Tymo

From TinyOS Wiki

Contents

TYMO: DYMO on TinyOS

TYMO is the implementation on TinyOS of the DYMO protocol, a point-to-point routing protocol for Mobile Ad-hoc Networks (MANETs). It was initially designed by the IETF to find routes dynamically on top of the IP stack. In TYMO, we changed its packet format and implemented it on top of the Active Message stack of TinyOS.

DYMO: Dynamic MANET On-demand

As a reactive protocol, DYMO does not explicitly store the network topology. Instead, nodes compute a unicast route towards the desired destination only when needed. As a result, little routing information is exchanged, which reduces network traffic overhead and thus saves bandwidth and power. Also, since little routing state is stored, DYMO is applicable to memory constrained devices like motes.

When a node needs a route, it disseminates a Route Request (RREQ), which is a packet asking for a route between an originator and a target node. The packet is flooded to the entire network or within a number of hops from the originator. When the packet reaches its target (or a node that has a fresh route towards the target), the node replies with a Route Reply (RREP). A route reply packet is very similar to a route request, but it follows a unicast route and no reply is triggered when the target is reached.

When nodes receive a RREQ or a RREP, they cache information about the sender at the link level and the originator, so that they know a route to the orginator that can be used later (if it is fresh enough) without sending a RREQ. The nodes have the possibility to accumulate the path followed by the packet in the packet itself. So, when nodes disseminate a RREQ or RREP, a lot of information can actually be obtained from the packet, much more than a route between two nodes.

When routes have not been used for a long time, they are deleted. If a node is requested to forward a packet through a deleted route, it generates a Route Error (RERR) message to warn the originating node (and other nodes) that this route is no longer available. As another route maintenance mechanism, DYMO uses sequence numbers and hop counts to determine the usefulness and quality of a route.

MH

The purpose of DYMO is to find routes on demand. To transport data along these routes, we designed and implemented a very simple transport protocol that we simply called MH for MultiHop. It does nothing more than forwarding packets unless the receiving node is the target of the packet.

Usage

If you are familiar with the TinyOS Active Message stack, sending and receiving MH messages should be fairly easy, since this is done through the same interfaces.

Interfaces, Components and Wiring

To send and receive MH messages, you need to declare the usage of a couple of interfaces:

```
module TestM {
  uses {
    interface SplitControl;
    interface AMPacket as MHPacket;
    interface Packet;
    interface Receive;
    interface AMSend as MHSend;
  }
}
```

- The SplitControl lets you start and stop the MH and DYMO services, which is required to use them afterwards.
- The AMPacket and Packet interfaces let you fill a packet before sending it or read the content of a received packet.
- The Receive and MHSend interfaces let you receive and send MH messages in exactly the same way as TinyOS link layer packets.

To ease the work of the user, we created a single component that wires together all the modules in order to provide a MH service that relies on the DYMO service to route packets. This component is called DymoNetworkC. Just as TinyOS does with actives messages on the link layer, DymoNetworkC provides an array of Receive and Send interfaces so that several applications can use MH independently. As a result, to wire your code to DymoNetworkC, you need a configuration similar to:

```
implementation {
  components TestM, DymoNetworkC;

  TestM.SplitControl -> DymoNetworkC;
  TestM.Packet       -> DymoNetworkC;
  TestM.MHPacket     -> DymoNetworkC;
  TestM.Receive      -> DymoNetworkC.Receive[1];
  TestM.MHSend       -> DymoNetworkC.MHSend[1];
}
```

Code example

Once this is set up, you can send and receive messages just as regular AM messages, for example (you need to set up the Boot and Timer interfaces in addition to the above for this to work):

```
    message_t packet;

    //MH and AM addresses are the same
    enum {
      ORIGIN = 1,
      TARGET = 6,
    };


    event void Boot.booted(){
      call SplitControl.start();
    }

    event void SplitControl.startDone(error_t e){
      if(call MHPacket.address() == ORIGIN){
        call Timer.startPeriodic(2048);
      }
    }

    event void Timer.fired(){
      nx_uint16_t * payload = call Packet.getPayload(&packet, NULL);
      error_t error;
      *payload = 1664; //Define the content of your message as you wish
      error = call MHSend.send(TARGET, &packet, sizeof(*payload));
      if(error == SUCCESS){
        //Good!
      } else {
        //Something's wrong...
      }
    }

    event void MHSend.sendDone(message_t * msg, error_t e){
      if((e == SUCCESS) && (msg == &packet) && (call MHPacket.address() == ORIGIN)){
        //Even better!
      } else {
        //The packet couldn't be sent!
      }
    }

    event message_t * Receive.receive(message_t * msg, void * payload, uint8_t len){
      if(call MHPacket.address() == TARGET){
        //Message received!
      } else {
        //This shouldn't happen...
      }
      return msg;
    }

    event void SplitControl.stopDone(error_t e){}
```

Configuration

There are various preprocessor variables that can be set to alter the characteristics of DYMO. The list and default values can be found in lib/net/tymo/dymo/dymo_routing.h. Here is the meaning of each of them:

- MAX_TABLE_SIZE: Number of entries the routing table can store. When this number is reached, the oldest route is deleted if a new route needs to be added.
- DYMO_HOPLIMIT: Number of hops a DYMO packet can go through before being dropped.
- DYMO_ROUTE_AGE_MAX: Maximum amount of milliseconds a route can be kept.
- DYMO_ROUTE_TIMEOUT: Maximum amount of milliseconds a route can be kept without being used.
- DYMO_APPEND_INFO: 1 to append info to forwarded routing messages, 0 otherwise.
- DYMO_INTER_RREP: 1 to allow intermediate RREP, 0 otherwise.

- DYMO_FORCE_INTER_RREP: 1 to send intermediate RREP even without target's seqnum in the RREQ.
- DYMO_LINK_FEEDBACK: 1 to use acks to detect broken links.

Please refer to the DYMO specifications for more information on this characteristics and their role.

Limitations

This project cannot be considered as stable yet. There are a couple of bugs hidden in the code, and there are various things to accomplish to improve the project:

- Debug the code and improve its efficiency;
- Test more thoroughly the code;
- Update the code to reflect latest changes in the draft specs of DYMO;
- Refactor some pieces of code to be reusable by other protocol implementers;

Developer documentation

Architecture

Overview

TYMO design is based on A Modular Network Layer for Sensornets, a paper that describes a generic layout to implement routing protocols, which is described below.

Original layout

To "ease the implementation of new protocols, by increasing code reuse, and enable co-existing protocols to share and reduce code and resources consumed at run-time", a representative set of various protocols for sensor networks was examined in order to identify their common parts. This made it possible to divide the protocols into several functions, some of which can be shared by all or some of the protocols. This was then used to design a general layout of components that provides a framework for implementing routing protocols.

The layout is divided into two parts: the data plane and the control plane. Implementing the control plane is not surprisingly much more complicated, since it implements the routing algorithms. The functioning of this layout is illustrated in figure 1.

Figure 1: The network layer decomposition.

The Dispatcher examines the header of the packets coming from the lower or upper layer in order to determine the protocol to which the packet belongs, and passes the packet to the appropriate protocol service. The latter is a set composed of a Forwarding Engine, a Routing Engine and a Topology Engine.

Though the Forwarding Engine is part of a protocol service, it is not aware of the protocol format and algorithms. It simply requests the Routing Engine to fill the routing header of a packet before forwarding it, or deliver the packet to the upper layer when the packet has reached its destination. The reason why the Forwarding Engine belongs to the protocol service is that it may perform packet aggregation or scheduling, and these tasks depend on the protocol.

The Routing Engine and the Topology Engine are the core components of a protocol: while the Routing Engine generates and processes control packets, the Topology Engine computes and stores the necessary information about the network topology, according to the data reported by the Routing Engine.

Finally, the Output Queue handles the packets to be sent from all the protocols running on the node. Since all packets must go through this component to be sent, the Output Queue can schedule them according to the node policy.

This earlier work provided us not only a good starting point to implement the DYMO protocol, but also general guidelines to ensure that our work is generic enough to be reused by the research community. Indeed, some parts of our implementation are not related to DYMO, and were implemented only because TinyOS does not provide them yet. Our aim is that these parts will be useful for other protocol implementers.

Implemented layout

The goal of the implementation is to provide a component to an application in order to transparently send and receive data in a multi-hop network. We have called this component DymoNetworkC, which is a configuration. The wiring provided by this configuration is illustrated in figure 2.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-zz2xEO03' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_a3e215621043-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 2: General layout of the components. To avoid complicating the diagram further, the Packet interfaces are not represented. They would have appeared each time another type of packet is used or provided.

Since DYMO is a routing protocol, the configuration must include a transport protocol to transport data on multi-hop routes. It can be any transport protocol using the same address format as DYMO, a 16-bit address in this case. In this document, we refer to the transport protocol as MH (for Multi-Hop). The configuration also provides the application with the possibility to inspect all the multi-hop data packets that travel through this node, and to decide if they should be forwarded. This is done via the Intercept interface.

To be used, the network layer need to be started with the SplitControl interface. This is implemented by a dedicated module, NetControlM, which waits for all other components to start before letting the application use the network layer (ActiveMessageC implements the link layer). The application can then send and receive MHPackets, which can be manipulated with the MHServiceC component.

The DymoServiceC and MHServiceC components are the protocol services: they are responsible for all the processing and packet manipulations related to their respective protocol. Both of them have their own sending and receiving queue, an instance of AMSenderC and AMReceiverC. This does not break the single Output Queue principle we have seen above: though it is not represented on the diagram for simplicity reasons, these queues rely on ActiveMessageC to exchange packets with the radio chip. It is this component that actually plays the role of the Output Queue, and it uses the parameterized wiring feature of NesC to deal and gather the packets to the AMReceiverC and from the AMSenderC components. This is also why there is no need for a Dispatcher component. The ActiveMessageC component provides the link-layer feedback as well: it is possible to request hardware acknowledgements for each packet sent and thus determine if the neighbor received the packet.

To make things clearer, the sequence diagram in figure 3 shows the interactions between components when the application wants to send a packet to an "unknown" node. The application is unaware of the routing operations, it thus sends the packet as if it was a single-hop packet. The packet is given to the MHServiceC component which does not know how the routing protocol operates, but is aware that obtaining a route may not be immediate. Therefore, when the routing table (which is shared by both of the protocol services) signals that the route is not available yet, the send command returns and the MH service will retry regularly to send the packet. In the meanwhile, the routing table signals to the DYMO service that a route is needed, and a route request is issued. When the route reply arrives, the routing table is updated, so that the next try from the MH service will be successful. The data packet is eventually sent to the next hop on the route, and the sendDone event is signaled to the application, so that it can reuse the packet buffer.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-SlW1CRk1' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_4babdfd63bbf-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 3: Sequence diagram for the sending of a packet triggering a RREQ. DymoTableC is shared by DymoServiceC and MHServiceC.

The DYMO Service

The DymoServiceC (figure 4) does not exactly follow the modular layout presented above. Indeed, it has a Routing Engine (the DymoEngineM and DymoPacketM components) and a Topology Engine (the DymoTableC component), but no Forwarding Engine. The main reason is that it would have added useless complexity. Since upper layers are not interested in DYMO packets, the delivering functionality of the Forwarding Engine is not needed. Furthermore, the DymoEngine is the only component that sends DYMO packets, therefore the Forwarding Engine would not need to request the DymoEngine to select a route, since it would have already been selected by the DymoEngine.

As a consequence, the DymoEngine is directly connected to the AMSend and Receive interfaces, and it handles the received packets.

Since processing a packet can take a long time, it is implemented as a split-phase operation, illustrated in figure 5. When a DYMO packet is received, it is given to the DymoPacketM module, which returns immediately and posts a task to read the packet. Each piece of information found in the packet is given to DymoEngineM via an appropriate event. The event handler uses the routing table to judge the usefulness of the information, and decides accordingly if the information should be propagated. It returns its decision to the DymoPacketM module, which in parallel constructs the packet to be forwarded.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-_U6Sb6l7' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_28e359684b48-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 4: Layout of the DYMO service component.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-ihTZt34d' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_57f13a4bcf7b-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 5: Sequence diagram of the processing of a DYMO packet.

The MH Service

The routes determined by the DYMO protocol need a multi-hop transport protocol to be used. Though we did not need such a protocol to implement DYMO, we need one to test and evaluate the implementation. Since no such protocol was available in the TinyOS 2.0 distribution, we implemented a very simple one. Implementing such a protocol also allowed to provide a directly usable multi-hop network layer to applications.

The protocol actually implements the Active Message interfaces on top of the existing Active Message stack.

Contrarily to the the DYMO service, the MH service (figure 6) does have a Forwarding Engine, which is actually more complicated than the control plane. When a MH packet is received from the AM layer or sent by the application, the Forwarding Engine requests MHEngineM to fill the AM fields (and the MH fields if necessary) in order to put the packet on the route toward its target. Given that the route may be unknown and that we are working with a reactive routing protocol, the Forwarding Engine does not discard the packet if no route is available. Instead, it puts it in a waiting queue and regularly retries to request the route. If the RREQ issued by the DYMO service is successful before a certain timeout, the packet is finally given to the sending queue. Since it does not have any functionality specific to the MH protocol, the Forwarding Engine was made as generic as possible and does not rely on any MH-specific interface. It may therefore be used by other protocol services.

The MHEngineM module is almost trivial. Unless the packet has reached its target, it requests the routing table for a route. If one is available, the packet header is updated and the Forwarding Engine can send it, otherwise the Routing Engine tells the Forwarding Engine to wait.

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-SpGpMRJj' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_21c70b582605-1.png' @ error/convert.c/ConvertImageCommand/3011.

Figure 6: Layout of the MH service component.

The Routing Table

The routing table is implemented by the DymoTableC component. Though it appears in the wiring of DymoNetworkC, DymoServiceC and MHServiceC, it is of course the same instance. The DymoTableC stores known routes, that is mainly a target address, a next hop, a sequence number and a hop count. Each routing entry is attached to several timers as suggested by the DYMO specifications to monitor the routes.

Routing information is retrieved from the table via the RoutingTable interface, a generic interface for routing tables (described below). The DymoEngineM module has more control thanks to the DymoTable interface, to update the table and know when a route is needed, so that a route request can be issued.

Routing Interfaces

To compose the network layer provided by our implementation and let components communicate with each other, a number of new interfaces were needed in addition to those provided by the TinyOS distribution. This section presents these interfaces.

Routing Table Interfaces

Due to the fact that the routing table is shared by two protocols with different purposes, two different interfaces to manipulate the routing table were needed.

RoutingTable

The first interface (figure 8) is a generic interface that could be used for other routing tables. It provides access to the information stored in the routing table through the getRoute or getForwardingRoute commands. The first one is called to send a packet while the second one is called to forward it. Two different commands are needed because some protocols take different decisions depending on whether the packet is sent or forwarded. DYMO is one of them: when a route is unknown, a RREQ is generated if the packet is to be sent, but a RERR is generated if the packet is to be forwarded.

Since being too generic would also mean too much complexity, the interface only applies to unicast routes. As a result, these commands only take an address as a parameter, in addition to the memory address of where to store the result of the command. They return a code to specify if the route exists, if it will soon (i.e., if a route request is pending), or if it is broken.

A user of the routing table can also be informed when a route is deleted from the table (in case it is relying on this route). This can happen when the route was replaced by a new one because the table was full, when the route become too old, or when a broken link is detected. This information is obtained via the evicted event.

Routing information is represented via the rt_info_t structure (see figure 7). It is therefore up to the implementation of the routing table to define this type, as well as the reason_t type. In our implementation, route entries may contain any piece of routing information that a DYMO packet can transport, plus the next hop on the route.

```
typedef struct {
  addr_t address;
  addr_t nexthop;
  seqnum_t seqnum;
  bool has_hopcnt;
  uint8_t hopcnt;
} rt_info_t;

typedef enum {
  REASON_FULL,
  REASON_OLD,
  REASON_UNREACHABLE
} reason_t;
```

Figure 7: Types associated with routing tables.

```
interface RoutingTable {
  command error_t getRoute(addr_t address, rt_info_t * info);
  command error_t getForwardingRoute(addr_t address, rt_info_t * info);
  event void evicted(const rt_info_t * route_info, reason_t r);
}
```

Figure 8: The RoutingTable interface.

DymoTable

The second interface (see figure 9) is specific to DYMO, and provides more information and control to the users. The two goals of this interface are to fill and update the routing table and to be aware of the needed routes.

The first goal is achieved with the update command. According to the parameters and the content of the table, the command decides if the information is better than what is available and updates the table accordingly. It is therefore a command called each time a piece of routing information is found in a DYMO packet.

The second goal is achieved with the two other commands: routeNeeded and brokenRouteNeeded. The first one is called whenever a route needs to be discovered, that is when a node wants to send a packet to an unknown route; while the second one signals that a route was expected but it is broken or absent, thus requiring a RERR. Since the DYMO engine relies on the routing table to find routes, it does not need to determine if a RREQ or a RERR is needed: relevant signals will be triggered by the routing table via the DymoTable interface.

```
interface DymoTable {
  command void update(rt_info_t * route_info);
  event void routeNeeded(addr_t destination);
  event void brokenRouteNeeded(const rt_info_t * route_info);
}
```

Figure 9: The DymoTable interface.

Packets

For either DYMO or MH, only one module knows how to manipulate a packet at the bit level. Each time another component wants to read or write into a packet, it must rely on the module. They are two such modules in our implementation, DymoPacketM and MHPacketM, and both of them are used via a dedicated interface.

DymoPacket

This interface (figure 10) is to be provided by a component that knows all the internals of the DYMO packet format.

In order to create or alter a DYMO packet, a component has two commands at its disposal: createRM and addInfo. The first one creates a DYMO message with the minimum amount of information, which is the message header, the target and originator nodes for a routing message, or the first unreachable node for an error message (in which case origin is not specified). Then, if the creator or a forwarding node wants to append additional information to the message, it can use the addInfo command. This command does not specify where (that is, in which block) the piece of information should be added, it is up to the implementer to choose a good place so that the packet size is minimized. This command can fail if the packet has reached its maximum size.

```
01  typedef enum {
02    ACTION_KEEP,
03    ACTION_DISCARD,
04    ACTION_DISCARD_MSG
05  } proc_action_t;
06
07  interface DymoPacket {
08    /* Returns DYMO_RREQ, DYMO_RREP or DYMO_RERR */
09    command dymo_msg_t getType(message_t * msg);
10
11    /* Returns the size of the message */
12    command uint16_t getSize(message_t * msg);
13
14    /* Creates a DYMO message with its heading routing information */
15    command void createRM(message_t * msg, dymo_msg_t msg_type,
16                      const rt_info_t * origin, const rt_info_t * target);
17
18    /* Adds a piece of routing information to a message */
19    command error_t addInfo(message_t * msg, const rt_info_t * info);
20
```

```
21    /* Processes msg and fills newmsg with the message to forward */
22    command void startProcessing(message_t * msg, message_t * newmsg);
23
24    /* The hop values have been read */
25    event proc_action_t hopsProcessed(message_t * msg,
26                              uint8_t hop_limit, uint8_t hop_count);
27
28    /* A new piece of routing information has been extracted */
29    event proc_action_t infoProcessed(message_t * msg, rt_info_t * info);
30
31    /* The message processing is finished */
32    event void messageProcessed(message_t * msg);
33  }
```

Figure 10: The DymoPacket interface.

Reading a packet is completely different. The goal was to be able to easily go through the list of pieces of routing information included in the message, since it was an important purpose of our simplified packet format. As a consequence, accessing a piece of information by its position would not be suitable, because the complexity of processing a message would not be linear. Returning a table with all the information would imply copying a large amount of data, and there is no such thing as iterators in nesC, as proposed by high-level languages. We thus decided to let the DymoPacketM module iterate through the packet and report each piece of information to the user component via appropriate events. This is also illustrated on the sequence diagram 5.

During the packet processing, the DymoPacketM module also builds the message that may be forwarded. For each event reported during the packet processing, the user (that is the DYMO engine) specifies if this information should be kept in the forwarded message. Also, when it has enough information about the processed message, it can request DymoPacketM to stop building the forwarded message if it is useless.

Route Selection

To let the routing engine of the MH service decide what should be done with a packet, the forwarding engine uses the RouteSelect interface (figure 11). It features a single command that will select a route towards a target (using the routing table), and fill the header of the message appropriately. The command modifies the message instead of simply returning the next hop address, so that the forwarding engine does not have to be aware of the packet format and routing options. The returned value specifies what the forwarding engine should do with the packet: sending, dropping, or giving it to the upper layer.

```
typedef enum {
  FW_SEND,      //Put the message in the sending queue
  FW_RECEIVE,   //Give the message to the upper layer
  FW_WAIT,      //Retry later
  FW_DISCARD,   //Discard the message
} fw_action_t;

interface RouteSelect {
  /**
   * Ask the routing engine to fill a message with routing
   * information, in order to send it to target.
   */
  command fw_action_t selectRoute(message_t * msg, addr_t target);
}
```

Figure 11: The RouteSelect interface.
