current,C,start); //create next
   insert_sorted(waiting,current);

   while(next=top(waiting) && get_arrival(next)=<now())
     schedule(ready,pop(waiting));  //call the scheduler

if (!top(ready)) {
service next=pop(waiting);
insert(ready,next);      //pre-schedule
     sleep_time= time_diff(now(),get_arrival(next));
     if(sleep_time>DISPATCH_OVERHEAD){
       set_timer1(sleep_time);
return; //defer loop to interrupt
}}}}}
```

Figure 13. Dispatcher implementation

We implemented the adaptation strategies jitter correction, starvation controller and the energy management. The overhead of the strategies is given in the Table 4. The real effort spent for the strategy heavily depends on its period. As a consequence, we state the basic overhead in µs and quote the period as we have implemented it.

Adaptation strategy	**Overhead**	**Strategy period**
Jitter correction	32.5 µs	Every dispatcher invocation
Starvation controller	6.5µs-191µs	Every 30 dispatcher invocations
Energy management	27.8µs (+ scheduling)	Every sensor period

Table 4. Adaptation overhead

VIII. Conclusion and Future Work

We presented an adaptive scheduling framework which is especially tailored to efficiently organize and processes on sensor network platforms which are deployed in unpredictable environments. The framework is implemented as a runtime environment for independent, non-preemptive services, which can be efficiently scheduled and executed even under high load and the varying computation times. We proposed two adaptation mechanism: jitter correction through the dispatcher and starvation control through a new service. Low jitter and a starvation control were the first two primary goals defined at the beginning of this paper. Our results show significant improvements regarding low jitter, and starvation control. The third goal of automatic energy management was also evaluated. The efficient implementation of the adaptive scheduling framework proved feasibility for sensor node platforms. For future work we will further investigate the scheduling framework and adaptation strategies. Crucial in our research will be the selection of appropriate service sets and their properties. A straight approach cannot be chosen here, since many effects we are investigating are directly influenced by the services in their specific combination. Our future work will also contain the investigation of other application areas. In particular, we see high potential of our approach in highly mobile scenarios.

Acknowledgment

The work presented in this paper was partially funded by the European Community through the project CoBIs (Collaborative Business Items) under contract no. 4270 and by the Ministry of Economic Affairs of the Netherlands through the BSIK project Smart Surroundings under contract no. 03060.

References

[1] M. Beigl, T. Zimmer, A. Krohn, C. Decker P. Robinson. “Creating Adhoc Pervasive Computing Environments”, Video at Pervasive 2004 in "Advances in Pervasive Computing", ISBN 3-85403-176-9, pp. 377-381, Vienna, Austria.

[2] P.Bratley, M.Florian, P.Robillard. “Scheduling with earliest start and due date constraints.” Naval Research Quarterly, 18(4), 1971

[3] C.Decker, A.Krohn, M.Beigl, T.Zimmer. “The Particle Computer System.” IPSN SPOTS, Los Angeles, USA, 2005

[4] A.Dunkels, O.Schmidt, T.Voigt. “Using Protothreads for Sensor Node Programming.” In Proceedings of the REALWSN 2005 Workshop on Real-World Wireless Sensor Networks, Stockholm, Sweden, June 2005.

[5] C.Han, R.Rengaswamy, R.Shea, E.Kohler, M.Srivastava. “SOS: A dynamic operating system for sensor networks.” Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, “System architecture directions for network sensors”, ASPLOS 2000, Cambridge, November 2000.

[7] T.J. Hofmeijer, S.O. Dulman, P.G. Jansen, P.J.M. Havinga. "AmbientRT - Real Time System Software Support for Data Centric Sensor Networks", ISSNIP 2004, Australia, December 2004

[8] K.Jeffay, D.F.Stanat, C.U.Martel. “On non-preemptive scheduling of periodic and sporadic tasks.” 12th IEEE Symposium on Real-Time Systems (December 1991), pp. 129--139.

[9] C.L.Liu, J.W.Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment.” Journal of the ACM 20(1), 1973

[10] F.Stajano, R.Anderson, “The grenade timer: Fortifying the watchdog timer against malicious mobile code,” MoMuC 2000, Waseda, Tokyo, Japan, Oct. 2000.

[11] J.Stankovic, K. Ramamritham. “The design of the spring kernel.” In Proceedings of the IEEE Real-Time Systems Symposium, December 1987