Semester Thesis WS06

Multi-Hop Routing for Wireless Sensor Networks

David Dominic Landis
dlandis@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Nicolas Burri, Pascal von Rickenbach

AMUHR - Another Multi-Hop Router

2nd April 2006

Contents

1 Introduction

AMUHR (**A**nother **Mu**lti-**H**op **R**outer) enables TinyOS motes to exchange messages among each other even if they are several hops apart. This document describes how to deploy AMUHR and provides background information on the implementation.
To get familiar with TinyOS please visit http://www.tinyos.net/

1.1 Working Plan

The goal of this semester thesis is to provide a generic multi-hop routing module for TinyOS that is tested with TOSSIM and considers hardware limitations of mica2 motes. First a simple DSR is to be implemented which then is to be optimized.

1.2 DSR

Probably everybody has heard of DSR. Therefore this section is kept short. There are to phases: route discovery and route maintenance. During route discovery the initiator broadcasts its request to find a route. Nodes that receive this request rebroadcast it after appending themselves to the path that leads back to the initiator. When the target is found the path is sent back to the initiator which then stores it in its route cache. During maintenance initiators have to assure that paths are valid.

1.3 Feature Overview of Optimized Module

To go multi-hop with an existing application and AMUHR, only a few lines of code have to be written. Many compile time constants are provided to allow fine tuning of routing parameters such as optimizing for dynamic or static networks or adjusting the minimum and maximum ttl to decrease path resolution time.
Because the standard packet length is very small (29 bytes) the path often doesn't fit into the packet along with the desired payload. Therefore a compile time maximum packet length has to be chosen and is guaranteed to be available independently of the path length. This is achieved by letting intermediate nodes store and complete missing parts of the route.
Some applications may not care if a few packets are lost and others may need reliable data transfer. Therefore the possibility to request an end-to-end acknowledgment is provided. For lossy networks implicit acknowledgments can be activated on a per packet basis to improve the chance of successful transmissions.
Since sending packets and resolving paths to distant motes takes an significant amount of time pipelining is fully supported.
Some mote types and some TinyOS versions support a link-layer ack field in TOS_Msg structure which indicates whether a message was successfully received by at least one receiver. AMUHR can be set to use this field and get a performance gain out of it.
For more information about optimizations chapter 6.6 *Implemented Optimizations* can be consulted.

2 Quick Start

Step 1: Installing

Extract AMUHR.tar.gz and read the *readme* file. It is recommended to to put the directory AMUHR in a subdirectory of your application because you'll modify some parameters during optimization. (→ to get more information about optimization read chapter 4 *A Step by Step Configuration*)

Step 2: Wiring

To use the multi-hop protocol following interfaces are to be connected:

- AMUHR.StdControl
- AMUHR.SendMsg (or AMUHR.SendMsgEx)
- AMUHR.ReceiveMsg

The example shows the wiring for an application called MHBlink with module MHlinkM.

```
configuration MHBlink {
}
implementation {
  components Main, MHBlinkM, AMUHR as Comm;

  Main.StdControl -> MHBlinkM.StdControl;
  Main.StdControl -> Comm.Control;
  MHBlinkM.Send   -> Comm.SendMsg[AM_INTMSG];
//  MHBlinkM.SendEx -> Comm.SendMsgEx[AM_INTMSG];
  MHBlinkM.Rcv    -> Comm.ReceiveMsg[AM_INTMSG];

}
```

Step 3: Sending/Receiving

To send messages use SendMsg.send:

```
  //result_t SendMsg.send(uint16_t address, //destination
                          uint8_t length,   //message length
                          TOS_MsgPtr msg)   //ptr to message

  if(!call Send.send(addr, len, &message)) {
    //sending failed or some queue is full
  }
```

Note: `sendDone` is called as soon as the packet is sent if no acknowledgment was requested otherwise it waits until the packet is acknowledged or a timeout occurs.

There is a second interface AMUHR.SendMsgEx which has an additional option parameter:

```
uint8_t options = MH_USER_FIND_IMP_ACK |
                  MH_USER_IMP_DATA_ACK |
                  MH_USER_WANT_ACK;

//result_t SendMsg.send(uint16_t address, //destination
                        uint8_t length,   //message length
                        TOS_MsgPtr msg,   //ptr to message
                        uint8_t options)  //options

if(!call Send.send(addr, len, &message, options)) {
  //sending failed or some queue is full
}
```

There are four supported constants for the option parameter that affect the way packets are sent. (To get an idea of the packet types refered to, take a look at the picture in chapter 3.1.)

MHSettings.h

- `MH_USER_FIND_IMP_ACK`: use implicit acknowledgments for find and return packets
- `MH_USER_IMP_DATA_ACK`: use implicit acknowledgments for data and response packets
- `MH_USER_WANT_ACK`: the receiver sends an acknowledgment after every data packet arrival
- `NO_IMPL_ACK_FOR_FIND_PACKETS` (static, cannot be used as send parameter): only use implicit acks for return packets and not for find packets

MHDefines.h

- `#define USE_TOS_ACK 1`: (static) use link layer ack field. Please read the hints in file `MHDefines.h` before enabling.

Supported combinations are:

- `//#define USE_TOS_ACK 1` deactivated
  - `MH_USER_FIND_IMP_ACK`
  - `MH_USER_FIND_IMP_ACK | MH_USER_WANT_ACK`
  - `MH_USER_FIND_IMP_ACK | MH_USER_IMP_DATA_ACK | MH_USER_WANT_ACK`
  - `MH_USER_WANT_ACK`
  - `0`
- `#define USE_TOS_ACK 1` activated
  - `MH_USER_WANT_ACK` must be set! (otherwise the packets lack of an id and could arrive several times)

To receive messages the following event handler is used:

```
event TOS_MsgPtr Rcv.receive(TOS_MsgPtr msg) {
  //do something
  return msg;
}
```

Step 3: Makefile

Add the path of AMUHR to the Makefile (where */YOURLOCATION* should be replaced with the location used in step one).

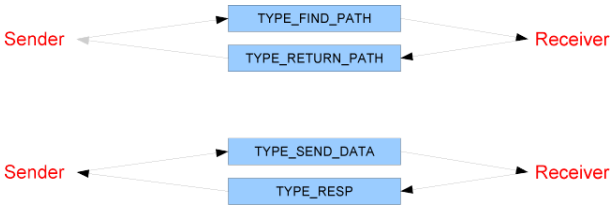
```
PFLAGS += -I/YOURLOCATION/AMUHR
```

Step 4: Tuning

To learn how to adapt MHSettings.h please read chapter *A Step by Step Configuration*.

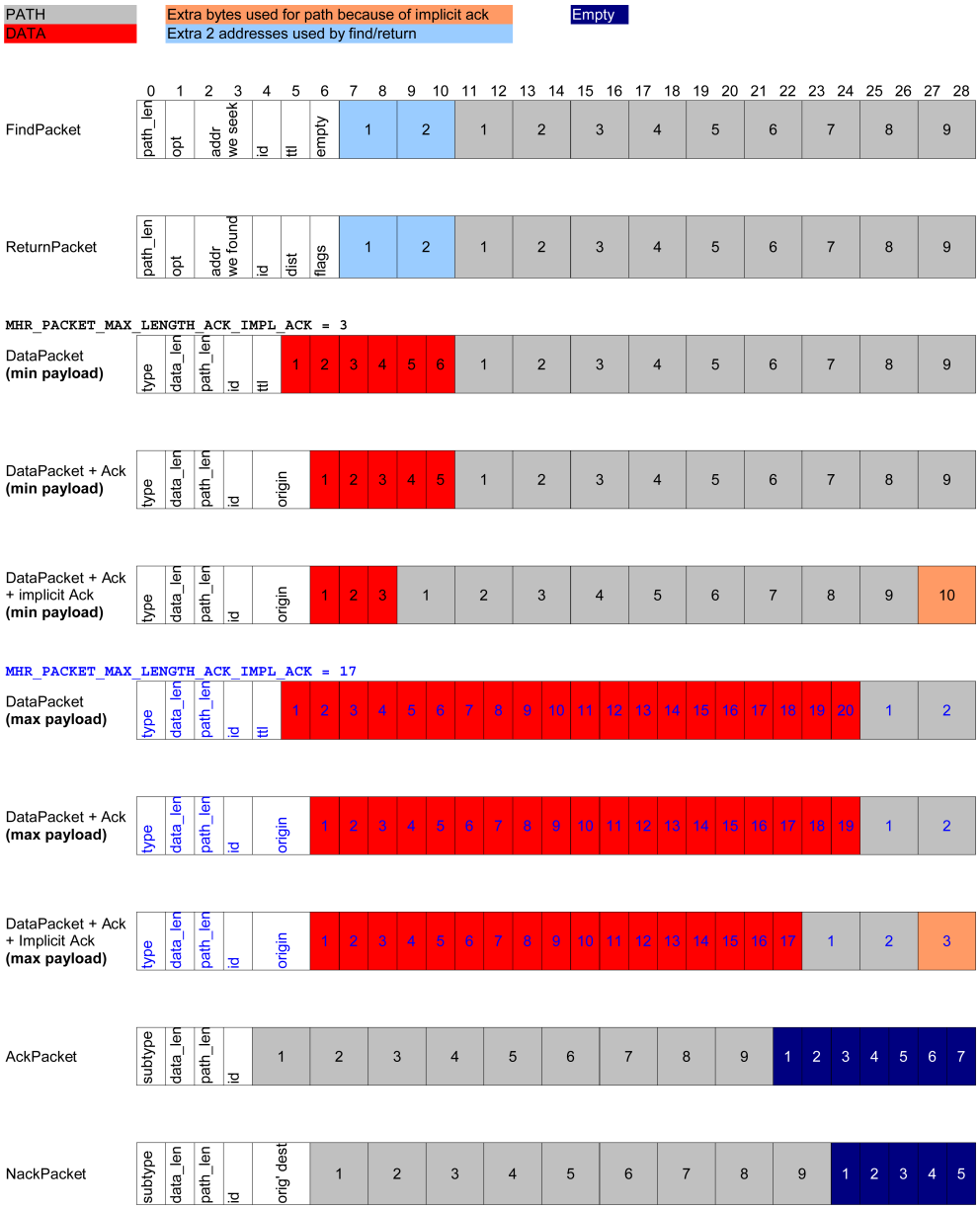
3 Routing

3.1 Packet Overview



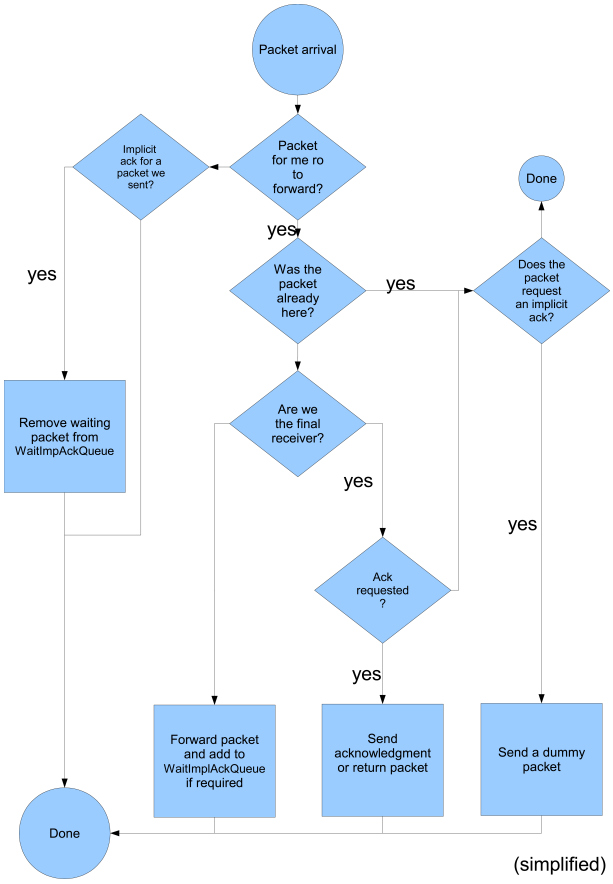
- A *find path packet* is sent if we don't know the route.
- A *return path packet* is the response to a *find path packet*.
- A *data packet* is used to wrap the message the user wants to send.
- A *resp packet* contains an ack or nack. *Resp packets* are not sent unless the sender requested an acknowledgment.

Below there is a detailed overview of the different packet types and their payload sizes. An example of a packet with minimum and maximum payload size based on default `TOSH_DATA_LENGTH` (=29 bytes) is given. To alter the maximum payload length the enum `MHR_PACKET_MAX_LENGTH_ACK_IMPL_ACK` in the file *MHSettings.h* can be changed.


PATH
DATA
Extra bytes used for path because of implicit ack
Extra 2 addresses used by find/return
Empty
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
FindPacket
path_len
opt
addr we seek
id
ttl
empty
ReturnPacket
addr we found
dist
flags
MHR_PACKET_MAX_LENGTH_ACK_IMPL_ACK = 3
DataPacket (min payload)
type
data_len
origin
DataPacket + Ack (min payload)
DataPacket + Ack + implicit Ack (min payload)
MHR_PACKET_MAX_LENGTH_ACK_IMPL_ACK = 17
DataPacket (max payload)
DataPacket + Ack (max payload)
DataPacket + Ack + Implicit Ack (max payload)
AckPacket
subtype
NackPacket
orig' dest

3.2 Packet arrival

Below the arrival of a routing packet (find,return,data,response) is shown.



3.3 Route Cache

Every mote contains a route cache that consists of entries similar to these:

```
Route Cache Node 0:
===================

Route 0: occupied:1 age: 2 len: 2 shortened: 0 route: {1, 5}
Route 1: occupied:1 age: 6 len: 4 shortened: 1 route: {1, 2,
                                                       3, 9}
 ...
```

Route 0 contains the path to mote 5: {0, 1, 5}.
Route 1 ought to contain the path to mote 0: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, but since there is not enough space to store the whole route *shortened routes* were introduced. A *shortened routes* contains the beginning of a path and the destination node and is marked with a flag. In the case above node 3 and node 6 must provide missing parts.

3.4 FAQ

- **Is it possible that packets arrive multiple times?**
  If there is enough memory and correct settings were used the answer is no otherwise following facts apply:
  - The enum value `DATA_PACKET_HISTORY_SIZE` defines how many data packets with id (8 bit) and sender (16 bit) can be kept for comparison. The oldest entry is overwritten if too many packets arrive.
  - The enum value `DATA_PACKET_HISTORY_MAX_AGE` defines a maximum age for an entry in the data packet history. After `(DATA_PACKET_HISTORY_MAX_AGE * DATA_PACKET_HISTORY_CHECK_INTERVAL_TIME)` ms the entry is deleted.
  - If implicit acknowledgments are used and the sender of a packet does not hear the receiver forwarding the packet it is sent again which might result in packets arriving twice if `DATA_PACKET_HISTORY_SIZE` is too small.
- **Although the history size is large enough packets still arrive twice!**
  Enabling `USE_TOS_ACK` without `MH_USER_WANT_ACK` is not allowed. (Because packets lack of an id they are indistinguishable and are forwarded every time they are received.)
- **If event sendDone signals SUCCESS can I be sure that the packet arrived?**
  If SendMsg or SendMsgEx interface was used with options `MH_USER_WANT_ACK` you can be sure. Otherwise SUCCESS means that a route to the destination was found and the packet was sent to the next hop.
- **If event sendDone signals FAIL can I be sure that the packet did not arrived?**
  No you can't. Maybe the route was not found or the data or acknowledgment packet were lost or a timeout occurred when finding the route or sending the data - all this is indicated by FAIL. `IMPL_ACK_TIMEOUT`, `TTLX_WAIT` and `TTLX_SCALE` influence waiting times.
- **I noticed some find packets being passed forward and backwards?!?**
  This might occur if you set `PACKET_HISTORY_SIZE` too small and too many find packets pass through a node at the same time. To prevent this increase the size of `PACKET_HISTORY_SIZE`. There is a ttl, so they will eventually die.

4 A Step by Step Configuration

Our goal is to build a 5x5 network grid and run it in TOSSIM. We will use cygwin syntax because it seems to be widely used. Please read also the comments in *AMUHR/MHSettings.h* - there are a lot of additional hints and explanations on optimization and the meaning of each constant!

1. Create a network layout for TOSSIM with LossyBuilder. The name should be 5x5.nss. Then convert the file to assure that only bidirectional and symmetric routes exist. You can use a tool in developer directory called *lossyConverter*.cpp*. The output should be 5x5.nss.bi. The comments in the c++ source files will provide information on compiling and running.

2. Open `MHSettings.h`. That's were the parameters can changed.

3. What is the maximum packet length used? What is the average path length we expect? Based on the previous two questions the value of `MHR_PACKET_MAX_LENGTH_ACK_IMPL_ACK` should be chosen. The bigger the maximum packet length the smaller is the route that can be stored in a data packet and the more route cache entries will be occupied along the path to the receiver. Let's assume you'll need 15 bytes per packet: `MHR_PACKET_MAX_LENGTH_ACK_IMPL_ACK=15`.

4. Although motes are capable of transmitting 20 packets/s we want to use less bandwidth to prevent thrashing. Especially if routes are not found or packets do not arrive it is very likely that too many packets are sent in too little time. `PPS_THRESHOLD=5` prevents messages to be sent if already 5 packets were received within the last second. `SEND_INTERVAL = 100` (milliseconds) specifies the mean time between two packet transmissions.

5. Should implicit acknowledgment for find packets and return packets or only for return packets be sent? (this decision is static)
   `NO_IMPL_ACK_FOR_FIND_PACKETS = 1`
   The previous statement means that only implicit acks for return packet will be sent. (`FIND_PACKETS_IMPL_ACK_RESENDS` will therefore be ignored)

6. We want to be sure that if a route was found, data and acknowledgment packets make their way to the destination and back to us. Therefore the number of sends is set to:
   `RETURN_PACKETS_IMPL_ACK_RESENDS = 3`
   `RESP_PACKETS_IMPL_ACK_RESENDS = 3`
   `DATA_PACKETS_IMPL_ACK_RESENDS = 3`
   (Although the variable is called RESEND it contains the maximum times a packet is sent in total!)

7. Now the maximum time to get from one hop to the next has to be estimated:
   `2*SEND_QUEUE_SIZE*SEND_INTERVAL*number of resends`
   `=2*8*100*3=4800ms`
   and can be set directly to `MAX_RTT=4800` (milliseconds). This is the absolute worst case and using the calculated value results in very slow path resolution. (ttl 1: 4800ms, ttl2: 4800ms+9600ms ttl4: 4800ms+9600ms+19200ms ...). It is recommended to take it as an upper bound because normally the queue is not that full and even if we have to resend packets
   `MAX_RTT = 2*(SEND_INTERVAL*SEND_QUEUE_SIZE)` will works fine (→ Performance tests). Nevertheless you're encouraged to play with this value.

8. Is the network static? If it is we can prevent routes from aging and only refresh routes if packets fail to arrive. (Attention: if you use packets without acks and a node dies it will never be detected)
   `ROUTE_CACHE_DO_NOT_AGE=1`
   `ROUTE_CACHE_MAX_FAIL=3` specifies that if the delivery of a packet fails three times (not necessarily three times in a row!) the route will be invalidated.

5 Performance

Please note that TOSSIM only models the radio stack and not timing. Therefore the end time cannot be interpreted.
For all tests `PACKET_STORE_SIZE = 8` and `ROUTE_CACHE_DO_NOT_AGE = 1` was chosen, other values are the defaults.
A typical command line call that was used to run was

```
./main -rf=6x6-10.nss.bi -b=0 -t=1000 36
```

The files used for lossy mode (*.nss.bi) can be found the the directory `Developer/Layouts`. Please consider these when judging the results.

5.1 Name -> Send Options

In the performance tables below symbolic names for the send-parameters will be used.

data_ack	(MH_USER_WANT_ACK)
data_imp_data_ack:	(MH_USER_WANT_ACK| MH_USER_IMP_ACK)
find_imp	(MH_USER_FIND_IMP_ACK)
all_imp_data_ack	(MH_USER_WANT_ACK| MH_USER_IMP_ACK| MH_USER_FIND_IMP_ACK)
no_ack	0

5.2 Meaning of the Fields

Due to the slow calculation time of TOSSIM and the big number of tests that were done, a limit of 1000 virtual seconds was set for TOSSIM runs. A test that did not finish in time is marked with an '*'. The question that seems to remain open is whether the test would have succeeded if there had been enough time. Therefore a field *found paths* was added. If all nodes were found we can assume that the settings allow a successful transmission and that the test would have terminated successfully. If not all paths were found it may have succeeded eventually, but the settings are definitely not practical.

total sent packets	sum of all packets sent by all nodes
CRC fail	sum of all failed packet CRCs of all nodes
CRC ok	sum of all packet CRCs that were ok of all nodes
sending attempts	number of times the application wanted to send a packet
successful arrivals	number of packets that arrived at the destination
success	finished in time and success = 1 otherwise 0
end time	approx. TOSSIM end time
found paths	what paths were found

When packets are broadcasted these packets are also received by nodes that are not on the path, but we still count these receptions to the failed and correct CRCs. A sending attempt corresponds to a call to SendMsg*.send.

5.3 Test 1

8 nodes connected only to their neighbors.

Node 0 transmits value 0 to node 1, 2, 3, 4, 5, 6, 7. As soon as all transmissions are acknowledged (or sent successfully - see send options) Node 0 continues with sending value 1 to node 1, 2,3,4,5,6,7, etc. A total of 70 packets was sent (10 to each node).
The test was done in TOSSIM with three different bit switch probabilities and five different send settings. If all packets arrived successfully the test was not necessarily redone with altered parameters.
After 1000 virtual seconds the test was stopped or earlier if nodes weren't even found. A stop is marked with an asterisk in the end time.

Estimation:

Note: To find node 5 we start with ttl 1, ttl2, ttl4 finally ttl8

The number of packets to find node 1 is: 1 find packet + 1 return packet
The number of packets to find node 2 is: $1 + 2$ find packets + 2 return packets
The number of packets to find node 3 is: $1 + 2 + 3$ find packets + 3 return packets
The number of packets to find node 4 is: $1 + 2 + 4$ find packets + 4 return packets
The number of packets to find node 5 is: $1 + 2 + 4 + 5$ find packets + 5 return packets
The number of packets to find node 6 is: $1 + 2 + 4 + 6$ find packets + 6 return packets
The number of packets to find node 7 is: $1 + 2 + 4 + 7$ find packets + 7 return packets

Therefore 56 find and 28 return packets = 84 packets routing overhead at least.

There are 70 packets to be delivered:
For destination 1 we will send a total of 10 packets.
For destination 2 we will send a total of 20 packets.
For destination 3 we will send a total of 30 packets.
For destination 4 we will send a total of 40 packets.
For destination 5 we will send a total of 50 packets.
For destination 6 we will send a total of 60 packets.
For destination 7 we will send a total of 70 packets.

A total of 280 packets for the data will be used.

Hence a perfect network with no packet loss will be able to deliver the 70 packets by using **364** packet transmissions.

Settings Test 1

	Test 1.1	**Test1.2**	**Test1.3**	**Test1.4**
FIND_PATH_MIN_TTL_STATE	TTL1	TTL4	TTL4	TTL4
FIND_PATH_MAX_TTL_STATE	TTL8	TTL8	TTL8	TTL8
RESOLVE_QUEUE_SIZE	(ROUTE_CACHE_NUM_ENTRIES/2 +1)	2	2	2
NO_IMPL_ACK_FOR_FIND_PACKETS	0	0	0	1
FIND_PACKETS_IMPL_ACK_RESENDS	3	4	4	4
RETURN_PACKETS_IMPL_ACK_RESENDS	3	4	4	4
RESP_PACKETS_IMPL_ACK_RESENDS	3	4	4	4
DATA_PACKETS_IMPL_ACK_RESENDS	3	4	4	4
SEND_INTERVAL	200*MILLI_SECOND			
PPS_THRESHOLD	5	5	5	2
PPS_SCALING	2	2	2	2
MAX_RTT	2*(SEND_INTERVAL* SEND_QUEUE_SIZE)		200*SEND_QUEUE_SIZE(=8)	
IMPL_ACK_TIMEOUT	MAX_RTT	MAX_RTT	MAX_RTT	MAX_RTT
FIND_PATH_NEXT_TTL	1000	1000	400	400
ROUTE_CACHE_MAX_FAIL	3	8	8	8
USE_TOS_ACK	undefined			

Test 1.1

bit switching probability: 0.0

	data_ack1	data_imp_data_ack1	find_imp1	all_imp_data_ack1	no_ack2
total sent packets	1051	807	358	868	325
CRC fail	23	18	14	17	7
CRC ok	1790	1367	575	1482	531
sending attempts	92	73	70	73	70
successful arrivals	70	70	58	70	62
success	1	1	0	1	0
end time	336064	202688	102720	207104	99904
found paths	1-7	1-7	1-7	1-7	1-7

bit switching probability: 0.01

	data_ack2	data_imp_data_ack2	find_imp2	all_imp_data_ack2	no_ack2
total sent packets	2590	2982	316	4581	854
CRC fail	521	606	63	1208	172
CRC ok	3310	3960	413	6376	1091
sending attempts	11	48	74	197	132
successful arrivals	5	34	33	70	36
success	0	0	0	1	0
end time	1955936*	2180160*	112736	1070336	694304
found paths	1-7	1-7	1-7	1-7	1-7

bit switching probability: 0.02

	data_ack3	data_imp_data_ack3	find_imp3	all_imp_data_ack3	no_ack3
total sent packets	1151	1602	798	4334	1489
CRC fail	706	713	530	2818	678
CRC ok	1153	1206	649	3641	1066
sending attempts	7	10	95	186	224
successful arrivals	3	3	13	6	2
success	0	0	0	0	0
end time	1023712*	1023808*	223744	1069984*	1056640*
found paths	1-3	1-3	1-7	1-7	1-3

Test 1.2

bit switching probability: 0.0

	data_ack1	data_imp_data_ack1	find_imp1	all_imp_data_ack1	no_ack1
total sent packets	884	779	313	795	313
CRC fail	21	18	7	10	11
CRC ok	1511	1336	514	1375	513
sending attempts	94	72	71	75	70
successful arrivals	70	70	61	70	58
success	1	1	0	1	0
end time	369312	201760	103904	212096	107552
found paths	1-7	1-7	1-7	1-7	1-7

bit switching probability: 0.01

	data_ack2	data_imp_data_ack2	find_imp2	all_imp_data_ack2	no_ack2
total sent packets	1007	1789	383	2748	661
CRC fail	230	353	93	701	140
CRC ok	1358	2591	493	3869	945
sending attempts	150	141	76	147	79
successful arrivals	6	63	25	70	2
success	0	0	0	1	0
end time	1023808*	1023808*	169152	888544	1023296*
found paths	1-6	1-7	1-7	1-7	1-7

bit switching probability: 0.02

	data_ack3	data_imp_data_ack3	find_imp3	all_imp_data_ack3	no_ack3
total sent packets	608	672	689	3325	542
CRC fail	304	335	456	2194	284
CRC ok	476	515	515	2854	392
sending attempts	154	145	87	151	138
successful arrivals	2	3	11	7	0
success	0	0	0	0	0
end time	1023488*	1022016*	305760	1021184*	1023424*
found paths	1-4	1-3	1-7	1-7	1-3

Test 1.3

bit switching probability: 0.00

	data_ack1	data_imp_data_ack1	find_imp1	all_imp_data_ack1	no_ack1
total sent packets	906	833	332	836	335
CRC fail	18	21	8	28	8
CRC ok	1563	1423	409	1412	562
sending attempts	90	74	70	73	71
successful arrivals	70	70	59	70	63
success	1	1	0	1	0
end time	323008	185312	96288	176800	102464
found paths	1-7	1-7	1-7	1-7	1-7

bit switching probability: 0.01

	data_ack2	data_imp_data_ack2	find_imp2	all_imp_data_ack2	no_ack2
total sent packets	1597	1842	374	1881	846
CRC fail	345	429	87	425	192
CRC ok	2180	2583	489	2556	1168
sending attempts	248	173	74	114	114
successful arrivals	6	70	37	70	5
success	0	1	0	1	0
end time	1023936*	776448	111360	627936	424768*
found paths	1-7	1-7	1-7	1-7	1-6

bit switching probability: 0.02

	data_ack3	data_imp_data_ack3	find_imp3	all_imp_data_ack3	no_ack3
total sent packets	1398	1517	512	4849	1398
CRC fail	691	805	283	3480	691
CRC ok	1099	1189	444	4123	1099
sending attempts	343	348	86	162	343
successful arrivals	1	4	12	20	1
success	0	0	0	0	0
end time	1023552*	1023840*	162816	1023744*	1023552*
found paths	1-3	1-3	1-7	1-7	1-3

Test 1.4

bit switching probability: 0.01

	data_ack2	data_imp_data_ack2	find_imp1	all_imp_data_ack2	no_ack2
total sent packets	1597	1842	292	1397	1227
sending attempts	248	173	78	105	206
successful arrivals	6	70	33	70	32
success	0	1	0	1	0
end time	1022784*	776448	114432	421248	675072
found paths	1-7	1-7	1-7	1-7	1-7

bit switching probability: 0.02

	data_ack3	data_imp_data_ack3	find_imp3	all_imp_data_ack3	no_ack3
total sent packets	930	1517	1061	1353	713
sending attempts	235	348	315	333	343
successful arrivals	1	1	12	1	1
success	0	0	0	0	0
end time	659232	1023840*	827840	1022688*	1023552*
found paths	1-3	1-4	1-7	1-6	1-3

Discussion

You've probably noticed the total number of packets sent can be less than our estimate. This is due to some optimization: if find packets from the same source with different ttls are inserted into the send queue, the packets are merged to one containing the bigger ttl value.

Bit switching probability 0.0

If acknowledgments are used all packets arrive otherwise about 80%. The least packets were sent in Test 1.2 because ttl was set between 4 and 8 and the the send interval was quite large to prevent collisions and hidden nodes.

Bit switching probability 0.1

Nodes are still successfully found, but with more effort. The main problem are nodes 6 and 7. The application waits until all nodes reply with an ack (or the path was found for transmissions with no ack) and continues after. If data packets never reach a node no other packets are sent. Therefore the field

successful arrivals is very low although transmissions close nodes are no problem. Another setting impairs performance: After ROUTE_CACHE_MAX_FAIL failed transmissions the route is invalidated and the route finding starts again.

Bit switching probability 0.2

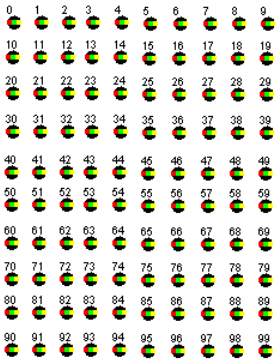
All runs failed to terminate in time with the exception of find_imp. The reason is that with implicit acks the routes are successfully found and packets are transmitted from node 0 to 1 (although they do not arrive at their destination in general). If more time had been given the runs that found all paths would have been successful. The most promising settings are those of test 1.3 with implicit acks for find and for data packets.

No implicit acks for find packets

In test 1.4 no implicit acks for find packets (but for return packets if `MH_USER_FIND_IMP_ACK` was set) were used. Looking at column find_imp it can be seen that the number of find packets is higher than in test 1.3. Therefore it is recommended to set `NO_IMPL_ACK_FOR_FIND_PACKETS` to 0 if failure probability is high.

Recommendations
If your TinyOS set up allows it you can also define USE_TOS_ACK. Test 2 shows a performance example.

5.4 Test 2



Node 0 (upper left corner) sends 10 packets to Node 99 (lower right corner)

Settings Test 2

	Test 2.1	**Test2.2**
USE_TOS_ACK	undefined	defined
FIND_PATH_MIN_TTL_STATE	TTL1	TTL4
FIND_PATH_MAX_TTL_STATE	TTL32	TTL8
NO_IMPL_ACK_FOR_FIND_PACKETS	1	1 (no influence)
FIND_PACKETS_IMPL_ACK_RESENDS	3	4 (no influence)
RETURN_PACKETS_IMPL_ACK_RESENDS	3	8 (no influence)
RESP_PACKETS_IMPL_ACK_RESENDS	3	8 (no influence)
DATA_PACKETS_IMPL_ACK_RESENDS	3	8 (no influence)
SEND_INTERVAL	200*MILLI_SECOND	400*MILLI_SECOND
PPS_THRESHOLD	5	2
PPS_SCALING	2	2
MAX_RTT	2*(SEND_INTERVAL*SEND_QUEUE_SIZE)	
IMPL_ACK_TIMEOUT	MAX_RTT	MAX_RTT
FIND_PATH_NEXT_TTL	$MAX\_RTT/2$	MAX_RTT

Test 2.1

LossyBuilder -d 10 10 -s 20 every switching probability replaced with 0.00

	data_ack1	data_imp_data_ack1	find_imp1	all_imp_data_ack1	no_ack1
total sent packets	429	450	229	319	332
CRC fail	766	783	346	346	780
CRC ok	5251	6510	2573	3743	3594
sending attempts	10	10	10	10	10
successful arrivals	10	10	10	10	10
success	1	1	1	1	1
end time	72832	78816	30912	57856	44896

LossyBuilder -d 10 10 -s 10 then converted to symmetric routes (=routes with same bit switch probability)

	data_ack2	data_imp_data_ack2	find_imp2	all_imp_data_ack2	no_ack2
total sent packets	3106	3106	676	2050	3106
CRC fail	27840	27840	5982	17484	27840
CRC ok	16264	16264	3944	13918	16264
sending attempts	9	9	11	18	9
successful arrivals	0	0	1	5	0
success	0	0	0	0	0
end time	1000000*	1000000*	160896	1000000*	1000000*

Test 2.2

LossyBuilder -d 10 10 -s 20 every switching probability replaced with 0.00

	data_ack2	no_ack2**
total sent packets	275	205
CRC fail	201	201
CRC ok	4049	2869
sending attempts	10	10
successful arrivals	10	10
success	1	1
end time	96928	59968

** enabling USE_TOS_ACK without MH_USER_WANT_ACK
is not supported! (Messages arrive an arbitrary number of times!)

LossyBuilder -d 10 10 -s 10 then converted to symmetric routes (=routes with same bit switch probability)

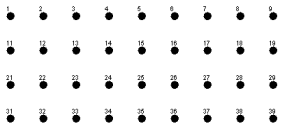
	data_ack2	no_ack2**
total sent packets	2332	3086
CRC fail	21383	28323
CRC ok	17870	24425
sending attempts	15	23
successful arrivals	10	70
success	1	1
end time	587808	513824

** enabling USE_TOS_ACK without MH_USER_WANT_ACK
is not supported! (Messages arrive an arbitrary number of times!)

Discussion

Sending less packets per second and activating USE_TOS_ACK improved the results remarkably for both setups. Other tests without USE_TOS_ACK (not shown) resulted in bad performance because node 99 was found but no return packet ever reached node 0.

5.5 Test 3



Node 0 sends to 5
Node 5 sends to 10
Node 10 sends to 15
...
Node 35 sends to 40 (never reached in test 3.1)

The purpose of this test is to show that AMUHR also works if there is more then one sender in a network.

Settings Test 3

	Test 3.1 and Test 3.2
USE_TOS_ACK	defined
PACKET_STORE_SIZE	8
TOS_ACK_RESENDS	10
FIND_PATH_MIN_TTL_STATE	TTL1
FIND_PATH_MAX_TTL_STATE	TTL16
SEND_INTERVAL	200*MILLI_SECOND
PPS_THRESHOLD	5
PPS_SCALING	2
MAX_RTT	2 * (SEND_INTERVAL * SEND_QUEUE_SIZE)
IMPL_ACK_TIMEOUT	MAX_RTT
FIND_PATH_NEXT_TTL	$MAX\_RTT/2$
ROUTEC_MAX_FIND_IN_CACHE	8

Test 3.1

Following test was done with 40 nodes as described above with 8 senders and 7 available destinations.

LossyBuilder -d 10 10 -s 20 every switching probability replaced with 0.00

	data_ack1
total sent packets	1550
CRC fail	2042
CRC ok	16432
sending attempts	76
successful arrivals	70
success	1
end time	133952

LossyBuilder -d 10 10 -s 10 then converted to symmetric routes (=routes with same bit switch probability)

	data_ack2
total sent packets	5959
CRC fail	37357
CRC ok	45474
sending attempts	101
successful arrivals	70
success	1
end time	431904

LossyBuilder -d 10 10 -s 20 then converted to symmetric routes (=routes with same bit switch probability)

	data_ack3
total sent packets	10780
CRC fail	34024
CRC ok	27593
sending attempts	143
successful arrivals	70
success	1
end time	751872

Test 3.2

Following test was done with 100 nodes in a 10x10 grid with 19 senders

LossyBuilder -d 10 10 -s 20 every switching probability replaced with 0.00

	data_ack1
total sent packets	15796
CRC fail	46327
CRC ok	166094
sending attempts	266
successful arrivals	190
success	1
end time	482912

LossyBuilder -d 10 10 -s 10 then converted to symmetric routes (=routes with same bit switch probability)

	data_ack2
total sent packets	98180
CRC fail	886185
CRC ok	307889
sending attempts	416
successful arrivals	57
success	0
end time	1000000*

LossyBuilder -d 10 10 -s 20 then converted to symmetric routes (=routes with same bit switch probability)

	data_ack3
total sent packets	46516
CRC fail	146819
CRC ok	109785
sending attempts	363
successful arrivals	101
success	0
end time	1000000*

Analysis has shown that in the second and the third test the send interval was chosen too small and that important route cache entries got overwritten

by unimportant find packet paths. Simply increasing SEND_INTERVAL to 300ms turned out to improve the second test in 3.2 (31 more packet arrivals and 15% less sent packets in 1000000* virtual seconds) - the others weren't tested, but are assumed to improve as well. To prevent route cache entries to get overwritten ROUTEC_MAX_FIND_IN_CACHE can be changed. Keeping SEND_INTERVAL=200ms and setting ROUTEC_MAX_FIND_IN_CACHE=4 resulted in {190, 76, 116} successful packet arrivals with about the same number of packets sent.

5.6 Conclusion of Tests

If your mote or configuration doesn't support the ack field in TOS_Msg AMUHR will work fine if loss is moderate. Otherwise it's recommended to use implicit acknowledgments. Other settings should be based on your needs. Setting `PPS_THRESHOLD` and `SEND_INTERVAL` to large values is good practice to prevent CRC errors due to packet collisions (and to save energy as well).

6 Implementation Details

6.1 Intro

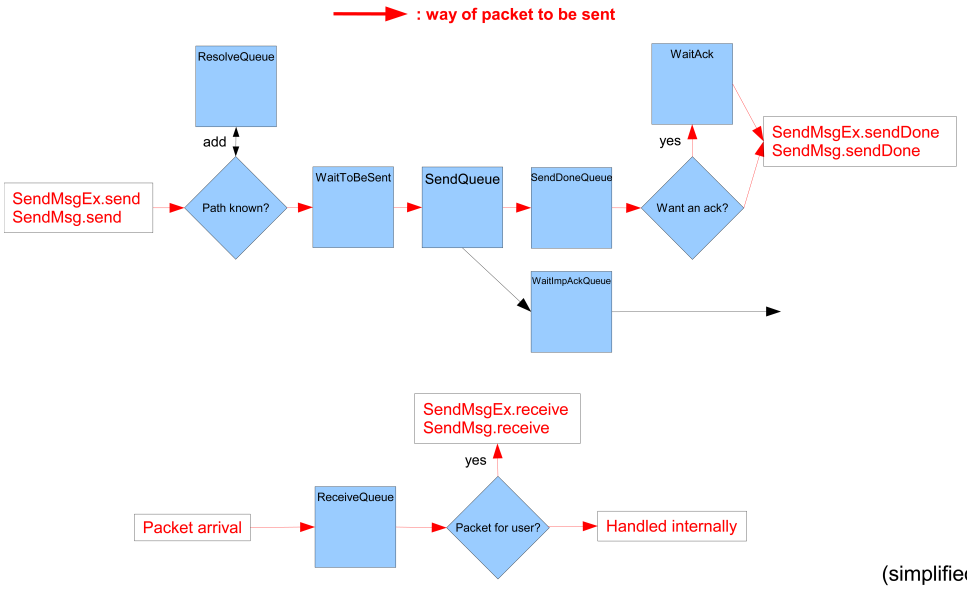
The task of this semester thesis was to implement a simple DSR and then to optimize it. When being almost done the tests indicated very bad results if packets were lost. Therefore the decision to add implicit acknowledgments was taken, which made the code quite big. Since the tests were still not convincing the ack field in TOS_Msg was considered more closely as an indicator whether a message arrived at the next hop or not. This flag was not considered earlier since it belongs to the link-layer and there was no documentation stating any guarantee that this field would be set in any TinyOS configuration. Therefore special care has to be taken by the user to assure that the ack field is set. In TOSSIM the ack field seems to be supported by the default configuration.

6.2 Files

Following files can be found in `AMUHR`.

MHSettings.h	defines behavior, memory usage etc.
MHConstants.h	content should not be changed
MHDerivedConstants.h	do not change either
MHPackets.h	defines data structures for packets
AMUHR.nc	needs to be wired into application instead of GenericComm
AMUHRM.nc	implementation of AMUHR
DataPacketHistoryM.nc	stores history of data packets by saving the sender and packet ids.
PacketHistoryM.nc	same as DataPacketHistoryM.nc, but for find- and return packets
ReceiveQueueM.nc	stores received packets until they're processed
ResolveQueueM.nc	stores the addresses we seek and keeps track of ttl in find packets
RouteCacheM.nc	stores routes that we need
SendDoneQueueM.nc	stores packets that have been sent and might need to signal sender
SendMsgEx.nc	defines interface of SendMsgEx
SendQueueM.nc	stores packets that need to be handed down to GenericComm
TOSMsgPacketStoreM.nc	provides empty TOSMsg packets for temporary use
WaitAckM.nc	stores user packets waiting for acknowledgments
WaitImpAckQueueM.nc	stores routing packets waiting for implicit acknowledgments
WaitToBeSentM.nc	store user packets waiting for the path
Omitted files:	contain interfaces to the described modules

6.3 Queues



(simplified)

6.4 Security

Every incoming or outgoing packet passes a security test assuring that the packet format is valid. However, this test does not prevent attackers from adding invalid entries to the route cache.

6.5 Extensibility

Some fields in the packets are not yet occupied or provide interesting but not yet used information. For instance there is an empty field in find packets that could be used to extend the ttl which would allow controlled routing up approximately 65000 hops. In return packets the distance from the destination node is returned but not used. Empty fields are also available in response packets. Every return packet contains the age of the route that is stored in the local cache of the route requester. Currently the age is just set the the maximum time. This behavior could be changed to for instance set the age to the lowest age of a route found in the path.

6.6 Implemented Optimizations

This is a selection of optimizations that were implemented. Omitted once are just not worth mentioning or were discussed in other sections of this document.

- Only the first find path packet from a source (initiator) is forwarded - because it contains the fastest path.
- If two or more different packets miss the path to the same destination only one find path packet is sent.
- If there are find packets from the same source, but with different ttls in our send queue these packets are merged to one packet containing the larger ttl value.
- If implicit acks are requested and a packet arrives a second time we send a *dummy packet*; that is a packet with a flag (dummy flag). We set this flag so that the next hop doesn't think that we didn't hear its implicit ack.
- If a dummy packet arrives, but we did not already get the packet we forward the dummy packet after removing the dummy flag if necessary.
- When a return packet passes through we cancel all find packets from the same sender that are in our send queue.
- If implicit acks are used also nodes that hear the transmission of return packets will not forward return packets or find packets from the same source with the same destination.
- Return packets are accepted as implicit acknowledgments for find packets. The same applies for data packets and response packets.

6.7 Tools

In directory `Developer` a debug version of AMUHR is located that allows
- to specify from what node a certain node may accept packets (to simulate a directed graph on real nodes)
- to keep track of the number of packets that were sent/received
- allows you to track the algorithm in TOSSIM with dbg outputs
- to look at the sent/received messages on standard out when run in TOSSIM

The best way is start reading LimitComm module to see how it works.
By the way the sample applications provided are based on a fully connected debug version.

7 Limitations

Data structures: Guaranteed to hold up to to 32765 entries.

Route length: The maximum route length that can be controlled in a network is 254 hops. There is an option for infinite routes but it contains the risk that packets are forwarded forever.

Route length to next hop: The route length within data-packet is limited to 13 hops.

Network: All motes in a group are expected to be compiled with the same routing settings. Packets are dropped if they are not conformant.

Payload length: Even if you increase `TOSH_DATA_LENGTH` the payload must always be smaller than 256 bytes.

Network assumption: Routes are assumed to be symmetric and bidirectional. If bit switching probabilities are not the same in the opposite direction AMUHR should also work if re-sending packets is enabled.

UART: AMUHR only supports communication with radio and does not forward to serial port.

8 Pros and Cons

- RAM usage
  A simple application with `PACKET_STORE_SIZE = 8` uses approximately
  - 32994 bytes in ROM
  - 1958 bytes in RAM

  and if you want to be very sure that packets do not get transmitted twice even more RAM can be used.

- granularity of timers should be increased: for example it would be nice to be able to specify that we don't care if the packet arrives 2 or 3 seconds later; this would save a couple of lookups while waiting for the path; currently some of the timers are coupled together

- some tasks are quite big and should be split

- bidirectional symmetric routes are assumed but it should work with just bidirectional routes also

- route cache saves paths that are not used from find packets and might overwrite more important paths

+ you can steer RAM usage by just changing parameters in MHSettings.h

+ payload length can be chosen almost freely

+ works on motes that do not support ack field in TOS_Msg

+ the way packets are sent (implicit acks, acks, no acks) can be chosen at runtime

+ SendMsg interface is fully supported to allow existing applications to be ported to multi-hop quickly

9 Outlook

As a next step following optimizations are suggested:

- adding a packet splitter to simplify the handling with decreased maximum allowed packet length that was set before compile time
- splitting the huge tasks into smaller ones
- allow unidirectional routes - up to now the same path is used by the sender and the receiver when answering
- the part that uses the most memory is the implicit ack support: by removing it and using the link-layer ack field a considerable amount of RAM could be saved.

10 Conclusion

AMUHR is a multi hop router that is easy to deploy and should work from scratch in small sensor networks. It is possible to optimize memory usage and performance based on a set of parameters and to adapt to bigger networks. Tuning is a bit tricky at first until the effects are fully understood – therefore a lot of performance examples were provided. Its RAM usage depends on preferences chosen, but 2000 bytes should be expected. The application was tested with TOSSIM with packet loss and a maximum of 100 motes.