

NIST DSR Model Appendix

Contents:

<u>Packet Descriptions</u>	3
<u>Data Packet</u>	3
<u>Request Packet</u>	4
<u>Reply Packet</u>	6
<u>Error Packet</u>	8
<u>Function Descriptions</u>	10
<u>Initialization Functions</u>	10
Function dsr_pre_init.....	10
Function dsr_user_parameter_init.....	10
Function dsr_tables_init().....	10
Function dsr_stats_init().....	10
Function dsr_route_init().....	10
<u>Route Discovery Functions</u>	11
Function dsr_transmit_request.....	11
Function dsr_transmit_request_from_error.....	11
Function dsr_handle_request.....	11
Function dsr_request_already_seen.....	12
Function dsr_forward_request.....	12
Function dsr_transmit_reply_from_target.....	12
Function dsr_transmit_reply_from_relay.....	13
Function dsr_handle_reply.....	13
Function dsr_reply_already_seen.....	13
Function dsr_forward_reply.....	14
Function dsr_insert_route_in_cache.....	14
Function dsr_promiscuous_reply.....	14
<u>Data Transmission Functions</u>	15
Function dsr_upper_layer_data_arrival.....	15
Function dsr_transmit_data.....	15
Function dsr_handle_data.....	15
Function dsr_schedule_no_ack_event.....	15
Function dsr_data_already_seen.....	16
Function dsr_forward_data.....	16
<u>Route Maintenance Functions</u>	16
Function dsr_transmit_error.....	16
Function dsr_handle_error.....	17
Function dsr_ckeck_gratuitous_reply.....	17
Function dsr_transmit_gratuitous_reply.....	17
<u>Other Functions</u>	18
Function dsr_in_transmission_range.....	18
Function dsr_insert_buffer.....	18
Function dsr_buffer_empty.....	19
Function dsr_extract_buffer.....	19
Function dsr_no_loop_in_route.....	19
Function dsr_send_to_mac.....	19
Function dsr_message.....	20
Function dsr_end_simulation.....	20
<u>Variable Descriptions</u>	21
<u>The Route Cache</u>	21

The Route Request Table (request_seen & request_sent)	22
The Route Reply Table (reply_seen)	23
The Data Packet Queue (received_packet_id_fifo)	23
The Send Buffer	24
The Acknowledgment Timer Queue (no_ack_fifo)	24
The Scheduled Reply Queue (reply_fifo)	25
Model files	26

Packet Descriptions

Data Packet

This packet is used to transmit data through the network, and is defined in the filename *Dsr_Data.pk.m*.

Type	SRC	DEST	RELAY	Seg_left	Size_Route		
Node_0	Node_1	Node_2	Node_3	Node_4	Node_5	Node_6	Node_7
Data							
TR_Source		Packet_ID					

Field Type

Type: 8 bits integer

Description:

This field contains the type of the packet, encoded as a number. Since the packet is a data packet, this field takes the value of the `DATA_PACKET_TYPE` constant.

Field SRC

Type: 8 bits integer

Description:

This field is used to store the DSR address of the data packet initiator node.

Field DEST

Type: 8 bits integer

Description:

This field is used to store the DSR address of the data packet final destination.

Field RELAY

Type: 8 bits integer

Description:

This field is used to store the DSR address of the next node on the source route contained in the header of the packet. That is, the next “supposed” destination of the data packet.

Field Seg_Left

Type: 8 bits integer

Description:

This field contains the number of hops remaining in the source route in order to reach the final destination. The value is decreased by one when a node forwards the packet to the next node.

Field Size_Route

Type: 8 bits integer

Description:

This field contains the size of the source route (in number of nodes).

Fields Node0, Node1, ..., Node7

Type: 8 * 8 bits integers

Description:

These fields contain the source route leading the data packet from the source node to the destination node. Each node is represented by its DSR address stored in an 8-bit integer, and obviously the route maximal size is 8 nodes, i.e. 7 hops. As a design choice, we choose to consider only routes whose sizes don't exceed 7. However it can be changed easily depending on the network studied.

Field Data

Type: 16 bits integer

Description:

This field is used to store the data provided by the upper layer and that this packet must carry through the network.

Field TR_Source

Type: 16 bits integer

Description:

This field is a hidden field used for the simulation of the transmission range. This field contains the transmitter node Object id, which is used in the receiver through the `dsr_in_transmission_range(...)` function to compute the distance between the two nodes.

Request Packet

This packet is used in order to perform the Route Discovery mechanism and is defined in the filename *Dsr_Request.pk.m*.

Type	SRC	DEST	Seq_Number	Seg_Left	Creation_Time		Size_Route
Node_0	Node_1	Node_2	Node_3	Node_4	Node_5	Node_6	Node_7
TR_Source							

Field Type

Type: 8 bits integer

Description:

This field contains the type of the packet. Since the packet is a Route Request packet, this field takes the value of the `REQUEST_PACKET_TYPE` constant.

Field SRC

Type: 8 bits integer

Description:

This field is used to store the DSR address of the Route Request packet initiator.

Field DEST

Type: 8 bits integer

Description:

This field is used to store the DSR address of the target node of the Route Request.

Field Seq_Number

Type: 8 bits integer

Description:

This field contains the sequence number associated with the request. The Route Request initiator sets this number, which will be used to identify the request.

Field Seg_Left

Type: 8 bits integer

Description:

This field contains the number of hops remaining for the Route Request packet to be propagated. When a node receives a request packet with a Seg_Left field equal to 0, it does not forward it. That is why for a non-propagating request this field is set to 0 by the Route Request initiator. Obviously, each time a node forwards a Route Request packet it must decrease this field by one.

Field Creation_Time

Type: 8 bits integer

Description:

This field contains the creation time of the Route Request packet. It is used in the `dsr_handle_request(...)` function in order to discard Route Request packets that are too “old”. Thus in addition to the Seg_Left field, the Creation_Time parameter is also used to control the lifetime of the request packets.

Field Size_Route

Type: 8 bits integer

Description:

This field contains the size of the route (in number of nodes). Each time a node forwards a Route Request packet, it must update this field by increasing it by one.

Fields Node0, Node1, ..., Node7

Type: 8 * 8 bits integers

Description:

These fields contain the actual route leading from the initiator to the current node. Thus, each time a node forwards a request packet, it must add its own DSR address in the first “route” field that is empty.

Field TR_Source

Type: 8 bits integer

Description:

This field is a hidden field used for the simulation of the transmission range. This field contains the transmitter node Object id, which is used in the receiver through the `dsr_in_transmission_range(...)` function to compute the distance between the two nodes.

Reply Packet

This packet is used to return a route in response to a Route Request packet, and is defined in the filename *Dsr_Reply.pk.m*.

Type	SRC	DEST	Seq_Number	RELAY	Seg_left	Size_Route	
Node_0	Node_1	Node_2	Node_3	Node_4	Node_5	Node_6	Node_7
TR_Source	Reply_From_Target						

Field Type

Type: 8 bits integer

Description:

This field contains the type of the packet. Since the packet is a reply packet, this field takes the value of the `REPLY_PACKET_TYPE` constant.

Field SRC

Type: 8 bits integer

Description:

This field is used to store the DSR address of the Route Reply packet initiator.

Field DEST

Type: 8 bits integer

Description:

This field is used to store the DSR address of the Route Reply packet's target node (which is the Route Request initiator).

Field Seq_Number

Type: 8 bits integer

Description:

This field contains the same sequence number that the Route Request, source of this Route Reply, has.

Field RELAY

Type: 8 bits integer

Description:

This field is used to store the DSR address of the next node that should receive the Route Reply packet.

Field Seg_Left

Type: 8 bits integer

Description:

This field contains the number of hops remaining before reaching the target of the Route Reply. The node must decrease this field by one before forwarding the Route Reply packet to the next node.

Field Size Route

Type: 8 bits integer

Description:

This field contains the size of the source route returned by the Route Reply (in number of nodes).

Fields Node0, Node1, . . . , Node7

Type: 8 * 8 bits integers

Description:

These fields contain the source route leading from the Route Request initiator to the Route Reply target. It is important to note that this path is also used in a reverse way in order to route the Route Error through the network. In these fields, each node is represented by its DSR address stored in an 8-bit integer, and obviously the route maximal size is 8 nodes, i.e. 7 hops.

Field TR Source

Type: 8 bits integer

Description:

This field is a hidden field used for the simulation of the transmission range. This field contains the transmitter node Object ID, which is used in the receiver through the `dsr_in_transmission_range(...)` function to compute the distance between the two nodes (see Section V.4.).

Field Reply From Target

Type: 8 bits integer

Description:

This field is used only for statistical purposes in order to indicate whether the reply was generated by the Route Reply target node or by a relay. It allows the user to evaluate the efficiency of the “reply from relay” mechanism.

Error Packet

This packet is used to inform a data packet generator and all the nodes along the path that there is a broken link in the packet's source route. It is used to perform the Route Maintenance mechanism, and it is defined in the filename *Dsr_Reply.pk.m*.

Type	SRC	DEST	RELAY	Pb_Node	Unreachable_Node	Seg_Left	Size_Route
Node_0	Node_1	Node_2	Node_3	Node_4	Node_5	Node_6	Node_7
TR_Source							

Field Type

Type: 8 bits integer

Description:

This field contains the type of the packet. Since the packet is a Route Error packet, this field takes the value of the ERROR_PACKET_TYPE constant.

Field SRC

Type: 8 bits integer

Description:

This field is used to store the DSR address of the node that detects the broken link and generates the Route Error packet.

Field DEST

Type: 8 bits integer

Description:

This field is used to store the DSR address of the Route Error packet destination node, that is, the data packet generator with the broken link on its route.

Field RELAY

Type: 8 bits integer

Description:

This field is used to store the DSR address of the next node that should receive the Route Error packet.

Field PbNode

Type: 8 bits integer

Description:

This field is used to store the DSR address of the node that has not received the data packet correctly. The couple (field SRC, field PbNode), identifies the broken link.

Field Unreachable_Node

Type: 8 bits integer

Description:

This field is used to store the intended final destination node (its DSR address) of the failed data packet. Thus the couple (field DEST, field Unreachable_Node) identifies, like

a couple (source, destination), the invalid route. The Route Error packet destination, which is the data packet generator, must use this couple to activate a new route discovery process.

Field Seg_Left

Type: 8 bits integer

Description:

This field contains the number of hops remaining before reaching the target of the Route Error. The node must decrease this field by one before forwarding the Route Error packet to the next node.

Field Size_Route

Type: 8 bits integer

Description:

This field contains the size of the route (in number of nodes). See below the fields Node0, Node1, ..., Node7, which define the route.

Fields Node0, Node1, ..., Node7

Type: 8 * 8 bits integers

Description:

These fields contain the route leading from the node that discovered the broken link to the data packet generator. This is a reversed part of the source route assigned to the data packet.

Field TR_Source

Type: 8 bits integer

Description:

This field is a hidden field used for the simulation of the transmission range. This field contains the transmitter node Object ID, which is used in the receiver through the `dsr_in_transmission_range(...)` function to compute the distance between the two.

Function Descriptions

Initialization Functions

Function dsr_pre_init

Header:

```
void dsr_pre_init()
```

Description:

This function pre-initializes the DSR process state machine, that is, initializes the MAC and the DSR addresses, and all Objids used by the node. Note that this address management is done through the dsr_support package.

Function dsr_user_parameter_init

Header:

```
void dsr_user_parameter_init()
```

Description:

This function initializes every parameter defined by the user.

Function dsr_tables_init

Header:

```
void dsr_tables_init()
```

Description:

This function initializes every table and variable using by the routing protocol (such as the route_cache and the request_seen tables).

Function dsr_stats_init

Header:

```
void dsr_stats_init()
```

Description:

This function initializes and declares every statistic that will be calculated and collected during the simulations by the DSR process model.

Function dsr_route_init

Header:

```
void dsr_route_init(sRoute* cache, int n)
```

sRoute cache: the cache in which the route must be initialized

int n: the index of the route (nth) to initialize

Description:

This function initializes the nth route of the sRoute table given as a parameter.

Route Discovery Functions

Function dsr_transmit_request

Header:

void dsr_transmit_request(int destination_dsr_address)

int destination_dsr_address: the dsr address of the Route Request destination node

Description:

This function builds and sends a Route Request packet. It also activates a timer that is the time before renewing a request. When this timer expires this function is called back in order to send a new request. Note that this function includes the non-propagating Route Request mechanism.

Function dsr_transmit_request_from_error

Header:

void dsr_transmit_request_from_error(int destination_dsr_address)

int destination: the dsr address of the Route Request destination node

Description:

After a node has generated and sent a data packet, it may receive a Route Error packet giving notice of a broken link along the route associated with the data packet. Then the node must call this function in order to find a new path to the destination.

This function calls the dsr_transmit_request (...) in order to start a new discovery process. However this function simulates the fact that a non-propagating request has been already sent. Thus the dsr_transmit_request (...) function starts the new discovery mechanism without sending any useless non-propagating request, but by creating and sending directly a new propagating request.

Function dsr_handle_request

Header:

void dsr_handle_request(Packet * pk_ptr)

Packet* pk_ptr: a pointer to the request packet to process

Description:

This function is called when a node receives a Route Request packet. If the node is the destination node specified in the Route Request, then a Route Reply is sent using the dsr_transmit_reply_from_target (...) function. Otherwise, if the node can reply to the request by using information in its Route Cache, the dsr_transmit_reply_from_relay (...) function is called. Finally, if the node cannot reply and if the packet is not too old (time & number of hops), the request is forwarded through the dsr_forward_request (...) function to the node neighbors.

All this processing is done only if the packet is received for the first time, a condition checked by the function dsr_request_already_seen (...), in order to avoid the propagation of redundant Route Request packets in the network.

Function dsr_request_already_seen

Header:

int dsr_request_already_seen (int source_dsr_address, int destination_dsr_address, int sequence_number)

int source_dsr_address: the dsr address of the request source

int destination_dsr_address: the dsr address of the request destination

int sequence_number: the sequence number of the request

Return: 1 if the request packet has been already seen

0 otherwise

Description:

This function returns OPC_FALSE the first time it is called for a Route Request packet identified by its initiator node, its target node, and its sequence number. These parameters are stored in the request_seen table in order to return OPC_TRUE if it is called back with the same parameters.

Note that only the last sequence number received for an origin/destination pair is stored in memory. Every request with a sequence number lower than the one in memory is considered “already seen”, since a request with a lower sequence number must be older than the one stored.

Function dsr_forward_request

Header:

void dsr_forward_request(Packet* pk_ptr)

Packet* pk_ptr: a pointer to the request packet to forward

Description:

This function forwards the request packet to its neighbors. But before this operation it updates the recorded source route by adding its DSR address and by decreasing the allowed maximum number of hops remaining of the Route Request by one (field seg_left)

Function dsr_transmit_reply_from_target

Header:

void dsr_transmit_reply_from_target(Packet* pk_ptr)

Packet* pk_ptr: a pointer to the request packet to reply

Description:

This function builds the Route Reply that the target node specified in a Route Request needs to send to return the route to the request initiator. The node knows the route from the request initiator to itself through the recorded source route contained in the request packet. Note that this path is also used in a reverse way in order to route the reply packet, thus allowing only the use of bi-directional links.

Then, when the Route Reply packet is constructed, it is sent immediately since the reply from target packets have the priority with their entirely confirmed route.

Function dsr_transmit_reply_from_relay

Header:

```
void dsr_transmit_reply_from_relay(Packet *pk_ptr)
Packet* pk_ptr: a pointer to the request packet to reply
```

Description:

For a relay node able to reply by using information in its Route Cache, this function builds the Route Reply packet that the node needs to send in order to return the route to the request initiator. The node computes the route from the request initiator to the target node by adding the route from itself to the target node stored in its cache, to the recorded source route contained in the request packet. Note that this second route is also used in a reverse way in order to route the reply packet. It is also important to notice that the `dsr_no_loop_in_route (...)` function is then called to ensure that the calculated route is loop free. This function removes every loop from the route. That is why a last checking is done to ensure that the current node is still in the path, and was not located within a loop.

When the Route Reply packet construction is finished, a delay is calculated depending on the calculated route size. A timer is activated with this delay, and the packet will be sent only at this timer's expiration in order to avoid reply storms and to ensure that only the shortest reply route will be considered.

Function dsr_handle_reply

Header:

```
void dsr_handle_reply(Packet* pk_ptr)
Packet* pk_ptr: a pointer to the request packet to process
```

Description:

First, this function checks, through `dsr_reply_already_seen (...)`, whether it is the first time the node has received the packet. If this condition is true, then the route contained in the reply packet is used to update the node Route Cache by calling the `dsr_insert_route_in_cache (...)` function. If the node is a relay, it transmits the packet to the next node by using the function `dsr_forward_reply (...)`.

If the node receiving the Route Reply is not in the packet route, then it uses the reply packet in promiscuous mode through the function `dsr_promiscuous_reply (...)`.

Function dsr_reply_already_seen

Header:

```
Boolean int dsr_reply_already_seen (int source_dsr_address, int destination_dsr_address,
int sequence_number)
```

`int source_dsr_address`: the dsr address of the reply source

`int destination_dsr_address`: the dsr address of the reply destination

`int sequence_number`: the sequence number of the reply

Return: 1 if the reply packet has been already seen

0 otherwise or if it is a gratuitous reply

Description:

This function returns `OPC_FALSE` the first time it is called for a Route Reply packet identified by its source, its destination and its sequence number. These parameters are

stored by the function in the reply_seen table in order to return OPC_TRUE if it is called back with the same parameters.

Note that only the last sequence number received for a specific source is stored in memory. Thus every reply with a sequence number lower than the one in memory is considered “already seen”, since a reply with a lower sequence number must be older than the one stored.

Function dsr_forward_reply

Header:

void dsr_forward_reply (Packet* pk_ptr)

Packet* pk_ptr: a pointer to the reply packet to forward

Description:

This function forwards the Route Reply packet to the next node contained in the packet's route.

Function dsr_insert_route_in_cache

Header:

void dsr_insert_route_in_cache(Packet* pk_ptr)

Packet* pk_ptr: a pointer to the Route Reply packet from which the route is extracted

Description:

This function fills in the Route Cache of a node that is on the path and receives a Route Reply packet. In fact, it extracts all the information contained in the source route of the Route Reply packet that could be useful for itself, and that is confirmed bi-directional. That is, the source route from itself to the final destination but also all the paths from itself to any intermediate node on the way toward the final destination of the source route.

Function dsr_promiscuous_reply

Header:

void dsr_promiscuous_reply (Packet* pk_ptr)

Packet* pk_ptr: a pointer to the reply packet to process

Description:

This function is called when a node receives a Route Reply and it is neither a relay nor the target for this packet. In this case, it uses the packet to process a mechanism avoiding reply storms. Actually, if the current node was planning to reply to the Route Reply corresponding to the same route discovery sequence identified by the triple (source, destination, sequence_number), then it must cancel the transmission of its own Route Reply.

Data Transmission Functions

Function dsr_upper_layer_data_arrival

Header:

void dsr_upper_layer_data_arrival (Packet* data_pk_ptr, int destination_dsr_address)

Packet* data_pk_ptr: a pointer to the upper layer data packet

int destination_dsr_address: the dsr address of the upper layer data destination

Description:

This function is called when the node receives some data from the upper layer to transmit. It creates the DSR data packet in which these data will be transported through the network. If there is no previous data packet to transmit to the same destination, and if the node knows a route to this destination then the packet is sent immediately by calling the dsr_transmit_data (...) function. Otherwise, the packet is stored in a queue buffer via the dsr_insert_buffer (...) function, and if no route discovery mechanism is currently in process a new one is activated by calling the dsr_transmit_request (...) function.

Function dsr_transmit_data

Header:

void dsr_transmit_data(Packet* pk_ptr, int destination_dsr_address)

Packet* pk_ptr: a pointer to the data packet to send

int destination_dsr_address: the dsr address of the data packet destination

Description:

This function fills in and sends the DSR data packet destined to the specified destination.

Function dsr_handle_data

Header:

void dsr_handle_data(Packet* pk_ptr)

Packet* pk_ptr: a pointer to the data packet to process

Description:

This function is used when a node receives a data packet. It first checks that the packet is received for the first time with the dsr_data_already_seen (...) function, which means that it must be processed. Then if the node is the final destination of the packet, the packet is transmitted to the upper layer process. Otherwise, if the node is the relay of the data packet, the data packet is forwarded to the next node contained in the source route by calling the dsr_forward_data (...) function. Finally, if the node is not concerned by the packet, the promiscuous mode is activated by calling the dsr_ckeck_gratuitous_reply (...). If a shorter path is found then a Gratuitous Route Reply packet is sent via the dsr_transmit_gratuitous_reply (...).

Function dsr_schedule_no_ack_event

Header:

void dsr_schedule_no_ack_event (Packet* pk_ptr)

Packet* pk_ptr: the data packet that will be transmitted

Description:

This function schedule a "no ack reception" event associated with a data packet that will be transmitted. It also associates some information in a structure, in order to be able to send an error packet if the link on which the data packet is transmitted is broken. This

error detection is presently done by the 802.11 MAC layer, as well as each acknowledgement message coming directly from this sub-layer. That is why the “no ack reception” timer is reset each time a message (error or acknowledgement) is received from the 802.11 process. Actually, if this timer ends and activates an interruption, that means that the 802.11 process does not work, resulting in the simulation termination.

Function dsr_data_already_seen

Header:

int dsr_data_already_seen(int pk_id)

int pk_id: the packet_id of the data packet

Return: 1 if the data packet has been already seen

0 otherwise

Description:

This function returns OPC_FALSE the first time it is called for a data packet identified by its packet ID. This parameter is stored in the received_packet_id_fifo queue in order to return OPC_TRUE if it is called back with the same parameter.

Note that each time this function is called, it removes every too-old packet ID in order to free the memory used by information that is not useful anymore.

Function dsr_forward_data

Header:

void dsr_forward_data (Packet* pk_ptr)

Packet* pk_ptr: a pointer to the data packet to forward

Description:

This function forwards the data packet to the next node contained in the packet's route.

Route Maintenance Functions

Function dsr_transmit_error

Header:

void dsr_transmit_error(sRoute route, int error_node_dsr_address)

sRoute route: the route to the error packet destination

int error_node_dsr_address: the address of the node with whom the communication link is broken (link from myself to him is broken)

Description:

This function builds and sends a Route Error packet. It is called when a node, which tries to relay a data packet, receives an error message from the 802.11 MAC layer. It uses the path in a reverse way in order to route the Route Error packet from itself to the initiator of the data packet. In addition, the Route Cache of the node is updated through the function dsr_clean_cache (...) in order to remove every route containing the broken link.

Function dsr_handle_error

Header:

void dsr_handle_error (Packet* pk_ptr)

Packet* pk_ptr: a pointer to the error packet to process

Description:

This function is called when a node receives a Route Error packet. Every node receiving an error packet must call the dsr_clean_cache (...) function in order to remove all the routes containing the broken link from its Route Cache. This is the promiscuous mode on the Route Error packet.

If the node is a relay it forwards the packet to the next node in the source route. Otherwise, if the node is the target of the error packet, that is the original source node, then the node must send another Route Request in order to find a new path to the destination by calling the dsr_transmit_request_from_error (...) function.

Function dsr_check_gratuitous_reply

Header:

int dsr_check_gratuitous_reply(Packet* pk_ptr)

Packet* pk_ptr: a pointer to the received data packet

Return: the index of the current node in the data packet path if a gratuitous reply is required

0 otherwise

Description:

This function checks when a data packet is not destined to the current node (neither destination nor next relay) if a shortest path physically exists and thus if a gratuitous reply packet is required. It returns the index of the current node in the data packet path if such a reply is required, OPC_FALSE otherwise.

Function dsr_transmit_gratuitous_reply

Header:

void dsr_transmit_gratuitous_reply (int current_node_index, Packet* pk_ptr)

int current_node_index: the index of the current node in the data packet path

Packet* pk_ptr: a pointer to the data packet that requires a gratuitous reply

Description:

This function builds a gratuitous reply packet by extracting some information from the data packet. Then it sends this packet immediately if the current node is the data packet final destination, or after a short delay depending of the calculated reply route size if it is a relay. This delay mechanism avoids Route Reply storms, and ensures that the only the shortest route will be considered.

Function dsr_clean_cache

Header:

```
void dsr_clean_cache(int first_node_dsr_address,int second_node_dsr_address)
int first_node_dsr_address: the dsr address of the first node identifying the broken link
int second_node_dsr_address: the dsr address of the second node identifying the broken link
```

Description:

This function removes from the Route Cache all the routes containing the link identified by the pair of nodes (first_node_dsr_address, second_node_dsr_address). At this point, it is important to note that our Route Cache is a path cache (two-dimensional table) that lists the path from a given source to a given destination. It is different from the tree structure, also called link cache, which was not the design choice for our OPNET model. However, the updating operation is equivalent to the one described in the third DSR specification, that is, each route containing the hop in error must be truncated at this hop.

Other Functions

Function dsr_in_transmission_range

Header:

```
int dsr_in_transmission_range(Packet* pk_ptr)
Packet* pk_ptr: a pointer to the received packet
Return:1 if the packet is within transmission range
      0 otherwise
```

Description:

This function is called each time a node receives a packet in order to simulate the transmission range of the nodes. The function computes the distance between the transmitter and the receiver of the current packet (by using the hidden TR_Source field contained in each packet), and if this distance is greater than the maximal transmission range the packet is rejected, otherwise it is accepted and processed.

Function dsr_insert_buffer

Header:

```
void dsr_insert_buffer(Packet* pk_ptr, int destination_dsr_address)
Packet* pk_ptr: a pointer to the data packet to store in the buffer
int destination_dsr_address: the dsr address of the data packet destination
```

Description:

When a node wishes to transmit a data packet to a destination for which it does not know the route, it calls this function in order to store the packet in a buffer during the route discovery process. Note that the OPNET node model is inherited from a queue process, thus it allocates one subqueue for each destination (see the OPNET subqueue package).

Function dsr_buffer_empty

Header:

int dsr_buffer_empty(int destination_dsr_address)

int destination_dsr_address: the dsr address of the eventual data packet destination

Return: 1 if the buffer is empty

0 otherwise

Description:

This function checks if the node buffer is empty for the given destination and returns 1 if it is the case.

Function dsr_extract_buffer

Header:

Packet* dsr_extract_buffer(int destination_dsr_address)

int destination_dsr_address: the dsr address of the data packet destination

Return: a pointer to the data packet extract from the buffer

Description:

Once a node has discovered a route to a specific destination, it must send every packet that has been stored in this destination's subqueue during the route discovery process. For that purpose it extracts the data packets one by one from the subqueue using this dsr_extract_buffer (...) function.

Function dsr_no_loop_in_route

Header:

void dsr_no_loop_in_route (sRoute* route)

sRoute* route: a pointer to the route that will be free from all eventual loops

Description:

This function checks and removes every loop in the given route. It is called by a node that has just constructed a route replying to a Route Request by using its route cache, and wants to ensure that its route is loop free.

Function dsr_send_to_mac

Header:

void dsr_send_to_mac(Packet* pk_ptr, int destination_mac_address)

Packet* pk_ptr: a pointer to the packet to send to the MAC LAYER

int destination_mac_address: the MAC address of the packet destination

Description:

This function is called as soon as the dsr process model wishes to send a packet on the physical layer. Thus this function is like is the interface with the 802.11 MAC layer, since it communicates the address and the packet to send to it.

Function dsr_message

Header:

```
void dsr_message (const char* message)
const char* message: the node message to display
```

Description:

This function displays the name of the talking node (thus the name of the current node) following by its message. It is used only in order to inform our DSR model user, what the model is currently doing.

Function dsr_end_simulation

Header:

```
void dsr_end_simulation()
```

Description:

This function is called at the end of the simulation. Its purpose is to free the memory used by the dsr process, to terminate the statistics collection, and to store all these statistics in a Text file.

Variable Descriptions

The Route Cache

Definition:

sRoute* route_cache

where:

sRoute is a structure describing a path with a maximal size equal to 8 nodes (design choice), and defined as follow:

```
typedef struct
```

```
{  
    int route[MAX_SIZE_ROUTE];    // the route = array of integer (dsr addresses)  
    int size_route;                // the size of the route  
} sRoute;
```

```
#define MAX_SIZE_ROUTE 8          // maximum number of nodes in a route
```

Description:

In our OPNET implementation of DSR, the Route Cache of each node is a dynamic table indexed by the destination node DSR address. This table lists the path from the current node to each possible destination, and thus is allocated dynamically with a size equal to the “number of nodes located the network.” Consequently, this table matches with the Path Cache described in the last DSR specification, except that only one route for each destination can be stored. Obviously, as described in the specification, this route is the shortest one known leading to the destination.

This Route Cache variable is different from the tree structure, also called Link Cache, more recommended in the DSR specification, which is easier to understand and to maintain, but more difficult to manipulate.

As mentioned in our overview of this model, we chose this cache strategy for certain reasons. On the one hand, our main goal is to evaluate the basic DSR mechanism’s behavior and performances, and not the influence of its different route cache strategies. Moreover, we believe that the real power and interest of a multiple route cache is located in its ability to be used in order to choose a route considering other metrics than the shortest path.

The Route Request Table (request_seen & request_sent)

Definition:

SRequestSent* request_sent

int** request_seen

where:

sRequestSent is a structure containing every useful information about the sent request, and defined as follow:

typedef struct

```
{
    int sequence_number;           // the sequence number of the request
    double scheduling_time;        // the time when a new request should be sent
    double waiting_time; // the total time we have to wait before scheduling_time
    Evhandle evt;                 // the event associated with the scheduling_time timer
} sRequestSent;
```

Description:

In the DSR specification, a route request table is defined in order to collect information about the Route Requests that have been recently forwarded or originated by a node. The goal is to use this information to avoid a large number of useless Route Requests in the network. We have “replaced” this table by two variables, one for the requests that have been recently originated by the node (request_sent), and one for the requests that have been recently forwarded (request_seen). These two tables are dynamically allocated in order to optimize the memory used according to the network size.

First, the request_sent table is indexed by the DSR address of the Route Request destination node. This table, which is located in each node of the network, contains all the information about the last Route Requests sent by the node to each specific destination. That is, the sequence number of the last Route Request, the maximal time that the node must wait for the Route Reply, and the event associated with this time. Obviously when this timer expires, the node must attempt a new Route discovery process for this destination.

The second table is the request_seen table. It is a two-dimensional table that is indexed by a pair (source, destination) and located in every node. Each of its entries contains the sequence number of the last Route Request that has been forwarded by the node for a given source and a given destination. This second table allows the control of the broadcasting mechanism: each node can forward only one time a given Route Request (identified by its source, its destination, and its sequence number).

The Route Reply Table (reply_seen)

Definition:

int** reply_seen

Description:

This structure is not described in the DSR specification. We implemented it in order to fix the small problem described in the report in the section *Route Cache Strategies – Bi-directional links*. Actually, since it happens that a Route Request and its Route Reply packet do not follow the shortest route, we have implemented a mechanism similar to the Gratuitous Reply on the Route Reply packet. Since in this case we need to process only one time the two received Route Reply packets, we have to memorize the fact that a reply packet was already handled.

In that way, we use a structure similar to the request_seen variable described above, that is, a two-dimensional table indexed by a pair (source, destination) and located in every node. Each of its entries contains the sequence number of the last Route Reply that has been forwarded by the node for a given source and a given destination. Note that this request_seen variable is also dynamically allocated in order to optimize the memory used according to the network size.

The Data Packet Queue (received_packet_id_fifo)

Definition:

SFifo received

where:

sFifo is a First In First Out structure provided by our own fifo external library. The main feature of this fifo or queue is to provide the multiplexing and the multi-type data handling services. Note that we store in this queue only the Ids (integer) of the last received data packets.

Description:

This variable is not described in the DSR specification. We built it as a consequence of our Gratuitous Reply mechanism implementation. Actually, as discussed in our overview, in the latest DSR specification it is written that a node must wait for the arrival of the “normal” (following the “normal” path) data packet, even if a shorter path is detected, in order to forward it. In our implementation, since our goal is to obtain the best performance possible, we chose to forward the first received packet since we are not sure to receive later the “normal” one. Thus we need to memorize the fact that the node has already forwarded the data packet, in order to forward it only one time. That is the purpose of this queue.

Actually, we store in this queue the Ids of every packet that has been received in the previous last five seconds. In that way, we are sure to avoid memory overflow, and to avoid any duplicated data packet in our network.

The Send Buffer

Definition:

See the OPNET subqueue package.

Description:

This buffer, which is located in each node, is used to store the packets that cannot be transmitted because the node does not yet have routes leading to their destinations. This variable is implemented through the OPNET subqueue package. In fact, each node is “inherited” from the OPNET queue object, and therefore can use this package that provides a queue for each destination.

The Acknowledgment Timer Queue (no ack fifo)

Definition:

sFifo no_ack_fifo

where:

sFifo is a First In First Out structure provided by our own fifo external library. The main feature of this fifo or queue is to provide the multiplexing and the multi-type data handling services. In this queue we store some structures containing some useful information concerning the last data packets (no ack received yet) sent by the current node. The following is the definition of this structure:

```
typedef struct
{
    Evhandle evt;           // event indicating that no ack has been received =>
    error (either the MAC layer does not reply, or no explicit dsr ack has been received)
    double schedule;       // time at which this event is scheduled
    sRoute route;         // route used by the data packet
    Packet* upper_layer_data; // upper layer data transmitted in the data packet
    int packet_id;        // packet_id of the data packet
} sNoAck;
```

Description:

This queue is filled in by the dsr_schedule_no_ack_event (...) function when the node sends a packet that requires an acknowledgement, that is, a data packet. Thus a timer is activated, and all information associated with it is stored in a sNoAck structure, which is put in the queue.

Note that in our present model, all acknowledgement and error messages are coming from the 802.11 MAC layer. Thus when an error message is received from the MAC, the information stored in the queue is used to send a Route Error through the network. Moreover, when a data packet is successfully transmitted, its information structure is removed from the queue since it is not useful anymore. Finally, it is important to underline the fact that the timer associated with each sent data packet is implemented in order to detect broken links at the DSR level in the future. However, we already use it at a MAC layer controller, and we stop the simulation when neither an

acknowledgement nor an error message is received by our DSR model, since it means that the MAC layer does not work.

The Scheduled Reply Queue (reply_fifo)

Definition:

sFifo reply_fifo

where:

sFifo is a First In First Out structure provided by our own fifo external library. The main feature of this fifo or queue is to provide the multiplexing and the multi-type data handling services. In this queue we store the following sReply structure containing all useful information about the scheduled reply:

```
typedef struct
{
    int sequence_number; // the sequence number of the reply
    Packet* pk;          // the reply_packet
    Evhandle evt;        // the intrpt event which will "say" when to send the reply
} sReply;
```

Description:

This queue is filled in by the dsr_transmit_reply_from_relay (...) and eventually by the dsr_transmit_gratuitous_reply (...) functions, when a relay node plans to send a Route Reply or a Gratuitous Route Reply. It activates a timer that corresponds to the moment when the node must send this reply packet. Note that this reply can be cancelled when the promiscuous mode detects a “better” reply on the network, in order to avoid reply storms.

Model files

In this section all files included in our DSR OPNET model are listed, and their contents are described briefly.

- dsr_routing_layer: our DSR process model
- dsr_interface.*: the DSR interface process model
- Dsr_Data.pk.m: the DSR packet used to carry upper layer data through the network
- Dsr_Reply.pk.m: the DSR packet used for the Route Reply
- Dsr_Request.pk.m: the DSR packet used for the Route Request
- Dsr_Error.pk.m: the DSR packet used for the Route Error
- Dsr_Upper_Data.pk.m: the upper layer data packet that will be transported by our DSR process model (inside the DSR data packet)
- Dsr_Ack_Ici.ic.m: the Ici used by 802.11 to transmit an acknowledgement message to our Dsr process model
- Dsr_Dest_Ici.ic.m: the Ici used by the DSR interface to transmit the random destination address associated with an upper layer data packet to our DSR process model
- Dsr_Error_Ici.ic.m: the Ici used by 802.11 to transmit an error message to our Dsr process model.
- Dsr_Wlan_Dest_ici: the Ici used by our DSR process model to transmit the next destination (next relay) address associated with the a DSR data packet
- billard_mobility.*: the process model simulating the node mobility (here the billard mobility)
- complex_intrpt.*: a home made library used in our DSR process for its ability to associate any type of data with an OPNET event
- dsr_sink.*: the upper layer that received the successfully transmitted packet from our DSR layer
- dsr_support.*: a home made package used to manage and check the DSR address attribution process
- fifo.*: a home made library used in our DSR process model for its multiplexing and multi-type data handling services.
- dsr_node.nd.m: our DSR node model (including DSR routing, and 802.11 MAC layer)
- nist_dsr_model-16_nodes_network.*: every files used by OPNET to manage a scenario. These files describe a network of 500m*500m, containing 16 of our DSR nodes.
- nist_dsr_model.prj: the files that manages an OPNET project. Thus our project contained the nist_dsr_model-16_nodes_network scenario.
- wlan_mac_dsr_Sept00.*: the 802.11 process model (note that this process model is provided by OPNET, but we made some modification regarding the specificities of our network and routing layer)

- wlan_mac_dsr_interface.*: the 802.11 interface process model (note that this process model is provided by OPNET, but we made some modification regarding the communication with our routing layer)
- wlan_mac.pk.m: the 802.11 data frame packet (provide with the OPNET 802.11 model)
- wlan_control.pk.m: the 802.11 control packet (provide with the OPNET 802.11 model)
- wlan_mac_ind.ic.m: an information Ici used by the 802.11 process model (provide with the OPNET 802.11 model)
- wlan_support.ex.c: an external support for the 802.11 address management (provide with the OPNET 802.11 model)
- wlan_chanmatch.ps.c
- wlan_ecc.ps.m
- wlan_propdel.ps.c
- wlan_rxgroup.ps.c: some C code sources used in the pipeline stage (provide in the 802.11 OPNET model)