An AODV Implementation for the ScatterWeb

Janos Kutscherauer
Freie Universität Berlin
Email: kutsch@inf.fu-berlin.de

(Technical Report)

***Abstract*—Mobile ad-hoc networks (MANETs) are collections of mobile devices, which autonomoulsy want to communicate. Routing protocols for *MANET*s are mainly challenged with the lack of knowledge about the topology of the network, as participating devices can move within the network. For the handling of this special requirement on *MANET* routing protocols there have been developed two families of approaches: Proactive and reactive routing protocols.**

The RFC-3561 introduces the reactive routing protocol for mobile ad hoc networks *AODV*. This (experimental) routing protocol has been implemented on the *ScatterWeb* platform. The implementation has been evaluated in real test field setups.

I. Introduction

THE major challenge for routing protocols in MANETs is that there is no stable knowledge about the topology of the network at any time. Because the participants of MANETs are expected to be *mobile*, devices can just appear in the network, move their position and leave. Routing protocols for MANETs have to deal with these uncertainities concerning locality as well as with all the challenges which wireless networks are facing anyway (e.g. unreliable wireless links).

The difficulty lies in discovering neighboring participants (devices) for every participant in the network, and also maintaining neighbourhood information. The neighbourship information is essential for every routing protocol to dicover and use routes to other devices for forwarding data. As participants can always move in the network, neighbourship information can only be temporary. It is up to the routing protocol to make sure, that known neighbours are valid and new neighbours are recognized.

There are two basic approaches for MANET routing algorithms to target the lacking topology knowledge:

A. Proactive

Every participant keeps and maintains information for the entire network. It does so by constantly asking into the network for information **before** routes to other devices are neede. When participants discover locality changes (of themselves or neighbours) they hasten to spread their new information into the network. This strategy of collecting and maintaining locality information is called *proactive*, because most of the work is in particular done **prior** to a route request.

The major advantage of proactive routing protocols is the short time it takes to find a route to any destination, because every node idealy has all information over the whole network already collected (and updated) at any time. The disadvantage however is the high traffic load it takes to keep the locality information up to date over the entire network. This can usually only be done by frequently repeated broadcasts or multicast, which causes constant maintainance traffic.

B. Reactive

A participant searches for a route to a destination only when it actually needs it. This is for instance, when a data packet is to be sent to the destination. Using a reactive approach, a route is only discovered *on demand*. To establish a route, broadcasts or multicasts can be used. Once a route has been found to a destination, there are usually actions taken to maintain that route - however, the loss of a route will not lead to further route discovery actions. Such protocols are called *reactive*, because they only **react** on demand. If no route is needed, a reactive protocol has just little work to do.

The main advantage of reactive routing protocols is that the network overhead traffic is kept at a minimal amount. As opposed to proactive routing protocols, the major disadvantage of reactive routing protocols lies in the time it takes to actually send data when no route to the desired destination has been established.

The *RFC-3561* defines the routing protocol *AODV*, which uses the reactive approach on dealing with the unknown topology and changes in locality in *mobile ad-hoc networks*.

This work is organized as follows: In section II an overview of considered routing protocols is given. The *AODV* routing protocol and it's function is described in section III. The *ScatterWeb* platform, i.e. the device and the operating system, is briefly indroduced in section IV. The basic focus of this work is done in section V, where the implementation of *AODV* on the *ScatterWeb* platform is introduced. The attempts on testing the implementation in a real test field are presented in section VI, and finally in section VII the experiences that have been made in both implementing the *AODV* routing protocol and testing the implementation are discussed.

II. Protocols Overview

A small variety of routing protocols from different protocol families have been considered prior to this work. These have been

OLSR *(Optimized Link State Routing Protocol)* [1] A proactive routing protocol which introduces the concept of *multipoint relays* as an enhancement against broadcasts.

DSR *(Dynamic Source Routing Protocol)* [2] A reactive routing protocol which supports multi-path routing and unidirectional links.

DSDV *(Highly Dynamic Destination-Sequenced Distance-Vector Routing)* [3] A proactive distance vector routing protocol that avoids the 'counting to infinity' and looping problems.

ZRP *(Zone Routing Protocol)* [4] A hybrid of proactive and reactive routing protocols.

ARA *(Ant Routing Algorithm)* [5] [6] A rather novel approach on network routing, based on the path discovery strategies of ants.

AODV *(Ad hoc On-Demand Distance Vector Routing)* [7] A reactive routing protocol based on the distance-vector routing protocol family, capable of both unicast and multicast routing.

DYMO *(Dynamic MANET On-demand Routing)* [8] The follow-up of *AODV*. Dymo enjoys the current engeneering focus on reactive routing protocols.

AODVjr *AODV Jr.* [9] A simplified version of *AODV*, which omits some features of AODV such as sequence numbers, hello messages and precursor lists.

The implementation was to be made on the *ScatterWeb* platform. The *ScatterWeb* operating system is a rather minimal operation system designed for RF applications for the *Texas Instruments MSP430* microcontroller family. A reactive approach on the routing protocol seemed to be appropriate for this work, owing to the highly limited ressources of the *MSP430* microcontroller.

For this work, *AODV* has been chosen to be the routing protocol to be implemented and tested. *AODV* is the most commonly known reactive routing protocol for mobile ad-hoc networks. Because the RFC3561 [7] has went through a draft lifecycle end ended up as an RFC (experimental), this routing protocol was considered as a suitable candidate to be implemented. The assumption was, that the RFC would shape the ideas and the function of the protocol in a clear fashion, so that most of the effort in this time limitated work could go straight into the actual implementation on the *ScatterWeb* sensor boards, and then into the testing.

The simplified *AODV* version *AODV Jr.* [9] was also considered for implementation. *AODV Jr.* would have spared much of the data organization work, which plain *AODV* requires, and therefore would have been more suitable for a 1-man project in the given timeframe. However, there was the wish to implement some real bigger, mature protocol with all it's snares, and so the hard way was kept by chosing *AODV*.

III. AODV Overview

The **Ad-hoc on-demand distance vector routing protocol** (*AODV*) is an reactive routing protocol designed for mobile wireless devices. These devices, which form themeselves into a wireless network, are referred to as **nodes** hereafter.

As many other routing protocols, the operation of *AODV* consists of two major parts: *route discovery* and *route maintenance*. The basic operation of *AODV* is briefly depicted next, while further details for the single operations follow. When a route to another node is needed (that is, when data is to be sent to a destination) and does not exist, this route must be discovered first. Once a route exists, it is only valid for a certain amount of time. Whenever valid routes are used by normal *AODV* operation, this is seen as a proof that the route is still valid and the lifetime increases. After an expiration of a routes lifetime, another route discovery must be performed before the route can be used again. When a node notices, that a route is not valid anymore, it propagates this information as error messages to other nodes. To provide connectivity information to other nodes, hello messages are used.

A. Route Discovery

If and only if a node has to send a data packet to a destination, for which it doesn't have a valid route available, it has to start the *route discovery* process. It does so by broadcasting a special *route request (RREQ)* message into the network. Each RREQ message contains a (temporarily) unique request ID to prevent circular broadcasting.

While the RREQ message makes it's way through the network, every node that forwards the RREQ creates a route entry in it's routing table, so that it will be able to provide a way back to the originator. Every node will only react on a RREQ message once per originator address and request ID. To control the dissemination of broadcast RREQs in multiple retries, *AODV* uses an expanding ring technique. Additionally, an exponential backoff timeout is used for repeated RREQ broadcasts.

When a RREQ message arrives at the desired destination, this node sends back a special *route reply (RREP)* message to the originator of the route request. This RREP message is unicast towards the originator of the RREQ. Because every intermediate node of the RREQ should have created a route towards the originator, the reverse route should exist. This is only true for bidirectional links though. However, *AODV* uses techniques to recognize unidirectional links. By keeping so called "blacklists" of neighnodes, of which is known that the link is unidirectional, a destination node will only reply to route requests, which it has received over a bidirectional link.

On the way back to the originator, the **number of hops** is counted in the RREP message. In the event that a route reply message arrives at the originator, this node has now a valid route for the destination, from which the route reply was received. Also, every other node on the path of the RREP message now has a valid route. These routes are valid for a certain amount of time and can be used to send and forward data packets.

AODV consideres several enhancements on replying to route requests. Intermediate nodes, who are not the desired destination of a RREQ message may also answer the request themselves with a RREP message, if they have an active route to the searched destination available. Also, to detect unidirectional links, replying nodes can claim one-hop acknowledgements for route replies.

B. Route Maintenance

When a node has a route to a specific destination in it's routing table, this route is only valid for a certain amount of time. Any incoming *AODV* message, which has new information about that destination (e.g. route replies from that destination) will refresh and possibly update the route. Sometimes, routes appear with a smaller hop count, which will then rather be used. *AODV* uses subsequently increased *sequence numbers* for each destination, to make it possible to identify whether an information is new or stale. These sequence numbers are kept by every node. Every time a node sends out an *AODV* message it increases it's own sequence number. To prevent the *counting to infinity problem*, thos sequence numbers are kept as unsigned numbers, though compared as signed integer values.

Once a node is part of an active route, it provides connectivity information to other nodes to keep the routes alive by sending special *HELLO* messages. These HELLO messages are sent frequently and repeatedly.

When the breakdown of a link is detected by a node, it propagates this information to neighbouring nodes using a special *route error (RERR)* message. A link breakdown can be recognized either by a route timeout expiration or by receiving a RERR itself. Also, a route error is detected as such, when a node is used to forward data packets to a destination which it hasn't got a route for. To provide this information pointedly to all nodes, who might have routes to the destination that has been lost, but to prevent flooding the whole network, each *AODV* node keeps *precursor lists* for every destination. Every node, which has been seen on a route to a destination by one node will be in this precursor list for this destination. The precursor list can be then used to address affected nodes at any time.

C. AODV *Features*

AODV intentionally provides the oportunity to operate over unidirectional links. Although links are expected to be symmetric (i.e. bidirectional) per default, *AODV* allows applications to signal that bidirectional connections are desired. *AODV* uses RREP acknowledgement messages to make sure, that a sent RREP message has been received by the next hop.

AODV is also capable of supporting subnet routing.

IV. SCATTERWEB

The implementation of the chosen routing protocol was to be implemented and tested on the *ScatterWeb* operating system, which was running on the ScatterWeb sensor boards *MSB-430*. Both the *ScatterWeb* operating system and the sensor board are depicted briefly in this section.

A. *The Sensor Board* MSB-430

The sensor board *MSB-430* consists of a *Texas Instruments* MSP430-F1612 microcontroller, which has a *Chipcon CC1020* RF transceiver connected to it. The sensor board has sensors such a 3-axis-acceleration sensor and a temperature/humidity sensor attached to it. The sensors however are not relevant for this work and weren't used.

There were several sensor boards available for the tests of the implementation. They differed slightly in equipment (e.g. some didn't have the temperature sensor), what was most relevant to the tests of the RF applications was the antenna. Those antennas are discussed in the Testing section.

The *MSP430-F1612* microcontroller is a 16-bit controller of the *Texas Instruments* MSP430 ultralow power microcontroller family. This version *F1612* has several controller-modules such as digital-analog, analog-digital and direct-memory access controllers. There are two 16-bit timers plus a *watchdog* timer, and also different serial interfaces such as UART and SPI. The *F1612* has 55kB flash memory and 5kB RAM.

The CC1020 transceiver is a single-chip device, designed for ultralow power applications and operates on the frequency bands 4xx, 868 and 915 MHz. It is connected via the SPI interface to the MSP430 microcontroller.

The MSB-430 boards all have a red LED attached to one I/O port of the MSP430 microcontroller. This LED is used to signal states in the tests, which will be described in this work.

The MSP430 controllers are programmed using the JTAG interface. On the sensor boards, there also exist 5-pin serial and power connectors for the FTDI USB connector cables. This way, serial input and output can be eaysily made over USB, whilst the whole module is supplied with power. Alternatively, to use the sensor nodes stand-alone, they can be powered with three AAA batteries.

B. *The Operating System*

The *ScatterWeb2* operating system is an experimental environment for the ScatterWeb boards, which has gone through quite a history of development by different developers. The operating system is programmed in C, and built with the GNU mspgcc compiler.

The ScatterWeb operating system provides functionality to register and unregister handler for system interrupts. One timer of the MSP430 is used to enable user applications to easily have routines called after certain delays. This timer function allows to register and unregister function calls with a parameter argument. A system clock is implementated to provide time and date information.

The basic function of the ScatterWeb operating system is the RF application. ScatterWeb utilizes a simple sort-of layer-2 protocol: every node has a 8-bit NodeID, the address 255 is the broadcast address, and the address 0 is undefined. Using these NodeIDs, ScatterWeb provides the functionality to send generic data to a specific address (including the broadcast address) to applications, and also to react on incoming data frames. A ScatterWeb data frame contains an application identifier (8-bit), the sender and destination NodeID, and several other information such as the RSSI value which indicates the transmission quality of the frame. So the ScatterWeb operating system enables applications to send and receive frames with individual user data. First tests showed, that the frame size is limited to about 80 to 100 bytes. A frame that is to be sent is buffered until it could be transmitted or had to be cancelled.

ScatterWeb has a nice and simple framework based on C-macros, with which it is easy to read serial input as

commands, and parse the passed parameters. The following listing shows an example usage of this framework. In the example, a `COMMAND` macro 'ping' is defined to capture he (case sensitive) serial input of the sequence '`ping`':

```
1 COMMAND(ping, 0, cmdargs) {
2   uint16_t node;
3   nodeID = String_parseUint16(cmdargs->args, NULL
        );
4   if (nodeID > 0) {
5     Net_sendPing((netaddr_t) nodeID);
6   }
7 }
```

Listing 1. Capturing serial input

This command facility is a powerful feature for processing user input. A bunch of system commands are already defined by the ScatterWeb operating system, such as `led` for setting and reading the led status, `txpwr` for adjusting and reading the transmission signal strength, and `reset` to restart the device.

In this work, command definitions were often used to define test programs which then could be started by user input. With a little hack into the operating system, such commands can also be generated by software. The special command `@<nodeID> <command>[ <args>]` will be interpreted as directing the command and the arguments to the specified node. For example, the following serial input would lead to turn all LEDs on, which can hear the broadcast of the node:

```
@255 led 1
```

The nodes LED itself will also be turned on.

C. Eclipse CDT

The ScatterWeb operating system source code exists as an integration for the *Eclipse CDT* IDE as well as a Microsoft Visual Studio integrated project. For this work, the Eclipse CDT was chosen to be the IDE. After the `JTAG-TINY` programmer and the `FTDI` connectors had been installed, and the `MSP-GCC` compiler chain has been setup, some additional makefile targets like `update`, `flashonlyusb` and `clean` were added. The `update` target only updates the edited code, without performing the actual build. The `flashonlyusb` makefile target only performs the flashing of already compiled and built code over the USB programmer, without actually building it. Doing this and using the Eclipse *Make Targets*-view, Eclipse became a very comfortable and really powerful developing environment from project management, over code editing and refactoring to the actual deployment of the program binaries.

Only the in-place debugging feature with views into the controllers registers, which integrated propriatary IDEs such as the Texas Instruments *Code Composer Essential* offers, wouldn't want to work with the MSP-GCC integration. This lack could be accepted though for this work, as the main focus was on the protcol implementation anyway.

V. Implementation

In this section, the implementation of the routing protocol AODV is presented. In the first part, some details on architectural decisions and also used utils are described, and in the second part the concrete implementation of the routing protocol is shown.

A. Implementation Overview

In this part, the basic approaches on the implementation on an embedded system, i.e. the MSP430 microcontroller, are described. Due to the highly limited resources compared with modern common computer systems, especially the limited memory did pose problems. The limited calculation power wouldn't bother too much. In the present case, which is an RF application, there isn't much serious calculation to be made. The speed, at which data is processed (i.e. forwarded, stored, compared), shouldn't be too critical for the chosen routing protocol.

1) Memory Management: For the AODV implementation, there is a small number of data needed to be buffered during the protocol operation. The storage of data isn't as trivial on a MSP430 microcontroller, as it could be at a 'normal system'. As there is only 5kB of RAM available, which is used for both heap and stack, any usage of the memory has to be considered with an economic focus.

There are three evolutional stages of the memory management for the present implementation, of which two are developed and the third one would evolute next. The first approach is the most simple one: Buffered data is simply stored in arrays of the particular data type. As arrays in C must be of fixed size, `#define` macros are defined on a central location to adjust these array sizes. These array-bufferes can then be used independently as ring buffers or fixed buffers as needed. In the moment that the project is built into an `ELF` file, it is clear whether the required buffer memory can be stored in the MSP430s memory or not.

Obviously, this approach is very easy, not dirty, though unflexible. The major disadvantage with this style of memory management is, that memory is allocated, which is not necessarily used. All allocated memory will be cut off from stack space. So especially when the maximum buffer space is used by the software, the stack will be limited to the minimal size.

After the fixed buffer approach worked fine, the management was changed completely. In the next approach, dynamically allocated memory is used to allocate just the memory that is effectively needed. To realize buffers using dynamic memory allocation in an easy way, double linked lists are used. Whenever a data entry needs to be stored, memory would be allocated for it, and an list entry would be created for that memory and appenden/inserted into a list. To remove a bufferd data entry, the memory would be freed and the list element removed from the list.

This memory management technique enables the software to always use just as much memory as it actually needs. However, when the needed memory reaches the amount of available memory, the allocated memory again blocks and depletes

memory space that would be needed for the stack. Well, this is indeed a situation which would always occur, if not some sort of management watches the amount of memory allocation to make sure, that there is always a specific amount left for stack usage. In the first memory management approach (fixed memory allocation), this could be done easily by adjusting the fixed memory sizes. In this second approach, a manegement unit would be needed.

This second approach has another larger downside: In every situation, where data needs to be stored, memory will be allocated for the single data entry. This produces quite an overhead on allocation management and fragmentation of the available memory space.

So in the third approach on memory management for the AODV implementation would use some more intelligent technique to allocate memory in larger blocks and to organize those lists in a - concerning fragmentation - more performant way. Also, some sort of guard would control the amount of dynamically allocated memory to ensure the operation of the application also under high needs. For the present AODV implementation, the easier usage of dynamic memory allocation was just fine and sufficient.

2) Lists: For the data buffers, a generic dynamic list implementation was useful to utilize the different needs of data buffers in a general way. Therefore, a double linked list was implemented as follows: Every list element may have a predecessor and a successor, and every list element stores a pointer to the data. Now, every list element can be the beginning of a whole list, and hence be the list itself. That is, when the predecessor element is `NULL`. Using these `PRE` and `SUCC` references, the list elements can be inserted, appended and removed. Also, the whole list can be traversed. The insertion of a list element is always dependent on the data, which the element is needed for. The list framework therefore doesn't allow to insert/append existing (maybe user created) list elements, but instead simply allows the creation of an entry for a particular data type. The list framework then allocates memory for the list element, also allocates memory for the data type, lets the list entry point to the data memory, connect the list entry to the list, and returns the pointer to the allocated data memory. The client application then only needs to copy the data that is to be stored into this data memory.

A list element is always be represented by the pointer to the user data. List elements can also be created with already existing data memory. In this case no memory is allocated for data. To delete data from the list, again the pointer to the data is used as the identifier, and the list framework does all the freeing and disconnecting of the linked list.

To provide performant insert operations, a new list element is always inserted at the beginning of the list to be the first list element.

The list framework is defined in the header file `aodv.genericList.h` and implemented in the source file `aodv.genericList.c`. The definition of the list data type is listed in listing 2. The functions, which the list framework provides are listed in listing 3.

3) Tools: The present AODV implementation uses a tiny logging facility to make the application better to understand.

```
1  typedef struct aodv_list_s aodv_list_t;
2  struct aodv_list_s {
3    aodv_list_t *prev;
4    aodv_list_t *next;
5    void *data;
6  };
```

Listing 2. List definition

```
1  void* list_createEntry(aodv_list_t **aList,
       uint8_t dataSize);
2  bool list_deleteEntry(aodv_list_t **aList, void
       *anEntry);
3  aodv_list_t* list_findEntry(aodv_list_t *aList,
       void *anEntry);
4  void list_clearList(aodv_list_t **aList);
5  uint8_t list_getSize(aodv_list_t *aList);
```

Listing 3. List interface

The ScatterWeb operating system directs every `printf` output (et al) to the USART controller to be transmitted to any serially connected receiver - as in the present case over the FTDI USB controller to any connected computer.

With this work, first the program `HTerm` by Tobias Hammer has been used to print the sensor nodes serial outputs. Later on, an own Java based application has been implemented. This tool also only printed the serial output of different connected sensor nodes - no storing of logfiles was implemented.

However, to obtain a readably formatted output, some little macro frameworking for serial outputs was done on the MSP-430 side. The framework provides macros to log a key plus any formatted characters. The key is thought to provide information about where the data comes from (e.g. 'routing table'), and the data is just the data which would just be delivered to `printf`. Also, additional logging lines without a key (but for the same, previously set key), can be done. The loggong macros format the output by inserting space characters after the key, so that the whole output can be read in two columns.

Because of the vast overhead, which serial data output brings, the whole logging can be turned off at compile by a macro definition. However, although logging may be turned on, they can be turned off at runtime by software. Of course, this can only be achieved by putting even more overhead on every logging call, that is, by checking before each output if the output is actually turned on.

B. AODV Implementation

1) Project Organisation: The project sources are organized similar to the example of the ScatterWeb implementation itself. The filenames of the header and source files imitate *namespaces*. The basic namespace is the 'package' **aodv**. Every header file, which is part of the present AODV implementation therefore has the naming scheme `aodv.<names>.h`, source files accordingly. Sub-packages are also used.

The main starting point within the ScatterWeb operating system for a client application (such as the present AODV implementation is a client application) is the file `ScatterWeb.Process.c`. ScatterWeb will inform user

applications over the function `void Process_init()`, that the module is started and initialized. Hoever, the ScatterWebs timer facility has not yet setup at this point. Perhaps because the DCO (digital controlled oscillator) is still adjusting itself, it takes several **seconds** until the first timed procedure will be invoked by ScatterWeb. To capture this event, the initialization is extended by placing the call

```
LED_ON;
Timers_add(1, (fp_timer) reportStarted, (void*)
      0xFFFF);
```

into the `void Process_init()` function. After an ideally delay of 1ms, but realistically after 10 seconds, the function

```
void reportStarted(void) {
  LED_OFF;
  // The system is ready to run
  // ...
}
```

is called and it is ensured that the module has started up, including the timer facility.

2) ScatterWeb Integration: The entry point for RF applications on ScatterWeb is the function `bool Process_radioHandler (struct netpacket_handler_args* args)`. This handler is called whenever a complete data frame has been received by the `CC1020` transceiver and transmitted to the `MSP-430` controller. **Any** data frames will be delegated to this handler. We actually can speak about such *frames* in terms of *layer-2* frames. As discussed aboce, the basic information such frames provide is their origination, destination and type.

To develop a specific RF application, first the layer-2 packet types must be declared. So far, the following packet types are already declared by ScatterWeb:

```
1  enum packettypes {
2    // Packet Type: Acknowledgement packet.
3    ACK_PACKET,
4    // Ping packet
5    PING_REQUEST_PACKET,
6    // "Pong" packet
7    PING_REPLY_PACKET,
8    // Command request packet
9    MESSAGING_REQUEST_PACKET,
10   // Command reply packet
11   MESSAGING_REPLY_PACKET,
12   // text messages for output by the observer
         node
13   DEBUGLOG_PACKET,
14   // First number for user defined packet types
15   USERDEFINED_PACKET
16 };
```

Listing 4. ScatterWeb layer-2 frame types

To avoid conflicts between the participating groups of the course, numbering conventions have been agreed to. So for this protocol implementation, the range frame type number range `50-59` could be used. Listing 5 shows, how the frame types for this implementation are defined.

In the above mentioned ScatterWebs frame handler method `Process_radioHandler`, the *AODV* application as well as test programs can decide whether an incoming data frame is relevant to them, and how the data needs to be processed.

```
1  #define MY_GROUP   50
2  enum userPackettypes_e
3  {
4    USERUNDEFINEDPACKET = USERDEFINED_PACKET +
         MY_GROUP
5    , TRAFFIC_GENERATOR_PACKET
6    , CONNECTION_TEST
7    , AODV_TEST_PACKAGE
8    /* AODV-Frames. */
9    , AODV_CONTROL_MESSAGE
10   , AODV_DATA_PACKAGE
11 };
```

Listing 5. AODV layer-2 frame types

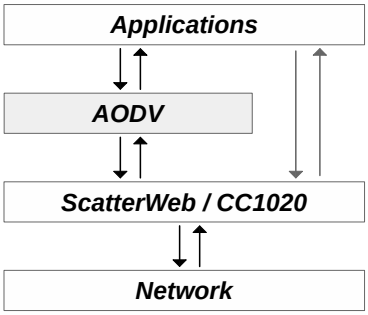

Fig. 1. AODV Integration Design

3) The Protocol Stack: As briefly mentioned earlier, the simple network protocol provided by the ScatterWeb operating system is treated as a layer-2 1-hop network protocol. The nodes 8-bit network addresses (as defined by the data type `netaddr_t`) are seen as `MAC-addresses`. *AODV*, as it is defined in the RFC [7], however uses a layer-3 network protocol to be implemented atop.

This implementation of *AODV* does not need a layer-3 infrastructure, especially IP, to work on. Protocol layers are slightly mixing up in this implementation, as node addresses (actually being seen as layer-2 addresses) are used also as sort-of layer-3 addresses. The present implementation of the AODV routing protocol is actually a tiny layer-3 protocol itself. It just also uses the existing layer-2 network addresses.

The only thing missing from the IP protocol is actually the *TTL* facility. These time-to-live values, which are decreased by every intermediate hop, are needed by AODV to realize the *expanding ring* search technique. So this *TTL* feature was simply added to this AODV implementation, see section V-B5 for details.

4) AODV Design: The AODV routing protocol application is designed to easily offer other applications on the sensor nodes to simply send and receive arbitrary data. However, changes to the ScatterWeb core weren't desired, so that applications could still use any existing features. That is in particular the so called layer-2 point-to-point frame transmission. Figure 1 shows, how AODV uses ScatterWeb facilities as any other application would, but also offers facilities to any application.

The interface, which the AODV routing application provides to clients, is defined in the header file `aodv.interface.h`. There are quite many functions defined, but the basic interface

consists of the following functions:

- `void initAODV(void)`
  Initializes the application once. If needed, memory is prepared and timers are started.
- `void sendData(destination, data, dataSize, flags, application, callbackHandler)`
  The interface for applications to send data to a specific `destination`. The `flags` parameter lets the application chose AODV specific details, such as that a bidirectional connection is desired. The `application` parameter is used to specify an application identifier, so that the receiver can match the incoming packet to a receiver application. The `callbackHandler` attribute is a function pointer to a function, which will be called after the transmission of the data has been made or has failed.
- `bool addDataReceivedListener(aListener)`
  This allows applications to register a handler function, which will be called for every data packet, which was received over the AODV routing protocol. There also exists a corresponding `bool aodv_interface_removeDataReceivedListener(aListener)` function to remove a registered handler function.

This tiny interface already fully enables applications to communicate using the AODV routing protocol implementation. The interface however offers some few more functions - for testing purposes three more types of listeners can be registered to capture events during AODV operation:

- `bool addRREQReceivedListener(aListener)`
  Registers a listener which will be called whenever a *route request* is received by this node.
- `bool addRREPReceivedListener(aListener)`
  Registers a listener which will be called whenever a *route reply* is received by this node.
- `bool addPacketForwardListener(aListener)`
  Registers a listener which will be called whenever a data packet is received and forwarded to another node.

Corresponding 'remove-listener' functions exist. These two functions enable test applications to intervene the AODV routing process. Logging of data and even changing the payload-data in the 'forward-packet' handler can be realised.

The registration/unregistration of all of listeners is realized using the list framework (V-A2).

5) Data Types: The data types for the control messages used by this AODV implementation are defined strictly according to the AODV-RFC [7]. These messages are defined in the header file `aodv.messageTypes.h`. Table I shows an overview over the AODV data types and their matches to the implementation definitions. There were just spare modifications made to the data types as they were proposed in the RFC. Some `reserved` and `flags` fields were not used by this implementation, but left in the data definition to keep/achieve alignments of the remaining attributes inside the struct.

In the RFC, the basic datatype size was based on 32 bit. Sequence numbers and node addresses (called IP-addresses in the RFC) were therefore be sized to be 32 bit long. The ScatterWeb however uses only 8 bit node addresses. Now this AODV implementation uses 8 bit node addresses to be compliant with ScatterWeb, but 16 bit for fields which may require larger values, such as sequence numbers and route request IDs. (16 bit is the microcontrollers natural register size).

TABLE I
AODV MESSAGE TYPES

Message	Intention	Type definition
RREQ	Route Request	`aodv_RREQ_message_t`
RREP	Route Reply	`aodv_RREP_message_t`
RERR	Route Error	`aodv_RERR_message_t`
RREP-ACK	RREP Acknowledgment	`aodv_RREPACK_message_t`

```
1  typedef struct {
2    netaddr_t destination;
3    netaddr_t originator;
4    uint8_t ttl;
5    uint8_t flags;
6    uint16_t application;
7  } aodv_header_t;
8
9  typedef struct {
10   aodv_header_t header;
11   const void *payload;
12   uint16_t payload_size;
13 } aodv_dataPacket_t;
```

Listing 6. AODV data packet

The `RREQ` message was added a `TTL` field. This is necessary for the utilization of the *expaning ring search technique* for route requests. Instead of using the layer-3 protocol IP (it doesn't exist on ScatterWeb), this simple addition to route request messages solves everything which AODV needs from this stack layer. The `TTL` value is decreased by every intermediate hop. When route request messages arrive at a node with `TTL=0`, they will be dropped.

There is a 'generic' type definition `aodv_ControlMessage_t`, which only consists of the attribute `uint8_t type`. Because all of the structs for RREQ, RREQ, RERR and RREP-ACK share the same first attribute, any AODV `control message` can be cast to this generic type to check, of what particular type the control message is.

The other AODV data type is the one used to transmit data over AODV. It is defined in the type definition `aodv_dataPacket_t`, and utilizes a sort-of layer-3 `packet`. As explicitely pictured in listing 6, such `AODV-Packets` stores payload and a header. The `header` is similar to an *IP-Header*: It stores the originator and destination addresses, an application identifier, some flags (for AODV) and the `TTL` value.

6) Buffers & Tables: AODV requires different data to be stored during it's operation. The required data buffers, their organisation and realization are depicted briefly next.

- **Payload data**
  When AODV is faced with sending data to a node, which it does not have a route available for, it initializes a *Route Request*. During the request - i.e. until a suitable route reply message arrives - the data has to be stored by AODV for a certain amount of time. As AODV lets applications

pass a callback handler, which will be called when the data has been sent or timed out, this callback handler must also be buffered.
This implementation uses an own buffer data type to store all necessary data, and the *List framework (→V-A2)* to realize the required buffer: When there is data to be stored, a struct of the buffer data type is created and stored in an newly created list entry. Whenever AODV has new information (e.g. `RREQ`s have arrived), the list can be searched for data that is ready to be sent now. Buffer entries can be easily deleted from the list when they have been sent or timed out. The detailed structure of this buffer is further described in section *V-C2*.
In the ScatterWebs timer framework, the list entry pointer can be used to point to a specific buffer entry, to process it in any routine after an amount of time (e.g. timeout deletion, retransmission attempt).

- **Routing Table**
  The routing table is the main element of the whole AODV implementation, and the basic part of the AODV operation. The routing table stores route availability information for destinations: It stores, if a route is still available, how up-to-date the route is (→ sequence numbers), and how many hops the destination is away from this node.
  The routing table is also organized in the *List framework (→V-A2)*. The list is easy to traverse (if it was sorted, it would be faster though), and especially the `insert` and `delete` operations can be done in $O(1)$.
  Again, the list entry pointer can be used in the ScatterWebs timer framework to point to a particular routing table entry after an amount of time, e.g. to disable or delete it.
- **RREQ Buffer**
  Every node may react on a `RREQ` that it has received only once to avoid circular rebroadcasts of the same message. To distinguish `RREQ` from one another, `RREQ`s store `RREQ-IDs`, which, together with their originators node address, is an identifying property (apart from arithmetic overflows). Every node therefor needs to remember, which `RREQ`s it has received, so it can ignore duplicate ones.
  To do so, again the *List framework (→V-A2)* is used.
- **Precursor Lists**
  For every routing table entry, every node stores addresses for other nodes, who have been active on this route. Those nodes will be interested in receiving route error messages when the route to the destination is detected to be broken. Those precursor lists are also realised with the *List framework (→V-A2)*.

As shown above, all data organization is reduced to one list framework, which uses dynamic memory allocation. Going this way, there is not too much care for the memory left during the implementation. The downside of the current implementation (with the current lists) is, as mentioned in section *V-A1*, that once the available heap memory is all allocated, there is neither stack space left nor can more, possible more important data than the already bufferd one, be stored. Having the list frameork encapsulated as it is, however provides the opportunity to improve just the framework to be more 'managing', and leaving the implemention as it is.

7) Configuration parameters: This AODV implementation uses the same configuration parameters as given in the RFC. The configuration parameters are static values, which define thresholds, timeouts and sizes for the AODV operation. This implementation blindly trusts the RFC to have the parameters pre-configured in a sensible way. Having the parameters seperated in an own header file within macro definitions, changes to them could be made easily for experimenting purposes.

The configuration parameter definitions can be found in the header file `aodv.configuration.h`.

C. AODV Operation

In this subsection, the basic realization concepts for getting AODV running are described. These are in particular the event triggered procedures, which arise during the AODV operation. First, a short overview is given over the general functioning of *AODV*.

AODV keeps a routing table, in wich next-hop nodes are stored for known destinations. When AODV is asked to transmit a data packet, the data is transported hop-by-hop over these next-hop nodes towards the destination. When AODV doesn't have a routing table entry for a destination, it starts the *route discovery*. This is done by broadcasting a *route request (`RREQ`)* message into the network. `RREQ` messages are re-broadcasted by other nodes, so that after a while the desired destination can receive the message. A *route reply (`RREP`)* message is unicastly sent back to the requestor of the `RREQ`. If the `RREP` arrives at the requestor, the route discovery procedure has terminated. If no `RREP` message arrives at a node in response to a route request, the procedure terminates after a timeout. More precise, *AODV* uses several attempts of route discovery with increasing `TTL` value, meaning increasing broadcast range. Therefore, the route request procedure terminates after a bunch of timeouts. The data, which was wanted to be transmitted when no route to the destination was available, is always stored by *AODV*. Stored data will be deleted when the timeouts for the route discovery have occured, but stored data can be transmitted, whenever a route for the destination of the data becomes available.

Once a route is decteded to exist for a destination, this route is not expected to be much stable. Instead, all routes suffer deactivation and deletion timeouts. After a certain amount of time, a route will be deactivated, when there was no activity observed on it (i.e. the route is used to transmit data, control messages received from the destination, etc.). Again after mor inactivity, a route will be deleted. On the other hand, when a better route to the destination is detected, an existing route is updated. To keep rotes updated, *`HELLO`* messages are used and sent frequently. `HELLO` messages are route replies (to virtual route requests), which are sent as layer-2 broadcasts.

1) Routing Table: The routing table in this implementation is designed according to the RFC suggestions. The routing table consists of entries of the type `aodv_routingTableEntry_t`, which are stored in lists (→ *List framework (V-A2)*). The entry type definition is shown in listing 7.

```
1  struct {
2    netaddr_t destAddress;
3    netaddr_t nextHopAddress;
4    uint16_t destSeqNumber;
5    unsigned validDestSeqNumber :1;
6    unsigned valid :1;
7    unsigned otherFlags :6;
8    uint8_t hopCount;
9    aodv_list_t *precursor_list;
10 } aodv_routingTableEntry_t;
```

Listing 7. The Routing Table

```
1  struct {
2    void *bufferedData;
3    uint16_t data_size;
4    aodv_header_t packetHeader;
5
6    aodv_applicationCallback callback_handler;
7    uint8_t rreq_flags;
8
9    uint8_t nextTTL;
10   uint8_t numRetries;
11   uint16_t nextBackoffTime;
12
13 } aodv_packetBuffer_entry_t;
```

Listing 8. The Packet Buffer

The routing table "interface" allows to **lookup** a destination in the routing table and update the table from incoming messages/packages/frames. The time-event triggered deactivation, reactivation and deletion of routing table entries (=routes) is achieved by using the ScatterWebs timer framework with the pointer to the routing table entry as the identifier. In fact, a routing table entry is **always** threatened of being first deactivated and then deleted by timer events. Activity on the route only restarts those timers on an entry.

2) Data Buffer: Application data (i.e. payload) is buffered by *AODV* until a route becomes available. Infact, this implementation buffers already the complete "layer-3" packet, which would be transmitted when a route was available. Also, the whole 'expanding ring' search technique organization data is stored together with the payload data. Listing 8 shows the structure of the data buffer entry type definition.

This has one massive advantage. The expanding ring search technique works as follows: A route request is performed with a small `TTL` value. This means, that the `RREQ` message will only be re-broadcasted a few hops far. When the route request times out without having a RREP received, the route is again requested with a larger `TTL` value. `AODV` performs repeated route requests with increasing `TTL` value, and then some few more attempts with a maximal `TTL` value. Now the timeouts can be achieved easily with the ScatterWebs timer framework, having the buffer list entry being the identifier and argument. So if all the retransmission parameters (i.e. `TTL` value, number of retry and backoff timeout) are stored together with the data and the callback handler, the retransmissions can be scheduled completely independently. When a scheduled (i.e. timed) data packet retransmission is due with the 'retry number' already at the maximal value, this data packet can be dropped and the callback handler can be called (faulty). On the other hand, when during the *AODV* operation a `RREP` message is received which enables a stored data packet to be transmitted, the transmission can be performed, the callback handler can be called (successfully), and then the stored entry can be deleted. Though if there is still another retransmission scheduled for this (now deleted) data packet, this won't be a problem: When the retransmission is due, the argument will be the pointer to a list entry struct. This list entry has been deleted though and won't be found in the list. The scheduled retransmission will be just dropped then.

D. What's missing

- **Lifetime**
  AODV uses absolute time values which are put into control messages to have such messages being removed from the network after they've been around for a certain amount of time. `RREP` messages for instance would be tagged with the current time plus a constant value (configuration parameter) when it is transmitted. Every node would ignore this message then, if the current time is older than the time value stored in the message.
  Such a mechanism requires all nodes to have synchronized times though. This requirement was absolutely unrealistic to realize for the ScatterWeb nodes, so the "lifetime feature" was not considered and hence not implemented in this implementation.
- **Route Errors**
  Route Error messages, their real effects and execution was so horribly badly described in the RFC, that the implementation expense went far beyond "the calculated time" (i.e. the time that was left for the implementation). Route error messages, although an important part in the *AODV* routing protocol, therefore unfortunately had to be dropped.
- **HELLO's**
  `HELLO` messages appeared to break the actually working *AODV* implementation by over-gossipping the network. No other traffic was possible, also with a hello-rate of one `HELLO` message per second. Also, those `HELLO` transmissions seemed overcharge the nodes. So the usage of `HELLO` messages was implemented to be an optional feature (by pre-compiler definition), but for the tests it was left turned off.
- **Blacklists**
  To avoid unidirectional link problems, every node can keep blacklists for neighbouring nodes, of which it is known that the connection is somehow unidirectional. This implementation hasn't got far enough to implement the blacklist feature.

VI. Tests

A. Preparations

1) Node Set-Up: To be able to realize comprehensable and ideally also reproducable behavior of the routing protocol implementation on a real field of nodes, the set up and arrangement of them needed much consideration. In an ideal environment, it is known and clear, which nodes are connected to which other nodes. In reality, it was absolutely impossible

to obtain such conditions. Furthermore, once a behavior was acceptably settled, and a setup of nodes could be made, it seemed impossible to reproduce the simplest setup. Also, the signal conditions seemed to be able to change dramatically just from one moment to another. Also, when a setup of nodes was reached that would fulfill the star topology for example, this would have been only testet using the ScatterWeb `COMMAND` messages. When sending real data, the reachability of the nodes was sometimes observed to be completely different when it was with small command frames.

2) Antennas: The sensor boards which were available for the tests were equipped with different antennas: some of them had antenna jacks (for putting on a real antenna), some had wires of about a half lambda length, and some weren't equipped with an antenna. It turned out that the choice of the antenna was an essential part for achieving good setups of the nodes for the tests.

The half-lambda wire-antenna was just too powerful in signal strength. The difficulty with strong signals is to get sensor nodes to not hearing each other to acchieve a more-than-one-hop connections. The antenna-jacks (which work as antenna as well) showed pretty well signal strength behaviour. The signals of those antennas seemed to be some sort of ´robust', i.e. it was easier to reproduce connectivity behaviour with those antennas. The signal range was about 30-50cm, which is large (for tests!), though still acceptable. Because only a few nodes were equipped with these antenna jacks, and because they are too expensive to upgrade, they couldn't be used. Instead, all nodes which were used for the tests were updated to have tiny wire antennas of length of a quarter or a sixteenth of lambda. These tiny antennas had a nice signal range, which could be adjusted with the CC1020s `txpwr` settings. The signal consistency though was still very poor: Sometimes, a node would reach another node all through the room (despite little `txpwr` setting), but couldn't reach 10 nodes standing next to it. A while later the behavior might be just completely different.

B. Shocking Experiance

The most work in the "test field" was actuall spent on arranging the nodes to get proper test setups. This appeared impossible due to the very unpredictable behaviour of the hardware. Most work was done in writing test programs, which should help to manually physically arrange the nodes, so that one node would only have two neighbours in a linear topology for example. Also, a node should of course "hear" his neighbours! Many attempts for helpful `COMMAND` programming were made to sort-of 'visualize' the range of the nodes. Something like that was not really possible to realize with only the one red LED on the devices.

When a topology was achieved, that would guarantee a three or four hop linear topology with `LED-on, LED-off` test routines, the shocking experiance was, that this topology would explicitely not work with *AODV*. The experiance was, that it is AODV's natural behaviour, to chose always the **worst** path for a destination! This may sound weird indeed. *AODV* indeed thinks, that it choses something good, but under the tough conditions in the real test field, the choices made by AODV are just not usable.

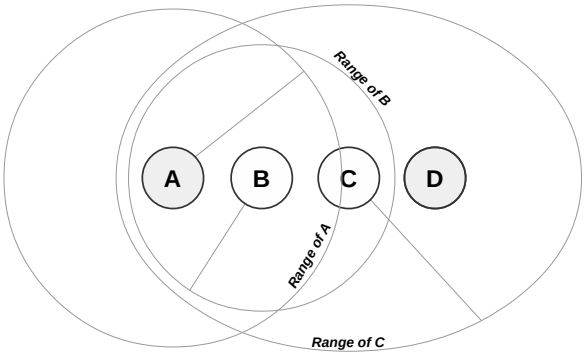

Fig. 2. AODV Range Races

Let's explain this. When *AODV* 'hears' a node, which is closer to a destination than another one is, this node will be the preferred neighbour for the destination. Although a node may have a perfectly working route to a destination, *AODV* will drop this route, if it hears from a closer node just only once. The horrible mistake is though, that under the poor wireless conditions concerning signal quality, this closer node is *very likely* to be a worse choice. That is, because the further a node is away, the even worse the signal quality becomes, and therefore the even much worse the route becomes. Put differently: Comparing two routes to a destination, the one with the more hops is likely to be the better one. Of course, this shouldn't be put into meanings such as "the longest route ist the best", because the longest route may containg a million circles.

An example for this "range race condition" is illustrated in figure 2. There are nodes A, B, C and D, while A wants to discover a route to node D. In A's range is node B, and also partly node C. "Partly" may mean 'with poor signal quality' or 'only sometimes'. On the other side though, node A is in good range of node C. So far: Node C can hear node A only sometimes, and node A can mostly hear node C. When node A starts a *route discovery* for node D by sending a `RREQ` message, nodes B will receive it. B may have already a 2-hop-route to D available over node C, and would answer with a `RREP`. Node A then has a 3-hop-route to D over B. However, also node C may receive A's `RREQ` message, and because C has a 1-hop-route available to node D, it answers with a `RREP` message. Node A receives the `RREP` message, and will replace the existing 3-hop-route with the **supposedly better** route, namely a 2-hop-route.

VII. Conclusions

AODV poorly failes on simply achieving even only 4-hop connections, because it is so very greedy with making as little hops as possible. All *route maintainance* features AODV has would only please ping-pong fans: Sending `HELLO` messages to keep routes active will be also received and processed by weak links. Of course, *route error propagation* may often solve the problems, but only until another next weak next-hop makes it into the routing table again.

The only feature which would promise some hope, is the *blacklist feature*, which avoids unidirectional deadlock problems. However, the main problem of *AODV* seems to lay in the poor judging and picking of best neighbours.

REFERENCES

[1] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," RFC 3626 (Experimental), Internet Engineering Task Force, Oct. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3626.txt

[2] D. Johnson, Y. Hu, and D. Maltz, "The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4," RFC 4728 (Experimental), Internet Engineering Task Force, Feb. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4728.txt

[3] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 234–244, 1994.

[4] Z. J. Haas, M. R. Pearlman, and P. Samar, "The zone routing protocol (zrp) for ad hoc networks," Cornell University, Internet-Draft draft-ietf-manet-zone-zrp-04, July 2002, expires in six months on January 2003.

[5] M. Günes and O. Spaniol, "Ant-routing-algorithm for mobile multi-hop ad-hoc networks," *Network control and engineering for Qos, security and mobility II*, pp. 120–138, 2003.

[6] M. Günes, M. Kähmer, and I. Bouazizi, "Ant-routing-algorithm (ara) for mobile multi-hop ad-hoc networks - new features and results," *In Proceedings of the 2nd Mediterranean Workshop on Ad-Hoc Net- works (Med-Hoc-Net'2003)*, June 2003.

[7] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," RFC 3561 (Experimental), Internet Engineering Task Force, Jul. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3561.txt

[8] I. Chakeres and C. Perkins, "Dynamic MANET On-demand (DYMO) Routing," Internet Engineering Task Force, Internet-Draft draft-ietf-manet-dymo-14, Jun. 2008, work in progress. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-14.txt

[9] I. D. Chakeres and L. Klein-Berndt, "Aodvjr, aodv simplified," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 3, pp. 100–101, 2002.