Slotted Programming for Sensor Networks

Roland Flury
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
rflury@tik.ee.ethz.ch

Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

ABSTRACT

We advocate a novel programming approach we call slotted programming that not only addresses the specific hardware capabilities of sensor nodes, but also facilitates coding through a truly modular design. The approach is based on the temporal decoupling of the different tasks of a sensor node such that at any time at most one task is active. In contrast to traditional sensor network programming, slotted programming guarantees that each of these tasks can be implemented as an independent software module, simplifying not only the coding and testing phase, but also the code reuse in a different context. In addition, we believe that the proposed approach is highly qualified for energy efficient and real time applications. To substantiate our claims, we have implemented slotos, an extension to TinyOS that supports slotted programming. Within this framework, we demonstrate the advantages of the slotted programming paradigm.

Categories and Subject Descriptors

D.1.0 [**Software**]: Programming Techniques General

General Terms

Design, Performance, Reliability

Keywords

modularity, context-free programming, time slicing

1. INTRODUCTION

Programming applications for networked systems can be painful. Software for distributed systems tends to be *heavy*, consisting of several components and layers that interact with each other in a non-trivial way to cope with various transient or permanent failures and most likely also some kind of dynamics. On top of that there is the whole process of software development, e.g. issues such as debugging between remote nodes. But programming embedded systems is not easy either. Embedded systems ask for *light* software that does not waste resources such as memory, processing power, or energy. In addition, the systems often need to meet tight run-time requirements and guarantee a predictable execution. Again the software development cycle is tedious, as the programmer does not have direct access to the hardware, but must go through cross-compilation and cross-debugging.

Wireless sensor networks clearly inherit all the problems from networked distributed systems as well as embedded systems. To make matters worse, running *heavy* distributed systems software on *light* embedded systems hardware sounds like an contradiction at first. However, sensor networks can be done, but the work to build such energy efficient protocols and applications shall not be underestimated. Current energy efficient implementations need to be optimized across the entire application. This comes at the price of a non-modular design, adding further functionality may require drastic changes in code, and even small adaptations to parameters or changes of the environment may trigger a serious investment in re-testing.

Not surprisingly, there is a trend towards simplifying the programming environment. Some proposals abandon the idea of energy efficient hardware altogether, and instead advocate a Linux or Java VM framework [2, 11, 19]. Some other proposals [12] envision an IP layer hiding a general purpose low-power networking stack. These efforts simplify the programming task tremendously. But at the same time, they introduce abstractions that hide alternative usages of the underlying hardware which may be even more energy efficient. In addition, the abstractions hide the complexity of the underlying implementation. This is dangerous especially for time sensitive tasks, where hardware devices such as the CPU or the radio must be available exactly at a given time. It is thereby not important to accomplish a task as fast as possible, but exactly at the desired time without being delayed. But today's best-effort operating systems cannot guarantee the necessary precision as they abstract away execution time [13].

In this work we outline the slotted programming paradigm which supports both modularity and energy-efficiency in a well-defined time model. This programming approach is not orthogonal to the current trend of providing high level abstractions. It rather extends the abstractions with a predictable execution scheme, giving back full control to the application developer. Our concept is simple: We consider the different tasks of a sensor node such as clock

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IPSN'10, April 12–16, 2010, Stockholm, Sweden.
Copyright 2010 ACM 978-1-60558-955-8/10/04 ...$10.00.

synchronization, routing, topology control, sensing, or code updates. Each of these tasks is given time slots during which it performs its operations in parallel on all nodes. The time slots need to be assigned in a non-overlapping way such that at any time, at most one task executes its code. If a job has not terminated at the end of its assigned time slot, it must be suspended and may only continue its execution during its next time slot. Thus, tasks do not interfere with each other, and they do not interrupt each other. Therefore, each task can be implemented as an independent *module* which can be exchanged easily. Indeed, alternative modules for the same task can be tested and compared within the same application.

2. RELATED WORK

The most prominent OS for sensor networks is TinyOS [22], which is a purely event driven framework supporting many hardware platforms. Over time, TinyOS was extended with new features to ease application development and reduce programming errors. Most recently, a safe type-system [4] and support for multi-threaded programming [17] were added. However, the quest for an optimal programming environment was not only led by TinyOS. For instance, the MANTIS OS [1] supported preemptive multi-threading well before TinyOS did, and Contiki [5] is based on lightweight protothreads [6]. The notion of virtual memory was introduced in the t-kernel [9] to protect the operating system from being corrupted by the user application, and SOS [10] supports the dynamic loading of modules, avoiding the static linking at compile time.

Closest to our programming model is Nano-RK [8], a reservation-based real-time OS, and the pixie OS [15], which allows for a dynamic resource allocation at runtime, predicting the energy draw of the operations in an online fashion. Both, Nano-RK and pixie operate on a deadline-based scheduling policy which executes the tasks within a certain time window. With the slotted programming model, however, we guarantee the execution of a code fragment precisely at a given time, e.g. to wake up the radio module of two nodes in parallel, without requiring any guard time. This is not possible with a real-time OS with deadlines where a task is executed before a given time. Slotted programs are much simpler, as they operate on a static schedule. But in return, they regain full control over the application and can provide execution guarantees (such as the execution of a task precisely at a given time) that cannot be given by the thread scheduler, which needs to solve an online allocation problem.

To reduce the learning curve and simplify the development of sensor network applications, TinyOS and other operating systems support multi-threaded programs. Furthermore, low power listening protocols can be used to provide a basic MAC layer to the application developer [7, 18, 21] . In fact, Hui and Culler [12] envision an IPv6 stack on top of such a MAC layer to hide the underlying networking to the application developer. We embrace this trend towards a standardized communication framework and believe it to be perfect for prototyping sensor network applications. However, when it comes to the development of project specific optimizations, these programming techniques and abstractions can be obstructive. Firstly, the execution of a multi-threaded application is hard to predict. As a result, the development and also debugging is much more involved as some errors may occur only sporadically under a specific preemption. Similarly, the introduction of a networking abstraction such as a low power listening MAC introduces a rather unpredictable use of the CPU and the radio device, possibly delaying other tasks. Thus, the developer does not have complete control over the application, which makes it harder to ensure a bug-free program. Last but not least, we believe that special purpose communication protocols can increase considerably the energy efficiency of many applications. For instance, the task of collecting a message from every node once a minute causes a duty-cycle of 0.65% with the IPv6 network stack of [12]. In Dozer [3], a data gathering protocol with a dedicated MAC layer, a message was collected every 30 seconds. The duty-cycle for this task was reported to be only 0.167%, reducing the overhead by a factor of 8. However, this improvement comes at the price of a higher latency and a completely asynchronous application that is hard to maintain. For comparison, our slotted clock synchronization algorithm receives and sends one message every 32 seconds and requires an average duty-cycle of only 0.06%. Of course, these numbers cannot be compared directly as they stem from different applications, but they indicate that there is still a lot of potential for saving energy. With the slotted programming paradigm, we demonstrate that it is indeed possible to write modular applications that are also energy efficient.

3. BACKGROUND

Access to the different hardware components such as the radio module, timers, flash chip, or the sensors is often done through software arbitration or the explicit knowledge that no other code fragment is using the desired component. Clearly, the latter approach is not only prone to undetected conflicts, but also renders code reuse very difficult. But software arbitration alone is not enough to write modular applications as we outline in the following points.

Real time applications.

The load of the CPU is hard to predict, as other pieces of the application may require computational complex tasks at any time. As a result, tasks and synchronous events may execute with a non-negligible delay, which poses problems to the implementation of time critical applications where some actions should happen exactly at the prescribed time. This is in sharp contrast to classic real time systems, where the tasks should execute within a larger time window before a given deadline. For instance, even the simple task of enabling the radio on two neighboring nodes at exactly the same time is hard to achieve, as any of the nodes may be occupied with other tasks and therefore delay the power-on command. A common approach to suppress this problem is to use a guard-time and enable the radio a bit earlier than arranged. Apart from the fact that it is hard to predict good guard-times, such applications use more energy than necessary and also gain in complexity. Alternatively, the radios could be powered on in an asynchronous code block, ensuring instantaneous execution if no other asynchronous event interferes. However, implementations which rely on preemptive code execution (e.g.[3]) are quite hard to maintain, as any modification or extension may affect any other part of the system. As such asynchronous systems are often very complex and even the slightest anomaly in the code can lead to misbehavior such as memory corruption

or even a deadlock, we believe that this approach should be avoided whenever possible. Multi-threaded operating systems are not improving the situation either, as they even explicitly permit preemption of tasks, delaying operations even further. Prioritization does not resolve the problem as the thread with the lower priority still experiences a delay.

Modularity.
In most sensor network scenarios, each network node needs to fulfill several tasks, e.g. clock synchronization, sense the environment, process the measurements, and disseminate the sensed data. To obtain a highly optimized application that uses as little energy as possible, the usual approach is to combine the required tasks as well as possible. In our example, this could mean that the messages required for the clock synchronization are piggybacked onto the data gathering messages or vice versa. Whereas such an implementation can be very energy efficient, it is also highly specific to the given problem instance. Therefore, any modification or extension needs to take into account the entire application, and partial code reuse in a different application is cumbersome.

Incomplete algorithm design.
We have already argued that any two tasks running in parallel may interfere by reducing the responsiveness of the node. Similarly, the software arbitration may temporarily block the access to a hardware resource such as the radio module while another task is using it. Also, if two hardware components happen to be connected to the CPU through a shared bus, at most one of them may be accessed at any time. This is the desired behavior, but it is often difficult or even impossible to predict such conflict patterns. Therefore, algorithms for sensor nodes are normally designed under the simplifying assumption of immediate access to the hardware, ignoring the fact that another task may run in parallel competing for the same resources. Adapting these algorithms to the existing hardware constrains is often a challenge, especially for time critical applications. Even if a delay may be tolerable, the unpredictable timing may introduce new side effects which must be verified to not break the original algorithm.

Configuration conflicts.
A wrongly configured hardware device not only provokes unpredictable responses but may even hang a sensor node. If several tasks require conflicting configurations, the application must be careful to always load the appropriate configuration. If a task interrupts the execution of another task due to asynchronous execution or multi-threading, this may not be possible at all.

4. SLOTTED PROGRAMMING

The above mentioned issues arise because several tasks of a sensor node may execute simultaneously. With the slotted programming paradigm, we introduce a temporal arbitration between the tasks to resolve the described problems, and at the same time, slotted programming fosters code modularity.

The slotted programming approach decouples the different tasks of a sensor node to render them as independent as possible such that each task can be implemented as a self-contained software module. The decoupling is achieved through temporal separation of the different tasks, assigning each module a time slot during which it may have exclusive access to all resources. Figure 1 shows a possible slot assignment on a sensor node with 4 distinct tasks. This is in sharp contrast to an uncoordinated execution model, where tasks may collide and experience unpredictable side effects.

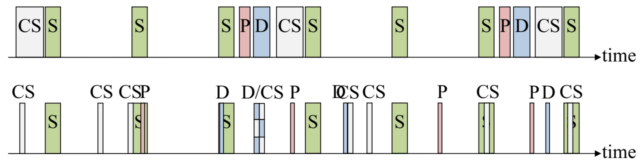

Figure 1: A slot assignment on a sensor node with 4 tasks (top): Periodically sample a sensor [S], process the sensor data [P], disseminate the processed data [D], and run a clock synchronization algorithm [CS]. In an uncoordinated execution model where tasks are not temporarily decoupled, several collision patterns may occur (bottom).

4.1 The Basics

The slotted programming paradigm builds on a synchronous execution model. It requires that all nodes have a global notion of time, i.e. they need to run some kind of clock synchronization algorithm. This network time is used to schedule the execution of the software modules, such that the same software module executes simultaneously on all nodes.

Each task of a sensor node is implemented as an independent software module providing the desired functionality. E.g. there may be a clock synchronization module, a data gathering module, a sampling module, and a data processing module. To obtain the temporal decoupling with other software modules, each software module must ensure that its code executes only within an assigned time slot and that it causes no side effects outside this slot.

The software modules are integrated into the slotted system by allocating time slots for each of them. Whereas arbitrary complex schedules can be built, simple periodic schedules similar to the one shown on top in Figure 1 are already sufficient for most applications. The only restriction for the overall schedule is that there may be no region where slots overlap, otherwise the temporal decoupling would be broken. Depending on the scheduling complexity, this property can already be tested at compile time. Alternatively, a run-time check may be applied.

The basic support for slotted programs is provided by two components: A clock synchronization module and a scheduler module. The latter executes the desired schedule by signaling each module the start and end of its time slots. Figure 2 illustrates a schematic view of a slotted system.

4.2 Discussion

With the slotted programming paradigm, we foster modular programs for sensor nodes. The key component is the temporal decoupling of modules which guarantees that at any time, at most one module is running its code without being preempted. This decoupling lays the basis for the following properties of slotted programming:

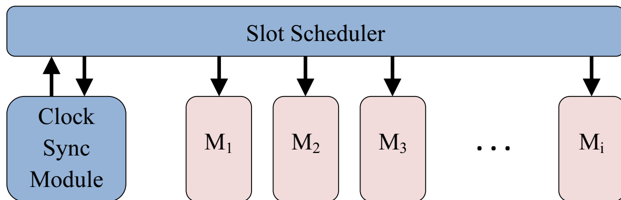

Figure 2: Schematic view of a slotted system: The slot scheduler starts and stops the software modules, the clock synchronization module is responsible for obtaining the network time.

- During the time slot of a module, full access to all hardware resources is granted to the module, it is neither blocked nor delayed by other tasks. This also includes the hardware configuration which is guaranteed to be consistent during the entire slot. Overall, the programmer of the module can count on the timely availability of the hardware, which allows for simpler and more efficient implementations using less energy.
- Each module can be implemented and tested independently as there are no side effects from other tasks. This greatly supports the software development, as it is much easier to code and test several small pieces instead of writing and testing an entire application altogether.
- The reuse of a module in a different applications is straightforward as each module is supposed to work independently of the context. The only thing that may need to be adjusted is the scheduling of the time slots.
- Last but not least, the modularity reduces the software complexity, which makes it easier to collaborate or continue on a given project. For instance, we have had several students working with our slotted programming environment during the past year and we believe that their success can also be traced back to the slotted approach.

4.3 Limitations

Whereas the above properties are desirable for all applications, the slotted programming paradigm has its limits. For instance, if a sensor node must constantly perform some action, e.g. sample a sensor at 100Hz, other modules cannot be scheduled without conflicting with the sensing task. In fact, there is no clean solution to this fundamental problem as a complete decoupling of the modules is impossible. Similarly, if a node needs to react to external events which may arrive at any time, e.g. triggered through an interrupt, the node should delay its actions until a dedicated time slot to handle the event is scheduled. This may not be acceptable for some applications where a rapid reaction is required. We believe, however, that in the case of such scenarios, the slotted programming is still useful. On the one hand, the application developer may be more aware of the existing conflicts, which helps to predict possible side effects. On the other hand, the conflicting modules can be designed to reduce their actions during foreign time slots to a minimum, shifting the non-time-critical operations to their assigned time slots. In addition, the sequential execution of the slots may require more CPU time than an optimally scheduled task list which runs non-conflicting tasks, e.g. radio and sensor accesses, in parallel.

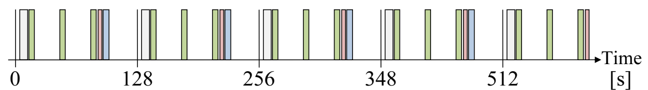

Figure 3: The slot scheduler repeats a partial schedule generating a periodic execution of the allocated time slots.

5. SLOTOS

To demonstrate the potential of slotted programming in practice, we developed slotos, an operating system that supports slotted programming. In general, any operating system can be adapted to provide this functionality. For our reference implementation, we chose to extend TinyOS 2.1. Being single-threaded in its default version, TinyOS is a perfect candidate for applying the slotted programming paradigm.

In slotos, the *slot scheduler* is responsible to invoke the execution of the software modules according to their time slot reservations. To simplify the scheduling of periodic time slots, slotos provides a periodic slot management with an allocation window of 128 seconds. Each module allocates its time slots within this allocation window which is repeated periodically by the slot scheduler to obtain the overall schedule, see Figure 3 for an example. With this scheduling approach, slotos facilitates the allocation of periodic time slots which we believe to be sufficient for most applications. The number of allocated slots as well as the length of the different slots has no influence on the overall performance of the application as the scheduling overhead consists only in setting a timer to start and eventually stop each of the slots. The size of the allocation window is currently set to 128 seconds, but it may be set to whatever fits best the application needs. For more sophisticated schedules, the dynamic time slot reservation of slotos may be used to reorganize the schedule at run-time.

Interface 1 sketches the events and commands provided by the slot scheduler. During the `init()` event, which is called upon booting the node, each module allocates its time slots. `addTimeSlot()` is used to reserve an additional time slot, and `modifyTimeSlot()` reschedules an existing time slot. Whenever a time slot starts, the `startSlot()` event is signaled on the corresponding module indicating an estimate of the current synchronization quality. Finally, the end of the slot is signaled by the `stopSlot()` event.

5.1 Policy Enforcement

The temporal decoupling of the modules is the key of slotted programming. Whereas a static schedule can be checked at compile-time to have no overlapping regions, dynamically generated schedules can only be verified at run-time. slotos detects overlapping time slots and refrains from scheduling them simultaneously. Instead, it provides a best-effort service delaying the start of the later time slot until

Interface 1: Slot scheduler interface

```
events
  void init()
  void startSlot(slotID, slotCmd, syncQuality, syncStatus)
  void stopSlot(slotID)
commands
  slotID addTimeSlot(startTime, length, slotCmd)
  void modifyTimeSlot(slotID, startTime, length, slotCmd)
  bool testSchedule()

  void stopScheduling()
  void continueScheduling()
  ⋮
```

the end of the active time slot. It is important to note that this error handling approach should not be exploited as a feature of the slot scheduler. It is rather a last resort to guarantee a continuous execution of all reserved time slots, based on the assumption that the broken schedule is only of temporal nature.

The temporal decoupling requires that the software modules operate only during their assigned time slots. This property, however, is much harder to verify or even enforce, as the modules should be allowed to execute arbitrary actions within their time slots. For instance, a module may set a timer to fire outside its time slot or initiate a split phase command whose callback returns only after the time slot terminates. Whereas the timer issue can be addressed by canceling timers that are set outside the current time slot, callbacks from the hardware are device specific and often hard to predict. The current implementation of slotos does not try to detect or avoid activity outside the assigned time slots and requires the module developer to adhere strictly to the time constraints. Thus, slotos does not enforce slotted programming but only provides a suitable execution environment.

5.2 Timers

It is often the case that two neighboring nodes wish to wake up simultaneously to exchange messages or perform other operations. Slotted programs support such interactions as the software modules are scheduled simultaneously on all nodes. Thus, it is sufficient for a module to set a timer to be waken up at the arranged time. To obtain a truly decoupled system, the modules should respect the following guidelines when setting timers:

- Most importantly, timers may only be set to fire within the current time slot. If a timer should fire during a later time slot, the module should remember this and start the timer only at the beginning of the desired time slot. Thanks to this restriction, timers never need to be updated to reflect a modified network time because no timer is active when the clock synchronization module adjusts the network time.

- Whenever possible, timers should be started relative to the start time of the current time slot. This results in a more precise timing than when using offset timers, as their fire time depends on the time when they are set. In TinyOS, this can be done using the `startOneShotAt()` method of the Timer interface.

- To account for fluctuations of the clock synchronization, a module may apply guard times to ensure that it does not miss an arranged meeting. In the case of a scheduled message transmission, the receiver wakes up a bit earlier and the sender sends the message a bit later to ensure that the two meet. The estimation of the clock synchronization passed on in the `startSlot()` event may be used to adaptively set the guard times.

5.3 Clock Synchronization

The clock synchronization module and the slot scheduler are tightly coupled: Whereas the slot scheduler depends on the network time to schedule the modules, the clock synchronization module itself is scheduled by the slot scheduler. This circular dependency is no problem as soon as the node is roughly synchronized, but special care needs to be taken while a node is not synchronized. slotos supports two approaches to break this circular dependency.

(A) The clock synchronization module may temporarily turn off the slot scheduler. During this time, no other module is scheduled to execute and the synchronization module is free to access the radio for an extended time to receive synchronization messages. Once an approximate synchronization is available, the slot scheduler can be turned on again.

(B) It is often undesirable to completely shut down the slot scheduler as this also stops any other activity on the sensor node, including the sensing tasks for which it was deployed. But if the clock synchronization module can only access the radio during its assigned time slots, it may never receive synchronization messages if the neighboring nodes have a different notion of time. To accommodate such scenarios, slotos explicitly permits the clock synchronization module to break the temporal modularity and listen on the radio also outside its assigned time slot. But this violation of the slotted programming paradigm needs to be taken into account by all modules that are scheduled during this time. In particular, other modules should refrain from using the radio or turning it off. For that purpose, the `startSlot()` command informs the modules about the current state of the synchronization. Fortunately, this restriction does not really limit the functionality of the node any further, as communication with neighboring nodes is likely to fail anyways while it is not synchronized.

The choice between the two approaches depends on whether the sampling or the synchronization is more mission critical. Clearly, the second approach is not as clean as we would like, but it is as modular as possible for the given application requirements. In either case, the clock synchronization module must provide the interface shown in Interface 2. Whenever the network time is modified, `timeChanged()` needs to be called such that the slot scheduler can adapt the schedule. Additionally, several methods to convert between hardware and network time and a method to query the current synchronization quality should be provided.

Interface 2: Clock synchronization interface

```
event
  void timeChanged()
commands
  time hardwareToNetworkTime(hwTime)
  time networkToHardwareTime(netTime)
  time networkIntervalToHardwareInterval(netDT)
  quality getSynchronizationQuality()
```

6. SLOTTED CLOCK SYNCHRONIZATION

As clock synchronization is a central part of slotos and slotted programming in general, we now describe in more detail the clock synchronization module that comes with slotos. Being implemented itself as a software module, we use this synchronization module to demonstrate the advantages of the slotted programming paradigm.

We chose to implement the recent PulseSync protocol [14], which was shown to be asymptotically optimal. In contrast to the often used FTSP algorithm [16] where the time signal traverses the network quite slowly, the PulseSync algorithm ensures that the time signal traverses the entire network in a pulse, providing all nodes with a fresh timestamp within a short time interval. As each node forwards the time signal shortly after receiving it, the degradation of the time signal due to local clock skew on the forwarding node can be reduced considerably.

The synchronization protocol is quite simple: A dedicated root node dictates its time to the remaining nodes. Nodes in the immediate neighborhood of the root node learn the network time directly from the root node. Once a node is synchronized, it disseminates synchronization messages itself such that nodes not directly connected to the root synchronize indirectly. Each node selects a single parent node to which it synchronizes, resulting in a tree algorithm. The secret of the PulseSync protocol is that a node should synchronize its children shortly after receiving a synchronization message from its parent. To avoid message collisions, each child waits for some random amount of time before sending out its own synchronization message. In each transmitted message, the sender includes the seed value of the random number generator that will be used to send forthcoming sync messages. From this information, the receiving child can predict when its parent sends the next sync message and enable its radio just for the required time period. The child can even predict the arrival of a sync message when it has missed several messages in between. This is possible as we use a circular random number generator where the generated number is used as the seed for the following draw. In the following, we show that the implementation of such a scheme is actually quite natural with the slotted programming approach.

6.1 Pipelined Synchronization

slotos offers a time window of 128 seconds to schedule the execution of the software modules. Our clock synchronization module reserves 4 time slots of 1 second in regular intervals such that the module is launched every 32 seconds, see Figure 4. Within these assigned time slots, the module may perform any actions as the slotted programming guarantees that no other task interferes. In our case, a synchronization pulse is sent through the entire network in each of these assigned slots. I.e. each node receives one sync message from its parent and broadcast a sync message to its children.

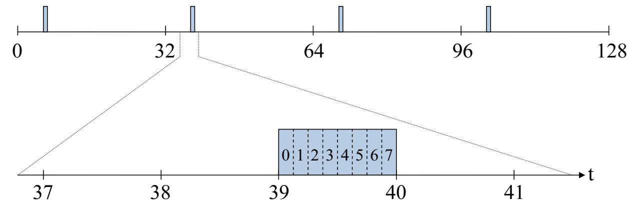

Figure 4: The default clock synchronization module of slotos allocates a slots of 1 second every 32 seconds. The remaining time can be used arbitrarily by other software modules. The synchronization module divides its time slot into k logical cells (in our case, $k = 8$). The root node broadcasts its sync message in the first cell with ID 0. A node that is h hops away from the root receives a sync message from its parent in the cell $(h - 1)$ mod k and sends its sync message in the cell $(h$ mod $k)$.

As we have already outlined above, we apply a simple pipelining scheme to schedule the transmissions along the synchronization tree. The synchronization module achieves this by dividing its time slot into k cells of equal length (see Figure 4) and restricting the transmission of the sync message to one specific cell. This cell assignment is based on the number of hops the node is away from the root node (on the synchronization tree). Note that nodes with a distance of k or more hops to the root reuse cells already assigned to nodes much closer to the root.

The choice of k is driven mainly by two constraints. On the one hand, k should be chosen as small as possible such that the sync messages can be spread over a longer cell, reducing the probability of collisions. On the other hand, if a node is $i \cdot k$ hops away from the root (with $i \in \mathbb{N}^+$), the node sends its sync message in the cell with ID 0, but receives the sync message from its parent only in the last cell with ID $k - 1$, not achieving the desired pipelining. Thus, the larger k is, the fewer nodes break the pipelining. In the current implementation of slotos, we have set $k = 8$ to ensure a perfect pipelining for our sample networks.

6.2 Experiments

We have logged the performance of our clock synchronization algorithm while testing our alarming system described in Section 7. During the 168 hours of the experiment, a total of 18900 synchronization rounds were performed. To measure the quality, a node determines its clock offset to its parent whenever it receives a sync message. Throughout the experiment, we measured an average offset of 1.45 time units with a variance of 1.21. This is equivalent to an average offset of 44μs as our hardware clock runs at 32768Hz.

The energy consumption of the clock synchronization is dominated by the energy used to send and receive messages. To get a first approximation on the usage, each node logged how long the radio module was enabled for each synchronization round. The cumulated up-time of the nodes is plotted in Figure 5. Note that the root node has a much smaller slope than the remaining nodes as it does not receive sync

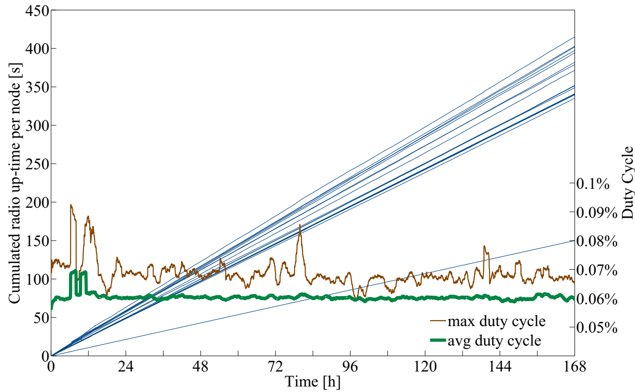

Figure 5: The nearly straight lines show the cumulated radio up-time of the 19 nodes using the left scale. The single line at half the rate belongs to the root node which does not need to listen for sync messages. The two horizontal curves indicate the temporal change of the duty cycle using the right scale.

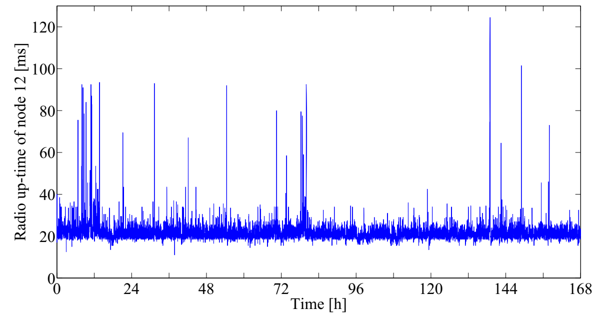

Figure 6: Node 12 has the highest duty cycle as it repeatedly loses its synchronization parent. This plot shows the up-time of node 12 for each synchronization round. The peaks occur when node 12 is looking for a new parent.

messages. It is also interesting to note that the lines of the non-root nodes diverge the longer the experiment runs. The varying slopes are due to our synchronization algorithm which tries to keep the guard time for receiving the sync messages as short as possible. Nodes that tend to loose packets adapt a larger guard time, slightly increasing their up-time. For instance node 12 (see also Figure 10) has the steepest slope. During the experiment, this node changed its parent several times, sometimes even synchronizing to node 7 across the building. The peaks in Figure 6 indicate when node 12 was looking for a new parent.

On average, the non-root nodes required an overall duty cycle of 0.06% for the synchronization, node 12 has the highest duty cycle of 0.07%. The temporal progress of the duty cycle is shown in Figure 5. The curves show the average (maximum) duty-cycle for the preceding 200 synchronization rounds. Again, the peaks fall together with parent elections.

6.3 Discussion

We sketched the implementation of a clock synchronization algorithm with the slotted programming approach, leaving out quite some details. For instance, the choice of a reliable parent is intrinsically difficult as the quality of a parent may change over time. Our goal was to demonstrate that it is indeed possible to write a modular clock synchronization that can be replaced by any other synchronization module without consequences for the remaining application. Furthermore, we would like to point out that even a simple implementation as the one described above can be quite energy efficient.

7. SLOTTED ALARMING

We now sketch the implementation of an alarm system in which any node of the network can alarm the entire network. Again, we will not describe all technical details of the alarming technique, but rather focus on the overall structure of the software to demonstrate how the slotted programming helps. In particular, we will compare two techniques to detect an alarm and compare the two approaches. Having a slotted environment, this boils down to switching between two different alarming modules and measure their success rate.

Our goal is to build an alarm system that is reliable, has low latency, and uses as little energy as possible. We thereby focus on the wake-up phase of the alarm system, leaving out the verification phase that should be executed before relaying an alarm to the upper layers. The end-to-end delay of the alarm can be minimized by pipelining the transmissions of the nodes. But as any node of the network may initiate an alarm, the pipelining would need to provide an any-to-all support. To overcome this complex task, we reduce the problem to the following two pipelinings: any-to-root and root-to-all, where the root is a dedicated node, e.g. a sink node or the node leading the clock synchronization. This results in a 2-phase approach, where the initiating node first signals the root node about the alarm (any-to-root) and the root then signals the alarm to the entire network (root-to-all).

The classic approach to both pipelinings is to build a BFS tree from the root node. When sending messages to transmit the alarm, however, care has to be taken to not cause interference. In the root-to-all phase, the root node first broadcasts its message to all its direct children in the tree. Upon receiving the message, the children should be coordinated to not forward the message at the same instant. The same problem exists in the any-to-root phase, where several children of a parent may wish to signal an alarm, or an alarm is propagated on several branches of the BFS tree. This issue can be solved by either using a global schedule or by applying randomization techniques. But both solutions are not satisfactory as they increase the end-to-end delay. Furthermore, the overhead to maintain a global schedule may be considerable.

7.1 Signaling of Binary States

As we are interested only in binary states (e.g. alarm ON, alarm OFF), a much simpler solution is possible. For instance, the value ON may be encoded by sending a message and the value OFF by being quiet. In the any-to-root

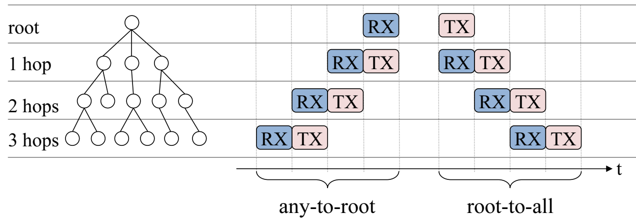

Figure 7: Alarms are disseminated in two steps. In the any-to-root step, any subset of nodes can signal the root node. The root in turn then relays the alarm to everybody in the following root-to-all step.

case, each parent only needs to detect whether any of its children sends a message, and depending on the outcome, send a message itself or not. To do so, the parent only needs to detect the activity on the radio channel, e.g. through an RSSI sniff, eliminating the interference problem. Therefore, several children may send their message simultaneously without causing any reception problems. The same holds for the root-to-all case where all children of a parent may send their message simultaneously. As a result, we can apply a tight pipelining which schedules the reception and transmission solely based on the hop count to the root node as depicted in Figure 7. Such a tight pipelining minimizes the end-to-end delay as well as the energy consumption, as any node only needs to receive and send twice, independent of the number of its children.

In the above example, the proposed encoding requires that the ON state strictly dominates the OFF state. I.e. whenever at least one node signals the dominant ON state, the ON state should be disseminated to the entire network. Only when no node of the entire network requires the dominant state, the subordinate OFF state is applied. This implicit conflict resolution between the two states is a natural choice for many application scenarios such as alarms (the alarm state is dominant), or any other form of wake-up signaling where a single node should be able to signal the root (using only the any-to-root pipeline) or all nodes of the network. Indeed, the proposed signaling approach may be useful in other context than alarms. For instance, it may be used to enable or disable entire software modules, e.g. a debugging or configuration module that needs not to run most of the time.

7.2 RSSI vs Waves

Many MAC protocols sample the radio channel prior to sending a message. If activity is detected, the transmission is delayed to avoid a collision with the ongoing transmission. This sampling is called clear channel assessment (CCA) and is often performed by an RSSI module on the radio chip which indicates the received signal strength. If the value is above a given threshold, the channel is assumed to be occupied. We propose to use exactly this tool to circumvent interference problems while signaling an alarm. Indeed, the approach works surprisingly well - as long as all nodes are deployed indoors. But as soon as we placed nodes on the roof of our building, the RSSI module started to intercept foreign signals, leading to roughly 30% false positives, i.e. the RSSI module indicates activity on the channel even though none of the nodes is sending.

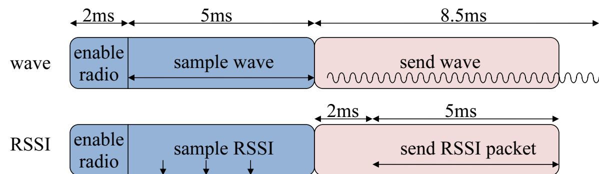

Figure 8: When a node is scheduled within the pipeline, it enables its radio, samples the RSSI or the wave, and depending on the outcome, transmits itself an RSSI packet or a wave, respectively.

To overcome this problem, we suggest the following approach: Instead of sending an arbitrary message which is detected through the RSSI module, a node that wishes to communicate an alarm sends an unmodulated wave of constant frequency which is detected at the receiving nodes. With this wave based approach, we can reduce the probability to misinterpret noise as an alarm, as the receiver expects a clearly predefined signal. It is important to note that several nodes may send an alarm in parallel, as the superposition of several waves results in a wave of the same frequency. Fortunately, the probability for a complete attenuation is extremely low, and our experiments indicate that the number of false negatives is in the range of 0.1% and comparable to the RSSI approach (See Table 1).

7.3 Slotted Signaling

In the remainder of this section we outline the implementation of a test application to compare the RSSI and the wave approach for the signaling task.

7.3.1 Signaling Module

The task of the signaling module is to run an any-to-root sweep to detect requests to change a state, followed by a root-to-all sweep to disseminate a potential request to the entire network. Both sweeps are implemented in a pipelined fashion as described in Section 7.1. As the timings for the RSSI as well as the wave approach are similar, we can use the same pipelining for the comparison, see Figure 8. The time to enable the radio is approximately 2ms and the time to send an RSSI message is 4.5ms. The RSSI value is sampled three times in a row, indicating a signal only if all three measurements show high activity on the radio channel. We hoped to reduce the number of false positives with this approach, but the experiments did not show a significant improvement compared to a single RSSI sniff, indicating a high correlation of the RSSI measurements. The sampling time for the wave is set to 5ms yielding 378 samples, and the transmission time for a wave is set to a total of 8.5ms to account for synchronization fluctuations.

The signaling module allocates itself a time slot of 1 second. The first 500ms is reserved for the any-to-root pipeline and the second 500ms for the root-to-all pipeline. As the overlap of the sender and receiver is only 7ms in each pipeline step, the depth of the pipelines may be up to 60 hops (leaving some time gap at the beginning and the end of the time slot). Thus, the current settings theoretically

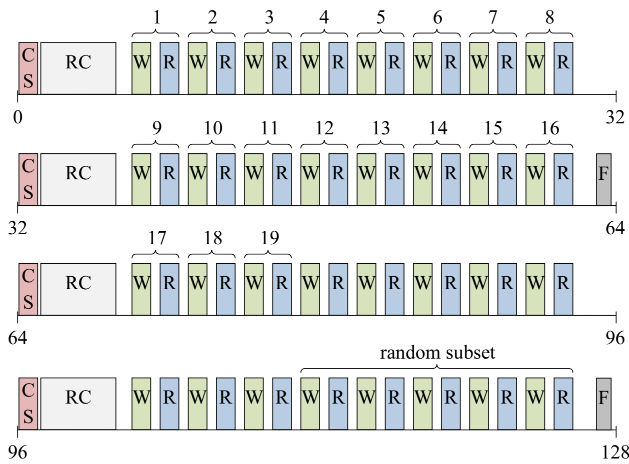

Figure 9: The test application to compare the wave signaling and the RSSI signaling allocates a total of 74 time slots within the scheduling window of 128 seconds. The following abbreviations are used for the modules: CS - Clock Synchronization, RC - Remote Control, F - Feedback composition, W - wave signaling, and R - RSSI signaling. The numbers above the time slots indicate the ID of the node which triggers an alarm.

allow for large networks with a diameter up to 120 hops if the root node is chosen in the center.

Whenever `startSlot()` is called, the signaling module retrieves the node's hop-count to the root from the clock synchronization module and determines its start-time in the any-to-root and the root-to-all pipeline. An adaptation of this hop-count is however necessary, as the links used by the synchronization tree may be unidirectional. This adaptation is implemented by a simple online search that is launched whenever the root-to-all sweep does not reflect a request sent on the any-to-root sweep.

7.4 Test Application

The integration of the signaling module into an application, in our case a test application to compare the RSSI and the wave signaling, is straight forward. All we need to do is to assign time slots to each module such as to obtain a non-overlapping schedule. In our test application, we utilize the following software modules:

- Clock synchronization module
- RSSI signaling module
- Wave signaling module
- Remote control module
- Feedback composition module

The remote control module is used for collecting the logs of an ongoing experiment and for debugging the application. The module supports multi-hop message forwarding to deliver log messages at a base station such that we can track the progress of an ongoing experiment. In addition, it can also route command messages from the base station to any given node in the network, which was convenient to debug the application in its initial phase. The feedback composition module collects and preprocesses the local data from the experiment and feeds the remote control module with the log messages. Both of these modules support our experiments, but they do not influence the synchronization and signaling tasks in any way as they are temporally decoupled. Therefore, they could be removed from the application without consequences for the other modules.

The test application uses an allocation window of 128 seconds within which it schedules its modules, see Figure 9. The clock synchronization module is scheduled every 32 seconds, followed by the remote control module that gathers the log messages. Every 64 seconds, the feedback composition module prepares the log messages which are sent in the forthcoming remote control time slot. The RSSI signaling module and the wave signaling module are assigned a total of 64 time slots within the remaining allocation window to test as many signaling rounds as possible. The two approaches are tested in an alternating fashion such that they undergo similar external influences.

Please note how simple the composition of our test application is. All we need to do is to schedule the desired modules. Thanks to the temporal separation, we know that there will be no interference between the different time slots. Consequently, the results obtained from our test with many signaling slots are also valid if the signaling slot is scheduled less often or in a different context. And that is exactly where the strength of the slotted programming lies: After testing a given module, the module can be reused in other contexts without requiring further test cases. With traditional programming approaches, however, any new composition of software modules needs to be retested from scratch to ensure that the modules do not interfere or produce undesirable side effects.

7.5 Deployment and Experiment Setup

We performed our experiments with the TinyNode 584 [20]. This sensor node features an MSP430 CPU with 10kB RAM and 48kB program space and works with the Xemics XE1205 radio which applies an FSK modulation. For receiving data packets, the radio is configured in a buffered mode with a built-in bit synchronizer and a pattern detector. As the bit synchronizer requires at least some bit transitions every now and then to operate correctly, we cannot use the buffered mode for detecting alarms, as they consist of a wave of a given frequency. We therefore use the continuous mode of the radio which skips the bit synchronization and pattern recognition and outputs the raw bit sequence on a data pin. Our application samples this stream during 5ms to obtain the described sampling. Whereas we could transmit waves in the buffered mode by sending a message with every bit set to 1 (or 0), this requires quite some modification on the radio stack. In continuous mode, the transmission of a wave is straight forward, as the radio sends the pattern that is applied to a given pin. Keeping the pin at 1 (or 0) results in the desired wave. For our experiment, we have deployed a total of 19 nodes, 6 of them are outside the building, see Figure 10. Node 1 acts as base station and as root for the clock synchronization. The goal of the experiment is to test the reliability of the proposed wave signaling and compare

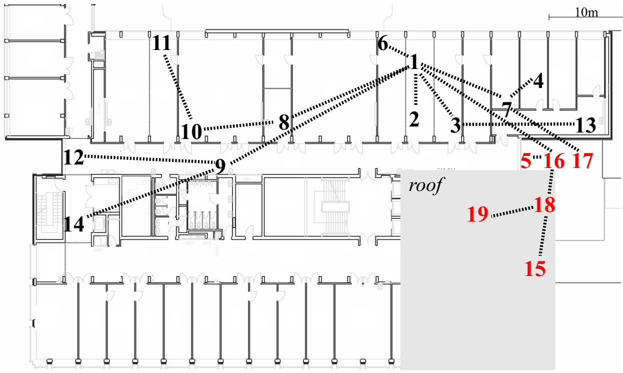

Figure 10: The nodes 5, 15, 16, 17, 18, and 19 are placed outside the building and are powered by battery. Whereas the nodes 15, 18, and 19 are on the roof, the nodes 5, 16, and 17 are placed on the facade along the four floors to the roof to establish a radio connection with the nodes on the roof. Node 1 is configured to be the root for the clock synchronization and the signaling. The dashed lines show an instance of the dynamic clock synchronization tree.

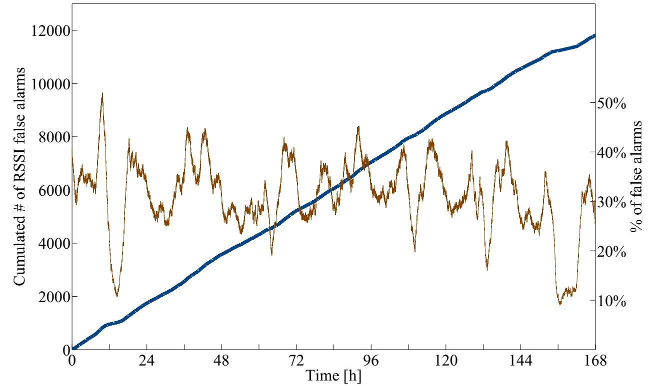

Figure 11: The solid line shows the cumulated number of false positives when using RSSI sniffs (left scale). The bumpy curve depicts the temporal variability of the error probability (using the right scale).

it to the RSSI approach. In particular, we are interested in the following two errors: A *false negative* signaling happens when a node fails to detect a signal sent by a neighboring node. This type of error is especially undesirable in alarm systems where the system should immediately react to a new situation. A *false positive* signaling occurs when a node detects a signal even though no node is transmitting. This fault is often called a false alarm.

To test for these errors, the following scenario is run through in each window of the slot scheduler, Figure 9 depicts the setup.

- Exactly one node triggers an alarm in the first 38 time slots. Each node is assigned a time slot pair (consisting of a time slot for the wave signaling and one for the RSSI signaling) during which it triggers an alarm. This is done by sending a wave or an message, respectively, at the assigned time within the any-to-root pipeline.
- None of the nodes triggers an alarm during the following 16 time slots.
- A randomly chosen subset of the nodes triggers an alarm in the remaining 5 time slot pairs. This last test case is used to explicitly examine the decoding resistance when several nodes signal synchronously.

For each of these time slots, we decide whether the signaling was successful or not. As the signaling module described in Section 7.3.1 performs an any-to-root sweep followed by a root-to-all sweep, all nodes are supposed to have the same state at the end of the signaling time slot. Therefore, if one or several nodes fail to detect the signal in the root-to-all sweep, we count the time slot as false negative. Similarly, if one or several nodes detect a signal in one of the 16 silent time slots, we count the time slot as false positive. Please note that we test the overall signaling procedure rather than each individual signal decoding as the latter is prone to report subsequent errors.

7.6 Results

We have run the above experiment for 168 hours. During this time, the schedule of Figure 9 was repeated 4725 times, generating 113400 rounds in which at least one node triggered an alarm and 37800 silent rounds without alarm. Table 1 summarizes the overall outcome of the experiment. On average, the RSSI had false alarms in 31% of the silent rounds, compared to only 0.8% of the wave approach. The probability for a dropped alarm (false negative) is comparable for both the wave and RSSI approach.

Table 1: Quality of the wave and the RSSI signaling.

	Wave signaling	RSSI signaling	Number of tests
False positives	304 (0.8%)	11815 (31%)	37800
False negatives	94 (0.08%)	132 (0.11%)	113400

Figure 11 depicts the cumulated number of RSSI false alarms, and the horizontal curve shows how the error probability changes over time. Whereas RSSI suffers from false alarms in 31% on average, the curve shows that the error probability is sometimes as low as 10% for an extended period of time. This fact can also be observed on the cumulative curve, where the slope flattens temporarily during the early afternoon of the first and last day of the experiment. (Please note that all plots start at midnight.) We belive that this improvement could be related to the weather, as only the first and last day of the experiment was sunny.

8. CONCLUSION

The advantages of the slotted programming paradigm are not only in the facilitated software development, just as important are the modularity and temporal separation of the different tasks which allow for a predictable execution of the application. This predictability is a basic requirement to build systems for which certain properties can be guaranteed. In fact, the development of provably correct software is very hard in general. The core difficulty lies in

the fact that it is hard to compare the application (written in a programming language) and the specification (written in a descriptive language). The slotted programming paradigm cannot solve this problem, but it shortens the gap between the two worlds by reducing the complexity of the software implementation, rendering the comparison easier. Most importantly, the slotted programming achieves this by temporally decoupling independent tasks such that the different components can be checked separately. In addition, the temporal decoupling renders unnecessary many asynchronous code blocks as timely execution is ensured implicitly by the slotted execution model. Such synchronous code is much easier to analyze as there are much fewer execution patterns to be considered and the simplified code itself reduces the risk of bugs.

As of now, slotos does not enforce the modules to perform their operations within the assigned time slots. In future work, we wish to extend slotos such that modules cannot execute code outside their assigned slots such that we can guarantee a smooth operation even in the presence of a misbehaving module.

9. ACKNOWLEDGMENTS

We would like to thank Jan Beutel, Nicolas Burri, Roman Lim, Philipp Sommer, and Pascal von Rickenbach for sharing with us their insights in the programming of wireless sensor networks. Also, we would like thank our reviewers for their comments that helped to improve this paper.

10. REFERENCES

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support for Multimodal Networks of In-Situ Sensors. In *WSNA*, 2003.

[2] N. Brouwers, P. Corke, and K. Langendoen. A Java Compatible Virtual Machine for Wireless Sensor Nodes. In *SenSys*, 2008.

[3] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In *IPSN*, April 2007.

[4] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient Memory Safety for TinyOS. In *SenSys*, 2007.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Emnets*, 2004.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *SenSys*, 2006.

[7] A. El-Hoiydi and J.-D. Decotignie. WiseMAC: An Ultra Low Power MAC Protocol for the Downlink of Infrastructure Wireless Sensor Networks. In *Computers and Communications*, 2004.

[8] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *RTSS*, 2005.

[9] L. Gu and J. A. Stankovic. t-kernel: Providing Reliable OS Support to Wireless Sensor Networks. In *SenSys*, 2006.

[10] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *MobiSys*, 2005.

[11] T. Harbaum. NanoVM. `http://www.harbaum.org/till/nanovm`, March 2009.

[12] J. W. Hui and D. E. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *SenSys*, 2008.

[13] E. Lee. Computing Needs Time. *Communications of the ACM*, 52(5):70–79, 2009.

[14] C. Lenzen, P. Sommer, and R. Wattenhofer. Optimal Clock Synchronization in Networks. In *SenSys*, November 2009.

[15] K. Lorincz, B. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource Aware Programming in the Pixie OS. In *SenSys*, 2008.

[16] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The Flooding Time Synchronization Protocol. In *SenSys*, 2004.

[17] W. P. McCartney and N. Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *SenSys*, 2006.

[18] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *SenSys*, 2004.

[19] Sentilla. Sentilla Perk. `http://sentilla.com/perk`, July 2009.

[20] Shockfish SA. TinyNode. `http://www.tinynode.com`, November 2008.

[21] Y. Sun, O. Gurewitz, and D. B. Johnson. RI-MAC: a Receiver-Initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks. In *SenSys*, 2008.

[22] TinyOS Alliance. TinyOS. `http://www.tinyos.net`, July 2009.