DSR in ns-2

[Back to *Network Simulator 2 for Wireless* Home Page]

Source code :

Not all flies in ./dsr/ directory are used by the ns-2. the routing agent is implemented as Agent/DSRAgent.
Thus, the source codes include:

- **dsragent.cc (h)**: DSR agent class. major state machine handling routings. Important variables; net_id, mac_id in ID type. (IP and MAC address), both of them are initialzied by tcl commands to set the initla valie, the commads are "addr" amd "mac_addr"
- **hdr_sr.cc(h)**: define hdr_sr class.
- request_table.cc(h)
- **Path.h(cc):** Path class. First, define struct ID, it has an unsigned long addr a enum of ID_type, and a time stamp t. and then in Path class, ID[] is the key members of the path, and operator [] is defined to return an element of ID array. thus whenever the SRPacket.route[n] will return to the reference of ID[n]. Other member variables include cur_index, len,
- **srpacket.h**: Just define SRPacket class which enclose the hdr_sr as a full packet. The SRPacket construct has two parameters, a normal packet and a SR (Source Route) Header (a path variable). The constructor of "path" class makes a path from the bits of an NS source route header. And the other two variables of the SR packet are "dst" and "src" IP addresses.

Data Structure for Route

It is found that srh->addr and p.route are two different structures. srh() is always along with the packet. however, when DSR agent received a packet, it will

```
SRPacket p(packet, srh);
```

This generate a "p" which is frequently used by all other functions. remember p does not go along with the packet leaving dsr agent. Before the packet was sent out of the agent, another statement will be used to update "SRH" in *sendOutPacketwithRoute*

```
p.route.fillSR(srh);
```

Also, the tap() entry is also generate a p for its use. however, another entry point of agent "xmitFailed" use srh() directly.

Special tcl interface.

Unless other routing protocol, the ns-2.1b9a,has a special node type named as "SRNodeNew". From those routines in the ns-lib.tcl. We can see that speical. Also, there is a tcl file in mobiloty/dsr.tcl is also related.

```
Simulator instproc create-node-instance args {
        $self instvar routingAgent_
        # DSR is a special case
        if {$routingAgent_ == "DSR"} {
                set nodeclass [$self set-dsr-nodetype]
        } else {
                set nodeclass Node/MobileNode
        }
        return [eval new $nodeclass $args]
}
```

```
Simulator instproc set-dsr-nodetype {} {
        $self instvar wiredRouting_
        set nodetype SRNodeNew
        # MIP mobilenode
        if [Simulator set mobile_ip_] {
                set nodetype SRNodeNew/MIPMH
        }
        # basestation dsr node
        if { [info exists wiredRouting_] && $wiredRouting_ == "ON"} {
                set nodetype Node/MobileNode/BaseStationNode
        }
        return $nodetype
}
```

DSR Signaling Packets in Brief:

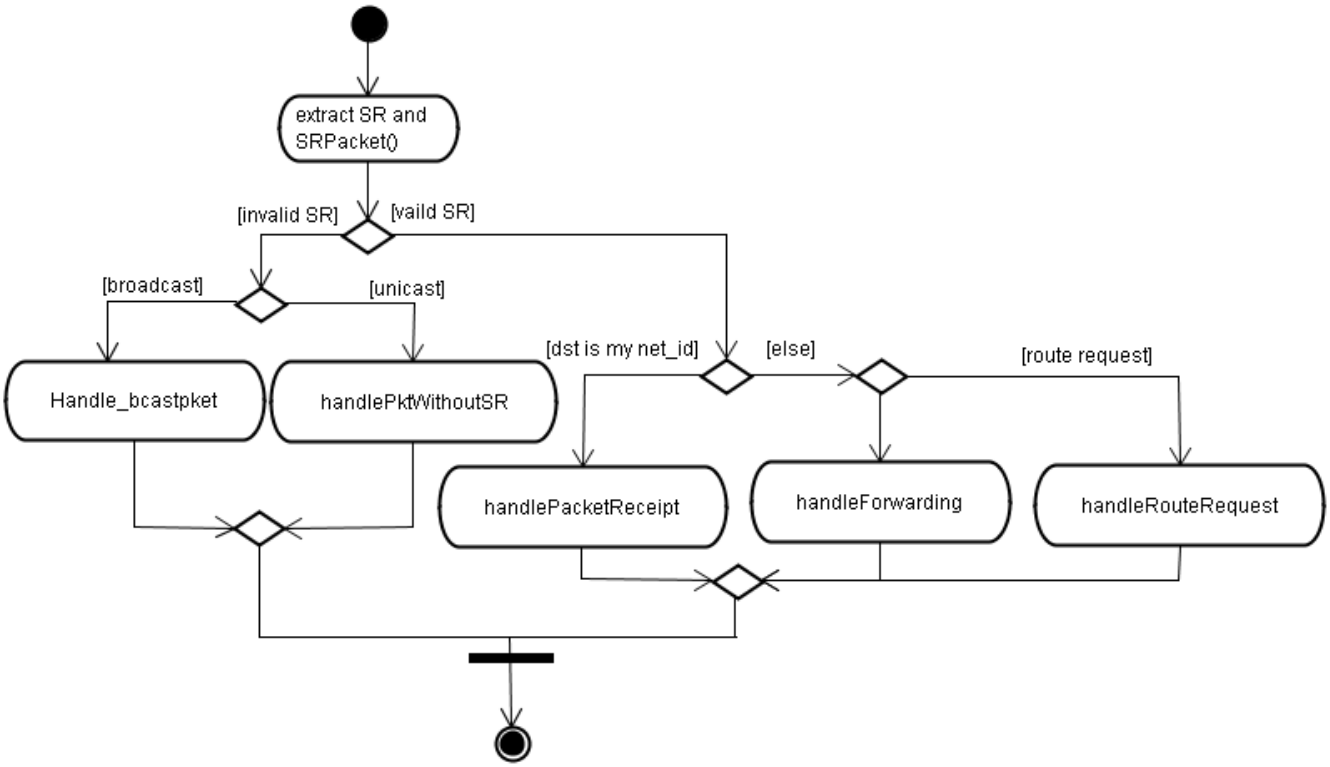
- route-request. the packet itself is a layer 3 packet with a unique destination address, but MAC_Broadcast (labeled in common header's next_hop())
- route-reply: unicasting in both layer 2 and 3.
- route-error. unicasting in both layer 2 and 3. Generated when tx_failure in lower layer.

Entry Points for DSR agent:

1. first, as normal, the recv() function which means a packet with a address destine to this node or from upper-target.
2. xmitFailed(). This is the callback function when a MAC transmission failes. Based on this chance, route-error message generated
3. tap(). This is a hidden entry when you turn promiscuous on. snooping the route and shorten the path.

Basic functions:

- *recv()*, the entry for a receving packet. depends on the ip address of the packet and the SR (Source Route) header, call different functions to handle it. like the diagram below:



The route-reply, route-request messages need special handling routines written in the *handlePacketReceipt()* funciton. Note that if the RRequest message reached the destination, the receiptent should send a RReply message, this is done by a function named *returnSrcRouteToRequestor(p)* which is called in *handlePacketReceipt()*. Otherwise, if the route-request and route-err

- or are not destined to us, for route-request, the function *handleRouteRequest()* is called
- *handlePacketReceipt()*. A signaling reached its destination. There are two case:
  - if it is a route-request and "not processed", sends back a route-reply, pkt "p" is forged in this function and *returnSrcRouteToRequestor(p)*
  - if it is a route-reply, call functiuon *acceptRouteReply(p);*
- *handleRouteRequest()*: From the version 2.27, we see some unused codes but probably under develop for future versions. It includes more close mac-routing cooperation, such as neighbor identity (is_neighnor()), and channel status (air_time_free()).
  Basically, this function has three branches:
  1. already processed, checked by function *ignoreRouteRequestp(p)*.
  2. has a cached route, done by *replyFromRouteCache(p)*, and "cached route" is enabled by the flag *dsragent_reply_from_cache_on_propagating*.
  3. append myself in route with *p.route.appendToPath(net_id);* and *sendOutPacketWithRoute(p, false);*
- *handleForwarding*. Forward packet on to next host in source route and snooping as appropriate. So, a route-reply message is not treat as exceptional. It is a normal packet with sr header and be snooped by this node. The snooping is enabled by the flag "*dsragent_snoop_source_routes*" . handleForwarding is dcallin "handleDefaultforwarding" for doing some simple operations for DSR rules. At last, the packet will be sent by *sendOutPacketWithRoute*
- *sendOutPacketWithRoute*: The function is used as for send packets, Take packet and send it out, packet must a have a route in it. return value is not very meaningful. if fresh is true then reset the path before using it, if fresh is false then our caller wants us use a path with the index set as it currently is. Basically,
  - cmn header's failure callback functon and data are set
  - cmn header 's next hop is set to addr of next-hop in dsr header. address type are also set.
  - move the pointer in SR header to the next (increase 1).

  Actually, the third operation is not valid in real DSR implementations. To undo this effect in error-handing, we'd better find current Ip address first and locate the position of this address in SR header. The other two operations are also invalid, because there are no common headers in a real

packet. For the "next_hop()" in common header, it is used by DSR only for those packets without valid SR header, refer to *recv()*.

- *returnSrcRouteToRequestor()*; this function
- *xmitFailed()*: when the common header->xmit_failure_ point to a callback function, thus, when the packet cannot be delivered, the callback function is used, and finally will generate a route-error messages. There is always a pointer in the SR header curr() . (Refer to **manet-ietf-dsr draft**, there is no such a pointer in DSR Source Route Option, but has a "*segments left*" field to indicate how many nodes still to visit to reach the destination.). Thus, an innovation needs to be done to re-interprete the "srh->cur_addr()" as a index number of the position in the path where fail happens. So, when not all nodes along the path handle the SR header, we need find the ip address of the node from srh. and set that index as cur_addr();
- *processBrokenRouteError(p)*; This should be another branch under the main recv() entry. It gives what to do when a Route Error message is receoived or heard (snooped). Snoop means the message was sent to another node but it passes myself, so i heard it.
- *tap(const Packet *packet)*: This is another entry point for DSR. When dsragent_use_tap flag is true. the mac is working in promiscuous mode and all overhearing packets will be processed if there is a SR header in it.

Other:
The longest route we can handle is defined in : define MAX_SR_LEN 16 // longest source route we can handle

Possible Reason for xmit_failure below IP layer:

- arp failure
- interface queue is full
- mac transmission failure (exceed the retry-limit)

DSR scheme options:
In the beginning of dsragent.cc, it define many bool selectors of some options like:

```
/*************** selectors ******************/
bool dsragent_snoop_forwarded_errors = true;
// give errors we forward to our cache?
bool dsragent_snoop_source_routes = true;
// should we snoop on any source routes we see?
bool dsragent_reply_only_to_first_rtreq = false;
// should we only respond to the first route request we receive from a host?
bool dsragent_propagate_last_error = true;
// should we take the data from the last route error msg sent to us
// and propagate it around on the next propagating route request we do?
// this is aka grat route error propagation
bool dsragent_send_grat_replies = true;
// should we send gratuitous replies to effect route shortening?
bool dsragent_salvage_with_cache = true;
// should we consult our cache for a route if we get a xmitfailure
// and salvage the packet using the route if possible
bool dsragent_use_tap = true;
// should we listen to a promiscuous tap?
bool dsragent_reply_from_cache_on_propagating = true;
// should we consult the route cache before propagating rt req's and
// answer if possible?
bool dsragent_ring_zero_search = true;
// should we send a non-propagating route request as the first action
// in each route discovery action?

// NOTE: to completely turn off replying from cache, you should
// set both dsragent_ring_zero_search and
// dsragent_reply_from_cache_on_propagating to false

bool dsragent_dont_salvage_bad_replies = true;
```

```
// if we have an xmit failure on a packet, and the packet contains a
// route reply, should we scan the reply to see if contains the dead link?
// if it does, we won't salvage the packet unless there's something aside
// from a reply in it (in which case we salvage, but cut out the rt reply)
bool dsragent_require_bi_routes = true;
// do we need to have bidirectional source routes?
// [XXX this flag doesn't control all the behaviors and code that assume
// bidirectional links -dam 5/14/98]

#if 0
bool lsnode_holdoff_rt_reply = true;
// if we have a cached route to reply to route_request with, should we
// hold off and not send it for a while?
bool lsnode_require_use = true;
// do we require ourselves to hear a route requestor use a route
// before we withold our route, or is merely hearing another (better)
// route reply enough?
#endif
```

About Flow State:

It is also desirable to disable the flow state stuff. it make the dsr code messy. flow state is not an orginal idea.

```
static const bool dsragent_enable_flowstate = false;
static const bool dsragent_prefer_default_flow = false;
```

About Packet Salvage

it's unknown how to complete disable the packet retransmission in layer 3. Even you change three expressions in dsragent.cc.

1. salvage_with_cache = false (from true)
2. salvage_max_request =0 (from 1)
3. salvage_times = 0 (from 15)

The trace file still show that the routing agent send a undeliverable packet again. See xmitFail() function. I guess, it is necessary to disable "GOD" also.

Reference:
Bryan's NS-2 DSR FAQ