

Semester Thesis SS06

Enhanced Task Scheduling in TinyOS 2.0
&
Channel Allocation in AMUHR

Raffael Bloch
blochra@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Pascal von Rickenbach, Nicolas Burri

Contents

1 Introduction	3
2 Enhanced Task Scheduling in Tiny OS 2.0	3
2.1 Tiny OS 2.0 Tasks & Scheduler	3
2.1.1 nesC Tasks	3
2.1.2 Scheduler	4
2.2 Implementing Priority Tasks	4
2.3 Analysis of the TinyOS 2.0 Concurrency Model	6
2.4 Conclusions	6
3 Efficient Data Transfer in AMUHR	7
3.1 AMUHR	7
3.2 Channel Leasing System	8
3.2.1 Local Cache	9
3.2.2 Data Transfer	9
3.3 Future Work	10
3.4 Conclusions	11
Appendix A	12
A.1 Wiring and Usage of AMUHR	12
A.2 Configuring the Cache	13
References	14

1 Introduction

This semester thesis consists of two independent parts. In the first part we study the inbuilt scheduler of TinyOS 2.0 and discuss its advantages as well as its limitations. We show how the scheduler can be extended at the example of a priority-based scheduler.

The second part however is an extension to previous work of David Landis, who wrote AMUHR (**A**nother **M**ulti-**H**op **R**outer for TinyOS). AMUHR is a multi-hop routing system based on the idea of Dynamic Source Routing, an algorithm which sends the routing information along with the payload in the same communication packet. Given the fact that the maximum packet size in the TinyOS system is limited to 29 bytes and a single path often consist of many hops, it becomes obvious that the available payload is very limited for the user of such AMUHR packets. We present a solution to this problem in the second part.

2 Enhanced Task Scheduling in Tiny OS 2.0

2.1 Tiny OS 2.0 Tasks & Scheduler

Traditionally, the scheduler of TinyOS follows a non-preemptive FIFO policy. According to the TinyOS concurrency model, all tasks run to completion and do not preempt each other. Waiting tasks are executed in the order of their arrival. As basic tasks are directly supported by nesC (the native language of TinyOS), we will refer to them as *nesC tasks*. We first need to understand how nesC tasks and the standard TinyOS scheduler work together before we can define a new task concept and modify the scheduler.

2.1.1 nesC Tasks

A task specification in nesC essentially is the specification of a function without arguments or return value. Unlike a function, a task can however not be called directly. To execute a task, we need to pass it to the scheduler using the keyword **post**.

A task specification in nesC:

```
void task mytask() {  
    // CODE  
}
```

and the corresponding post instruction:

```
post mytask();
```

Each component in Tiny OS 2.0 containing such a nesC task specification **uses** the TaskBasic interface. This means it implements the event runTask() which holds the

user's implementation of the task and can be signalled by the scheduler. The scheduler itself **provides** the TaskBasic interface and implements the command postTask(). All the proper wiring work is done automatically by the compiler.

```
interface TaskBasic {
    async command error_t postTask();
    event void runTask();
}
```

2.1.2 Scheduler

The scheduler of TinyOS 2.0 **provides** the Scheduler interface that offers three commands. Init() obviously initialises the scheduler during the boot sequence. runNextTask() executes the next task in the queue if there is one. Whether or not a task was found and executed, is indicated by the return value. Finally, taskLoop() can be called to enter in an infinite loop which continuously executes tasks as long as there are some waiting. Otherwise, if the task is empty, the CPU goes into a state of low power consumption. To enter and return from the sleep mode, the scheduler **uses** the interface McuSleep.

```
interface Scheduler {
    command void init();
    command bool runNextTask();
    command void taskLoop();
}
```

Using the TaskBasic interface of the previous section, we can summarise the complete signature of the TinyOS scheduler as shown below:

```
module SchedulerBasicP {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t taskID];
    uses interface McuSleep;
}
```

2.2 Implementing Priority Tasks

The first step in implementing a new kind of task is to define its interface analogously to the BasicTask interface we studied earlier. For our priority tasks, the interface could look as follows:

```
interface TaskPrio {
    async command error_t postTask(uint8_t priority);
    event void runTask();
}
```

The only difference to the TaskBasic interface forms the parameter priority which defines the priority level of the task. To make our newly defined tasks available to the programmer, we need to modify the standard TinyOS scheduler located in the file **TinySchedulerC.nc** in the **tinyos-2.x/tos/system** folder. This configuration file reveals that the standard scheduler's implementation can be found in the file **SchedulerBasicP.nc**. We replace this implementation file by our own MyScheduler.nc file supporting priority tasks. Hence, the new configuration file looks as follows:

```
configuration TinySchedulerC {
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t id];
  provides interface TaskPrio[uint8_t id];
}
implementation {
  components MyScheduler as Sched;
  components McuSleepC as Sleep;
  Scheduler = Sched;
  TaskBasic = Sched;
  TaskPriority = Sched;
  Sched.McuSleep -> Sleep;
}
```

For the implementation of the new scheduler, we can use the old implementation **SchedulerBasicP.nc** as a template. The only notable difference to the implementation of the TaskPrio interface is the replacement of the existing FIFO queue by a priority queue. Of course, we still need to support the nesC tasks by implementing the TaskBasic interface, as our scheduler would otherwise not work with the existing system. The new scheduler component looks like this (omitting the actual implementation):

```
module MyScheduler{
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t id];
  provides interface TaskPrio [uint8_t id];
  uses interface McuSleep;
}
implementation
{ ... }
```

The new priority tasks are now ready to be used by the programmer. Though, the specification and usage are slightly different to what we know from nesC tasks. The keyword task can no longer be used. Instead, we implement the event runTask() for every task we need in our program.

```
module MyApp {
  uses interface TaskPrio as MyPriorityTask;
}
implementation {
  event void MyPriorityTask.runTask()
}
```

Finally, we wire the **used** TaskPrio interface of our application to its counterpart **provided** by the scheduler. Note that if we are using more than one priority task, every single one of them has to be wired separately. The configuration file for our application looks like this:

```
configuration MyAppConfig
{
implementation
{
  { ... }
  components TinySchedulerC as Scheduler;
  { ... }
  MyApp.MyPriorityTask -> Scheduler.TaskPrio[unique("TS.TaskPrio")];
}
```

2.3 Analysis of the TinyOS 2.0 Concurrency Model

We have seen that the scheduler of TinyOS 2.0 can be modified very easily to support different kinds of tasks. On the one hand, the non-preemptive TinyOS concurrency model simplifies the handling of race conditions, deadlocks and other concurrency-related issues for the programmer. On the other hand, it also leads to serious problems in many applications as soon as long running tasks are involved. Even priority tasks may still be delayed, reducing the responsiveness of the system significantly. Another problem is the inconsistency between nesC tasks and the user-defined tasks. Having two different notations for quite the same thing, seems unnecessarily complicated. Though, this design decision is understandable considering that TinyOS 1.1 had no support for user defined tasks at all.

2.4 Conclusions

The concurrency model of TinyOS is best suited for simple, non time critical applications. Programmers may even circumvent the problem of non-preemptable tasks by splitting large tasks into smaller pieces, thereby reducing the maximum delay for incoming priority tasks. However, in the future, the computational capabilities of sensor nodes will certainly increase and their applications will probably become more complex, such that a change of the restrictive concurrency model would be advisable. I would also recommend giving the programmer more flexibility and thereby more responsibility for the concurrency issues. The current system with its advantages as well as its drawbacks may be preserved as an option offered to those users, who want to benefit from “easy-to-use” concurrency and the automatic conflict analysis.

3 Efficient Data Transfer in AMUHR

3.1 AMUHR

The basic communication pattern of AMUHR consists of two subsequent phases. In the first phase, a node which intends to communicate with another node, broadcasts a *find* message containing his communication partner's address. Such *find* packets are rebroadcasted by the neighbours if they received the message for the first time. This process continuous until the message reaches the actual recipient (Flood Search). Thereby, the addresses of the intermediate hops between sender and receiver encountered during the search process are stored in the find packet. The receiver so gets all information he needs to send a *returnpath* message back to the sender, along the just discovered path. As this returnpath message contains the entire hop sequence, both sender and receiver can store the path which allows the bidirectional communication between two of them.

Once a path between two nodes is found, DATA packets can be exchanged between sender and receiver in a second phase. Note that such DATA packets always need to contain the routing/path information besides the actual payload, because the nodes along the path do not have any routing information. If the user has activated explicit acknowledgements, the receiver of a data packet sends an ACK packet back to the sender. **Figure 1** shows the cascade of messages triggered by node 1 which attempts to send a data packet to node 4.

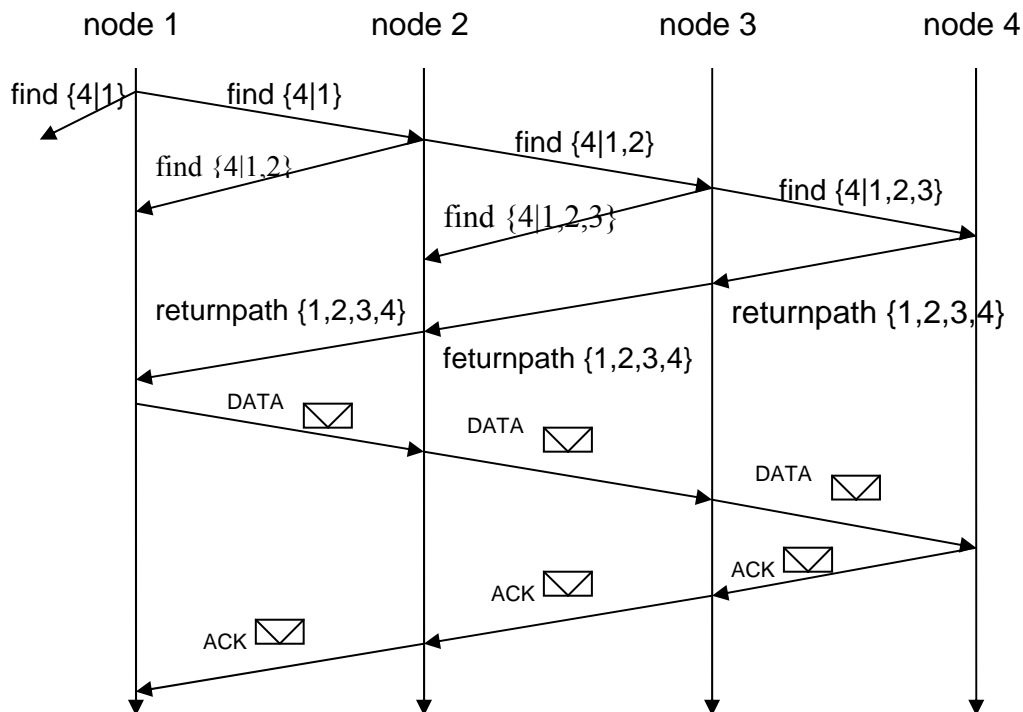


Figure 1 Two-phase communication pattern of AMUHR. We assume that only neighbouring nodes can communicate with each other.

The user of AMUHR can choose from various configurations that influence the details of the two-phase communication process described above. There are for example implicit and explicit acknowledgements available. By explicit acknowledgements, we mean end-to-end acknowledgements, whereas implicit acknowledgements assume that all radio links are bidirectional and that a sender of a packet can hear the transmissions of its neighbours. A node checks whether or not the neighbouring node forwards a specific packet and initiates a resend if necessary. A more detailed introduction to AMUHR can be found in [1].

Depending on the length of the path (number of hops) and the configuration of implicit and explicit acknowledgements, the payload per data packet can be relatively small. Even though the system is quite useful for applications which only transfer small amounts of data, it was considered necessary to introduce a more efficient way of data transfer between nodes. By introducing communication channels, the available payload per packet should be substantially increased. Such channels are ideally used if large amounts of data have to be transferred between nodes. A possible application may be the transfer of entire buffers or caches to a master node for the purpose of their analysis. In the next section, we will look at the basic idea of Channel Leasing and the corresponding implementation in AMUHR. We will then have a look on how AMUHR could be improved in the future. Appendix A holds a number of useful information on how to wire AMUHR into your applications. The possible configurations of the newly introduced channels are also listed in the appendix.

3.2 Channel Leasing System

The basic idea of communication channels in AMUHR is to move the path information from the data packet to the nodes along the path. Once the routing information is locally stored, the available payload per packet is independent of the path's length. One important goal of the implementation is the reuse of as much of the existing functionality as possible to avoid unnecessarily increasing the code size. Another goal was to make the management of the channels transparent for the user. In fact the user can choose between the traditional Source Routing style of communication and the new channel-based style, by simply setting the `options` parameter in the standard AMUHR `send` command.

```
command result_t send(uint_16_t receiver,
                    uint8_t msg_length,
                    TOS_MsgPtr msg,
                    uint8_t options);
```

Using `options = MH_USER_LOCAL_PATH` activates a communication channel between the calling node and the `receiver` node. There are other flags which activate explicit and implicit acknowledgements. More details on AMUHR's `send` command and valid combinations of flags can be found in Appendix A and in [1].

Since AMUHR already has the capability to find paths, we directly use the existing functionality to allocate a cache entry on every node along the path. This is done every

time a *returnpath* message arrives at a node (**Figure 1**). Once a channel has been established between two nodes, it can be used for efficient data transfer in both directions. The leasing system assures the availability of channels while they are in use. In the following sections, we will first have a look at how the cache is organised and how the leasing system works, before we focus on how data is transferred using channels.

3.2.1 Local Cache

The cache structure which is needed to store the route information locally on the nodes, is located in the file `RouteCacheLocalPathM.nc`. Every cache line holds the addresses of the start and end nodes of the channel, as well as the next and previous node address. An additional byte holds a time stamp, which represents the age of this particular cache line. The example cache line in **Figure 2** is stored on node 4 of the path $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$. If a message with destination node 6 arrives, the cache line instructs us to forward the message to node 5. If the destination node is 2 we send the message to the previous hop 3. Note that this cache line holds sufficient information to operate the channel between nodes 2 and 6 in both directions.

src_addr	dest_addr	nxt_hop	prv_hop	age
2	6	5	3	10

Figure 2: Example cache line on node 4

On every node, a task is executed periodically to reduce the value of the age counter. Once the value of the counter reaches 0, the entry is treated to be out of date and may be overwritten if a new cache line has to be allocated by the system. Consequently every access to a cache entry resets the counter to its maximum value, which can be configured along with the number of cache lines by the user. This procedure ensures that no entry that has recently been used will be overwritten. Therefore, established channels stay available as long as they are frequently used. A longer traffic-free period on the channel disposes the occupied cache line. Appendix A gives more information on how to configure the cache.

This channel leasing system automatically manages its resources and therefore prevents the user from manual task allocation and de-allocation, which often lead to problems due to resource mismanagement.

3.2.2 Data Transfer

Once a channel is established the user can send and receive data packets (**Figure 3**) of a fixed size efficiently. The maximum payload is 23 bytes per packet. In addition to the data, a packet holds a header consisting of the sender and receiver addresses as well as of a type field which identifies the packet's type (AMUHR supports many different packet types which are all described in [1]). Finally the packet ID field is used to identify resends of the same packet as well as acknowledgements.

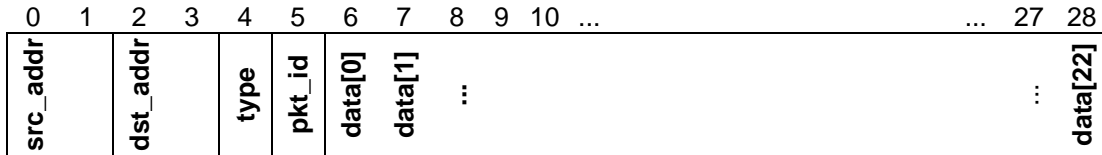


Figure 3: Data packet layout.

Received data packets are always stored in a packet history to avoid delivering a packet twice. This mechanism assures that even if a packet is sent multiple times, it is delivered only once. Such a situation may occur if an acknowledgement message gets lost due to transmission errors. An acknowledgement packet only consists of the addresses of the sender and receiver of the packet as well as the type field and of course the packet's ID to identify the acknowledged data packet. Details are shown in **Figure 4**. Note that the data packet and the acknowledgement packet described in this section are new packet types which have been introduced to work with channels. They significantly differ to the data and acknowledgement packets used in Source Routing communication.

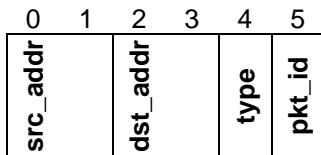


Figure 4: Acknowledgement packet layout.

3.3 Future Work

To further improve the functionality and usability of AMUHR I would suggest addressing the following problems:

- AMUHR offers a large variety of configurations. Such flexibility comes at the cost of an increasing code size. This problem is particularly bad since AMUHR is currently designed as one monolithic component. The system always has to be entirely loaded, even if only a fraction of its functionality is actually used. I therefore recommend a complete redesign of AMUHR by focussing on modularisation of the code. This would not only simplify future extensions of AMUHR but would also address the increasing memory usage problem.
- Extensive testing of AMUHR's various configurations under real world conditions is necessary to detect and fix bugs. For this purpose, AMUHR should offer an interface to the serial port of a sensor node.
- A lot of the program code is heavily compiler-dependant. The extensive use of pointer arithmetics for memory addressing should be replaced by normal accesses to proper data types.

- For real world applications, aspects of security and reliability are very important. Perhaps, the communication protocols could be improved to match higher expectations.
- Implicit acknowledgements can currently not be combined with channels. This is mainly to avoid a further increase of the code size. During a redesign of the system this restriction could be easily removed to enable full functionality.

3.4 Conclusions

In its current version, AMUHR is a multi-hop router which is easy to deploy. The system offers a large variety of configurations making AMUHR easily adaptable to different application scenarios and network environments. The new channel-based communication scheme fits well in the existing system without requiring changes to its user interface. The complexity of channel management is thereby completely hidden from the user. Without any modifications, all applications working with the old version of AMUHR should also run with the extended version.

Appendix A

In a first part, this appendix gives a short introduction on how to wire AMUHR in your application as well as how to use its features. The second part then explains how to configure the cache in a way that serves your application best.

A.1 Wiring and Usage of AMUHR

The communication between your application and the AMUHR component takes place through the interfaces `SendMsgEx` and `ReceiveMsg`. Both of these interfaces are **provided** by AMUHR and your application should therefore **use** both of these interfaces.

```
interface SendMsgEx {
    command result_t send(    uint16_t address,
                             uint8_t length,
                             TOS_MsgPtr msg,
                             uint8_t options);
    event result_t sendDone(TOS_MsgPtr msg, result_t success);
}

interface ReceiveMsg {
    event TOS_MsgPtr receive(TOS_MsgPtr msg);
}
```

Using `SendMsgEx` and `ReceiveMsg` interfaces in your application requires you to implement the two events `sendDone` and `receive`. The skeleton of your application then looks as follows:

```
module MyApp {
    uses {
        interface SendMsgEx;
        interface ReceiveMsg;
    }
    provides {...}
}

implementation {
    event result_t SendMsgEx.sendDone(TOS_MsgPtr msg, result_t success) {...}
    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msg) {...}
}
```

The last step involves the actual wiring of both interfaces to AMUHR which is done in a separate configuration file shown below.

```
configuration MyAppConfig {}
implementation {
    components AMUHR;
    MyApp.SendMsgEx -> AMUHR.SendMsgEx[MSG_TYPE];
    MyApp.ReceiveMsg -> AMUHR.ReceiveMsg[MSG_TYPE]; }
```

Note that both interfaces have to be parameterised by the message type intended to be sent and received. A detailed explanation of parameterised interfaces and the wiring mechanism of nesC in general can be found in [2].

Now that we have properly wired our application to AMUHR, we may start sending messages through the `send` command provided by the `SendMsgEx` interface. A typical call goes as follows:

```
call SendMsgEx.send(receiver,msg_length,msg,options);
```

Parameters:

- `uint16_t receiver` the address of the receiver
- `uint8_t msg_length` the length of the message in bytes
- `TOS_MsgPtr msg` pointer to the message
- `uint8_t options` options

Options: (AMUHR/MHConstans.h)

<code>MH_USER_WANT_ACK</code>	Activates end-to-end acknowledgements
<code>MH_USER_IMP_DATA_ACK</code>	Activates implicit ack's for data packets
<code>MH_USER_FIND_IMP_ACK</code>	Activates implicit ack's for find packets
<code>MH_USER_LOCAL_PATH</code>	Activates a communication channel

IMPORTANT:

The combination of channels and implicit acknowledgements is currently not supported.

The `send` command just delivers the message to the send queue of AMUHR. In case you activated acknowledgements the `sendDone` event is signalled after successful reception of the acknowledgement. Otherwise the `sendDone` event is signalled as soon as the message has actually been sent over the radio.

A.2 Configuring the Cache

There are three important settings that influence the behaviour of the route cache. All of them can be found in the file `AMUHR/MHSettings.h`.

- **`ROUTE_CACHE_LOCAL_PATH_CHECK_INTERVAL = 5000`**; This constant determines the interval in which the age counter will be decremented in milliseconds. The standard value is 5000.
- **`ROUTE_CACHE_LOCAL_PATH_NUM_ENTRIES = 10`**; The number of entries the cache can hold. This value directly influences the memory consumption. Each cache line occupies 9 bytes of memory. The standard value of 10 cache lines therefore uses 90 bytes of RAM on each sensor node.

- **ROUTE_CACHE_LOCAL_PATH_MAX_AGE = 20;** The maximum value the age counter can have. When the counter reaches 0, the cache line can be overwritten and the corresponding channel becomes invalid. The product of the check interval and the maximum age counter define the longest idle time a channel is guaranteed to stay available.

The correct configuration of the cache is crucial for the successful usage of channels in your application. You need to know the usage pattern of your application to make good decisions. If your application is engaging only a small number of channels between key nodes, it may be effective to have long living channels even if they are only seldomly used. In such a case you might want to increase the check interval to avoid unnecessary computations. If your application needs your sensor nodes to establish a lot of channels to changing destinations, you should increase the size of the cache and decrease both the check interval and the age counter's maximum size.

References

- [1] D. Landis. Multi-Hop Routing for Wireless Sensor Networks. *Semester Thesis*, Distributed Computing Group, Department of Information Technology and Electrical Engineering, ETH Zurich, March 2006.
- [2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D.Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.