Specification for Multicast Optimizations in B.A.T.M.A.N. advanced

Simon Wunderlich, saxnet gmbh
Linus Lüssing, saxnet gmbh

December 7, 2010

Simon Wunderlich, saxnet gmbh
Linus Lüssing, saxnet gmbh

Abstract

This document provides a detailed software development specification of the multicast optimizations for B.A.T.M.A.N. advanced, the implementation of the B.A.T.M.A.N. routing protocol in kernel space for layer 2 routing. This approach is focused for applications with symmetric multicast relationships, but can easily be extended for more general use cases. The main focus of this document is to provide both an a priori guideline for a clean and structured integration as well as an a posteriori documentation for easing prompt reviewing and long-term maintenance.

Contents

1 Terminology **3**

2 Concept **4**

2.1 Algorithm 4

2.1.1 Multicast Announcements 4

2.1.2 Distribution Infrastructure 4

2.1.3 Multicast Packet Forwarding 6

2.2 Limitations 7

2.2.1 Symmetric Multicast Group Memberships 7

2.2.2 Multicast Optimizations for B.A.T.M.A.N. Nodes Only 8

2.3 Parameters 8

3 Interfaces **9**

3.1 User Interfaces 9

3.2 Program Interfaces 9

3.2.1 Import Interfaces 10

3.2.2 Export Interfaces 10

4 Integration **12**

4.1 MCA attaching 12

4.2 Multicast Tracker Packet Scheduling 12

4.3 Outgoing multicast data packet on batman interface 12

4.4 Incoming multicast data or multicast tracker packet on batman port 13

5 Implementation **14**

5.1 Methods 14

5.2 Packet Types 15

5.3 Data Structures 16

1 Terminology

soft interface The virtual tunneling interface of B.A.T.M.A.N. advanced (e.g. bat0)

hard interface An interface utilised by B.A.T.M.A.N. advanced for transportation for the mesh network layer.

node A host running B.A.T.M.A.N. advanced

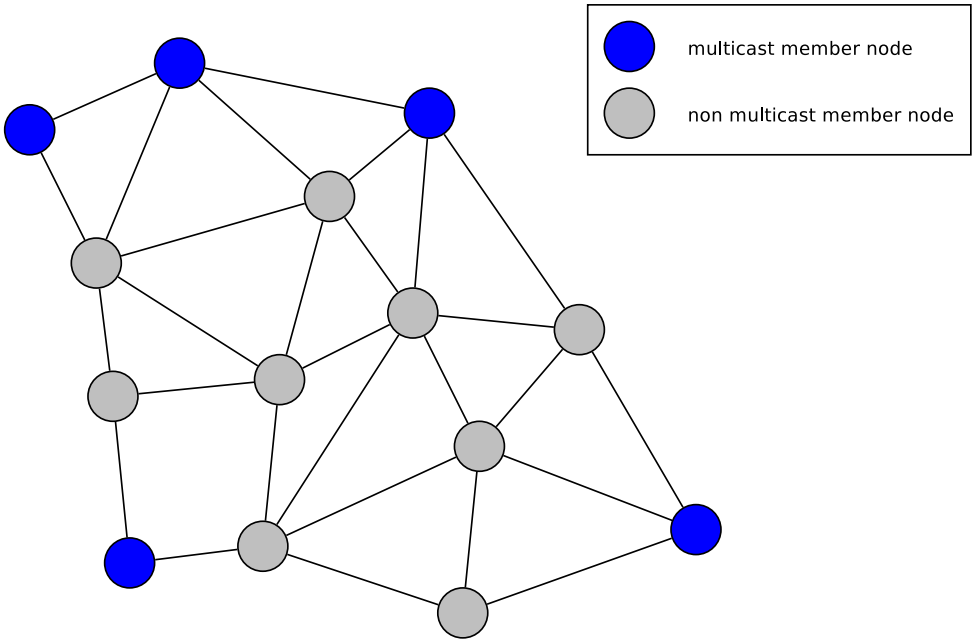
multicast group B.A.T.M.A.N. advanced will not distinguish between different multicast group IDs / IP protocols on layer 3. Instead, the term multicast group will be used analogously for a multicast mac address.

2 Concept

2.1 Algorithm

The basic ideas of the algorithm are described in the following section.

2.1.1 Multicast Announcements



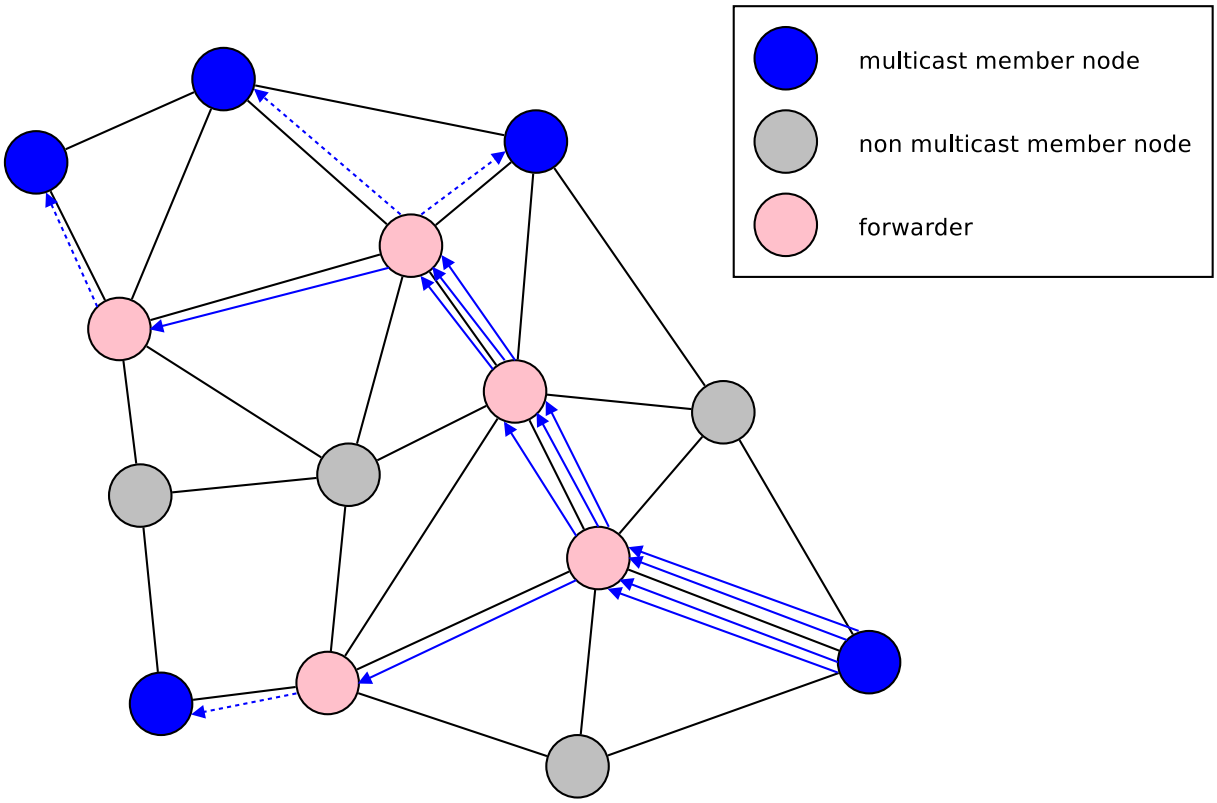
The nodes sense the multicast groups which they are part of, e.g. by inspecting IGMP packets or monitor kernel and routing settings. IGMP (Internet Group Management Protocol) messages are sent to announce multicast memberships when a connection to a multicast network is established. The group information will be attached to B.A.T.M.A.N.'s routing control packets (originator message, OGM) as a *multicast announcement*, MCA for short, and will then be flooded periodically through the mesh network, too. Every node will then have a complete list of other multicast members in any multicast group.

An example mesh network with one group is illustrated in the figure above. The blue nodes are members of the multicast group we will analyze in the example. With this step, every node knows which nodes are present in the mesh, how to reach them and which nodes are in the same multicast group.

2.1.2 Distribution Infrastructure

After receiving the membership announcements, every node knows which other nodes are part of the same multicast group. The distribution infrastructure is built up by periodically sending individual "tracker" packets to every other group member. When a node on the way receives a tracker packet, it knows that it has to work as a forwarder for this sender and this multicast group which is written in the tracker packet. The tracker packets are sent via unicast, and will therefore track the "best" path as found by the routing protocol from the sender to the receivers.

The algorithm is illustrated for the tracker tree of one multicast member in the figure below. Details of the algorithm are described below. It is assumed here that a multicast sender is also a multicast receiver (see limitations in section 2.2 below). Every multicast member will send the tracker packets to every other multicast member.



A tracker packet contains the following information:

- Originator: the original sender of the tracker packet
- Destination: the multicast member where the tracker packet should be sent to
- Multicast Group: the multicast group for which the path is built up
- other information: TTL, B.A.T.M.A.N. advanced packet type and protocol version

When a tracker packet is received by another node, it creates an entry in an internal multicast table with the following information:

- the multicast group from the tracker packet
- the originator from the tracker packet
- the next hop where it would forward the tracker packet to.
- a timestamp to allow removal of the entry after a certain timeout to avoid that the node is still used as a forwarder when the topology changes.

After adding the table entry, the tracker packet is scheduled for sending to the next hop towards the destination. A tracker packet might be scheduled for sending by the originator or by a

forwarder. Unlike "normal" unicast packets, the tracker packet is only sent if the next hop is not the destination - it is not necessary to send the tracker packet to the final destination because the destination should not further rebroadcast the tracker packet as it is the last node in the chain.

In the figure above we can see the unicast packets sent to other group members. Solid lines show transmissions which are done, and the dotted lines are transmission which are skipped according to the optimization described above.

Multicast tracker packets are further aggregated as one bundled packet with all according multicast and destination addresses on the initial sender side and scheduled periodically to reduce overhead. They are then splitted on forwarding nodes if the destination paths branch. Additionally, multicast tracker packets will not be sent to the final destinations, as only the forwarding nodes in between need to be notified.

2.1.3 Multicast Packet Forwarding

Based on the multicast table built up by the tracker packets, the multicast packets can be sent over this infrastructure. In B.A.T.M.A.N. advanced, the multicast packets are encapsulated in a special packet type which contains the originator (original sender) and a sequence number for duplicate checking.

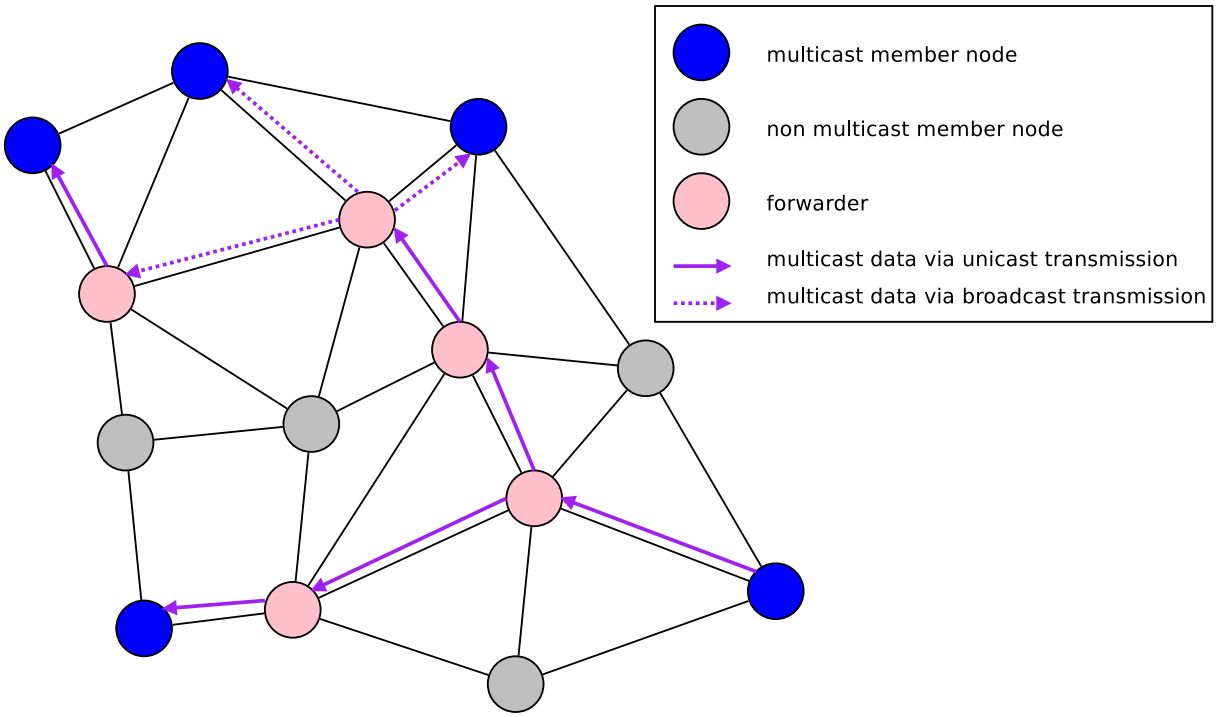
When a multicast packet is received, the following algorithm is followed:

1. if the multicast packet is a duplicate, drop it, otherwise proceed to step 2.
2. if the node is a member of this multicast group, pass it up to the layer above. Proceed to step 3 in any case.
3. if the node is a forwarder for this multicast group and this originator (i.e. entries exist in the multicast table), schedule the multicast packet for sending.

A multicast packet can be scheduled for sending by the Originator or a forwarder. The multicast packet is sent with the following algorithm

1. Compile a list of next hops from the multicast table for this multicast group and originator.
2. If the number of next hops are equal or below the *mcast_fanout* (see section 2.3), send the multicast packet individually via unicast to the next hops. Otherwise use broadcast and send the multicast packet *bcast_num* times.

The *mcast_fanout* is the threshold until which number of next hops individual unicast transmissions should be used. Unicast transmission is expected to be more efficient even if multiple next hops have to be served individually because of higher bitrates and ARQ usage. Packets are only sent once in unicast mode because the WiFi ARQ mechanism will resend the packets if required. The *bcast_num* defines the number of broadcasts which should be used to transfer packet with a good reliability. From experience in B.A.T.M.A.N. advanced we would expect 3 times, or even less, to be a good choice.



The algorithm is illustrated in the figure above. Multicast data is only sent along the forwarders, and WiFi unicast transmission is used to forward the data if only one or 2 next hops have to be served, as indicated with the solid lines. In one case of the example there are 3 next hops, and with an assumed *mcast_fanout* of 2 we will decide to broadcast the packet this time as indicated by the dotted lines.

2.2 Limitations

Note, that due to the otherwise greatly increased complexity the following limitations exist (which could be optimized further in the future, but are not part of this specification):

2.2.1 Symmetric Multicast Group Memberships

A pure IP multicast sender is not actively joining a multicast group and therefore would only allow on-demand path marking when B.A.T.M.A.N. actually notices multicast data packets destined for a host in the mesh network. However, reactive sending of tracker packets has more complexity and needs to be done and balanced with care to not lose any important multicast data on the one hand and to not introduce unnecessary and uneffective overhead on the other.

To avoid adding too much complexity in the beginning and to ease first practical evaluations, it is assumed that every multicast sender will also be a multicast receiver of the same group. A hosts own multicast receiver status can easily be fetched from the kernel. Multicast tracker packets can then proactively be sent between any pair of members of the same multicast group. If a multicast sender is not a multicast receiver in the same group, then flooding is being used as a fallback mechanism instead of dropping such packets (otherwise e.g. IPv6 would be broken).

2.2.2 Multicast Optimizations for B.A.T.M.A.N. Nodes Only

Due to only fetching multicast membership information from the local kernel and acting reactively, we are not able to sense multicast senders/members which are brigded into the mesh network before receiving a multicast data packet. Such packets are then only optimized if the responsible B.A.T.M.A.N. node is a multicast member of the same group.

Unfortunately the Linux kernel (up to 2.6.37) also has no internal, generic interface for sensing bridged in multicast members (e.g. via IGMP or MLD snooping). So far, either a multicast userspace daemon like mrd6 would need to be started to fill the kernels link local multicast caches, the snooping support of the Linux kernel bridge code would need to be exported or a third way would need to be found. Since all ideas so far have their drawbacks and need further discussion, this functionality therefore is a task for later development as well.

2.3 Parameters

Shows the list of parameters for tweaking the multicast behaviour. They can also be changed during runtime.

mcast_tracker_interval This interval is responsible for how frequent multicast path selection packets from a multicast sender to a multicast receiver will be send. Interval is in milliseconds or auto (equaling 1/2 OGM interval).

mcast_tracker_timeout This interval is responsible for how fast a node previously selected for forwarding for a specific multicast group and originator shall discard its forwarding status again if it is receiving no more multicast tracker packets. Interval is in milliseconds or auto (equaling 5x mcast_tracker_interval).

mcast_fanout If the number of next hop nodes for a specific multicast data packet is less or equal to *mcast_fanout*, then the data packet will be copied and send via unicast to the according neighbors each.

bcast_num Sets the number of (re)broadcasts of broadcast data packets. Nodes with weak links should choose higher values to improve the probability of arrival of broadcast packets; nodes with good links and/or with many neighbor nodes should choose a lower value to reduce the medium usage a lot. Note, that this parameter also affects the stability of multicast data forwarding as depending on the *mcast_fanout* (section 2.3) multicast data packets can be forwarded via broadcast too.

3 Interfaces

3.1 User Interfaces

This section lists all options (mostly from section 2.3) that can be changed during runtime from userspace and where to find them in the sysfs.

/sys/class/net/<*meshif*>/mesh/mcast_tracker_interval
See mcast_tracker_interval in section 2.3 for details.
Unit: in milliseconds or 'auto'
Default: auto

/sys/class/net/<*meshif*>/mesh/mcast_tracker_timeout
See mcast_purge_timeout in section 2.3 for details.
Unit: in milliseconds or 'auto'
Default: auto

/sys/class/net/<*meshif*>/mesh/mcast_fanout
See mcast_faunout in section 2.3 for details.
Unit: count
Default: 2

/sys/class/net/<*meshif*>/mesh/bcast_num
see bcast_num in section 2.3 for details.
Unit: count
Default: 3

/sys/class/net/<*meshif*>/mesh/mcast_mode Sets the multicast optimization mode. classic_flooding is the most simple and clear mode which has been the default so far. 'proact_tracking' is an optimized mode, which will build optimized multicast paths between all members of a multicast group without distinguishing between multicast senders and receivers (therefore each sender-only node nevertheless has to join the group as a receiver to be tracked by B.A.T.M.A.N.).

Values: classic_flooding, proact_tracking
Default: classic_flooding

3.2 Program Interfaces

This section describes the methods and data structures the current B.A.T.M.A.N. advanced kernel module needs to access and use to achieve multicast functionality.

3.2.1 Import Interfaces

Lists structures and methods the kernel provides and which need to be imported and used by the multicast methods to support optimised multicast functionality.

Multicast Member Detection The kernel itself stores and offers a list of multicast mac addresses registered to an interface, both via userspace (/proc/net/dev_mcast) but also from kernel space (directly from a net_device - was struct dev_addr_list *mc_list in $< 2.6.27$, is struct netdev_hw_addr_list mc since 2.6.27). We are going to use the kernel variant, as it is the faster approach without having to parse any data. Note that this approach also needs proper locking before traversing its list to avoid race conditions.

This information just covers local multicast groups, it is not possible for B.A.T.M.A.N. to detect multicast members being bridged into the mesh, as there is no generic interface for IGMP/MLD snooping in the kernel. However, the local detection is way simpler than implementing snooping support in B.A.T.M.A.N. advanced and is independent from the network protocol - so even non-IP packets with multicast mac addresses can be optimized here.

*Implemented by: struct dev_addr_list *mc_list / struct netdev_hw_addr_list mc (linux/netdevice.h, struct net_device)*

3.2.2 Export Interfaces

Lists new functionality that needs to be implemented and exported to the current kernel module to provide optimised multicast forwarding.

Announcing Own Multicast Groups A nodes own multicast groups need to be announced throughout the whole mesh network so that a multicast sender is aware of its receivers. The so called MCAs (Multicast Announcements) need to be appended to an OGM in a similar way as it is currently done with the HNAs (Host Network Announcements). It utilises the information provided by the import interface for own group membership detection (see above).

Implemented by: static void add_own_MCA() (send.c)

Maintaining Database of Global MCAs The received MCAs need to be stored so that scheduler for the tracker packets has all the information for generating such packets later. The database needs to be updated with every received OGM.

Implemented by: static void update_MCA() (routing.c)

Generating Multicast Tracker Packets Periodically after every tracker interval, a node needs to check if one or more of its own multicast groups matches one of the groups of another node. If so, a combined multicast tracker packet with all the according multicast groups and their receivers needs to be build.

Implemented by: static void mcast_tracker_timer(), mcast_proact_tracker_prepare() (multicast.c)

Sending and Forwarding Multicast Tracker Packets Both, a newly received as well as a just locally generated tracker packet need to be checked if they must be forwarded or not. If there are even multiple possible next hops to forward such a packet to, a tracker packet further needs to be split accordingly so that a node only receives tracker packets with destinations attached which it will be responsible for forwarding multicast data to. This means, that also a next hop which is already a final multicast data receiver will not need to receive a multiast tracker packet.

Implemented by: void route_mcast_tracker_packet() (multicast.c)

Receiving and Parsing Multicast Tracker Packets B.A.T.M.A.N. advanced needs to recognise a multicast tracker packet and update its multicast forwarding table. A node receiving a tracker packet will is marked for forwarding for the according multicast group(s) and originator.

Implemented by: recv_mcast_tracker_packet(), update_mcast_forw_table() (routing.c, multicast.c)

Receiving Multicast Data Packets A node needs to recognise a batman multicast data packet and check whether it is part of the according group (it might have received multicast data packets, although this node is not a multicast receiver for this group due to broadcasts). If it is a receiver, it needs to make a copy without the batman header and forward it to its soft interface (e.g. bat0).

Implemented by: int recv_mcast_tracker_packet() (routing.c)

Preparing a New Multicast Data Packet for Sending When B.A.T.M.A.N. advanced receives a multicast data packet on its soft interface (e.g. bat0), it firstly needs to recognise this packet and encapsulate it in a batman multicast header. If a multicast sender is not a multicast receiver of the same group then as a fallback mechanism the packet gets encapsulated in a batman broadcast header instead and is flooded through the mesh.

Implemented by: int mcast_send_skb(), mcast_may_optimize() (multicast.c)

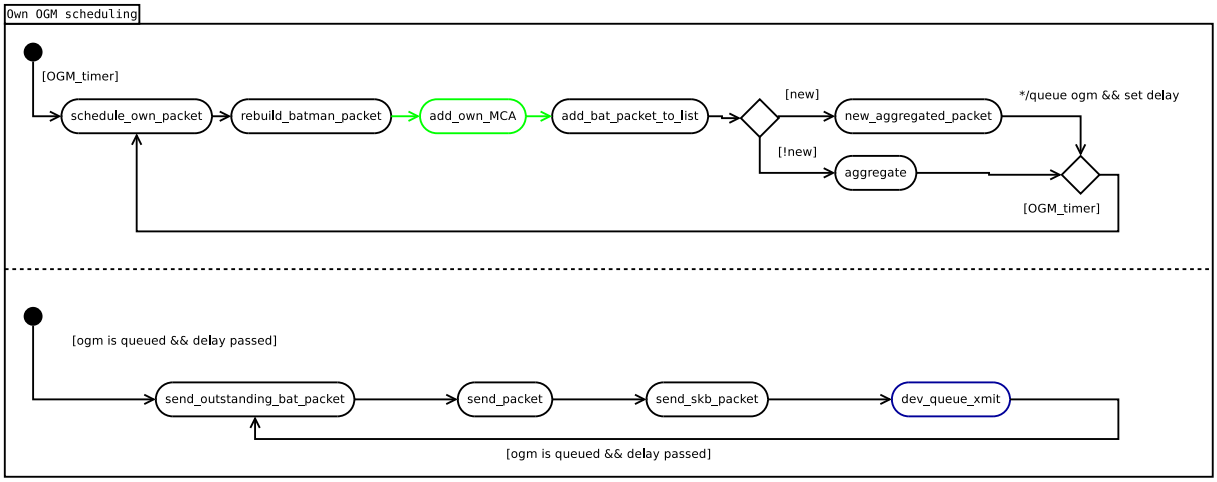
Sending and Forwarding Multicast Data Packets If a node receives or sends a multicast data packet itself, it also needs to check its multicast forwarding table: In case of a matching entry for the according multicast group and originator, it has to forward the multicast data packet to the according neighbor(s). Furthermore, if there are more than mcast_fanout neighbors on a single interface that need to receive this packet according to the forwarding table, then the forwarding shall be done via broadcast. Otherwise, each such neighbor on this interface will get a seperate unicast packet.

Implemented by: void route_mcast_packet() (multicast.c)

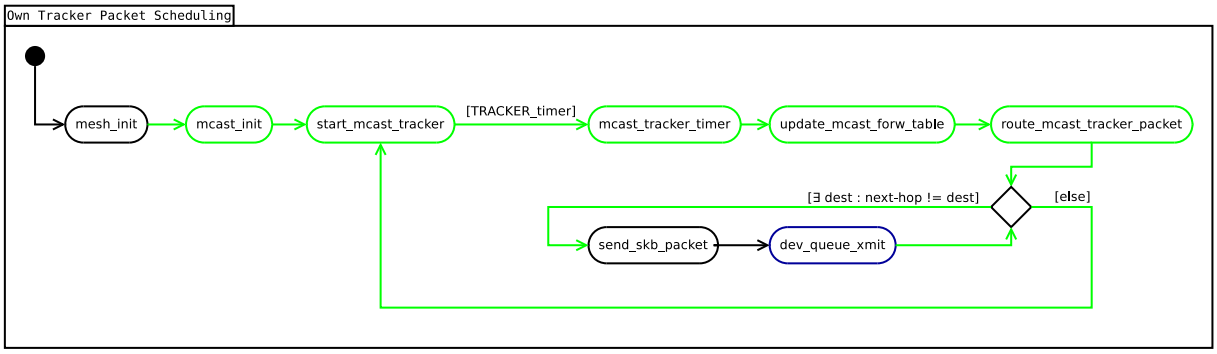
4 Integration

Black: already implemented in B.A.T.M.A.N. advanced
Blue: external methods of Linux kernel
Green: new methods for multicast optimizations in B.A.T.M.A.N. advanced

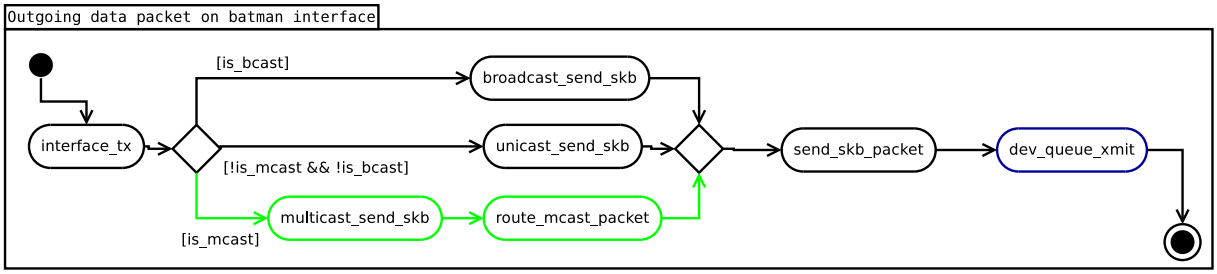
4.1 MCA attaching



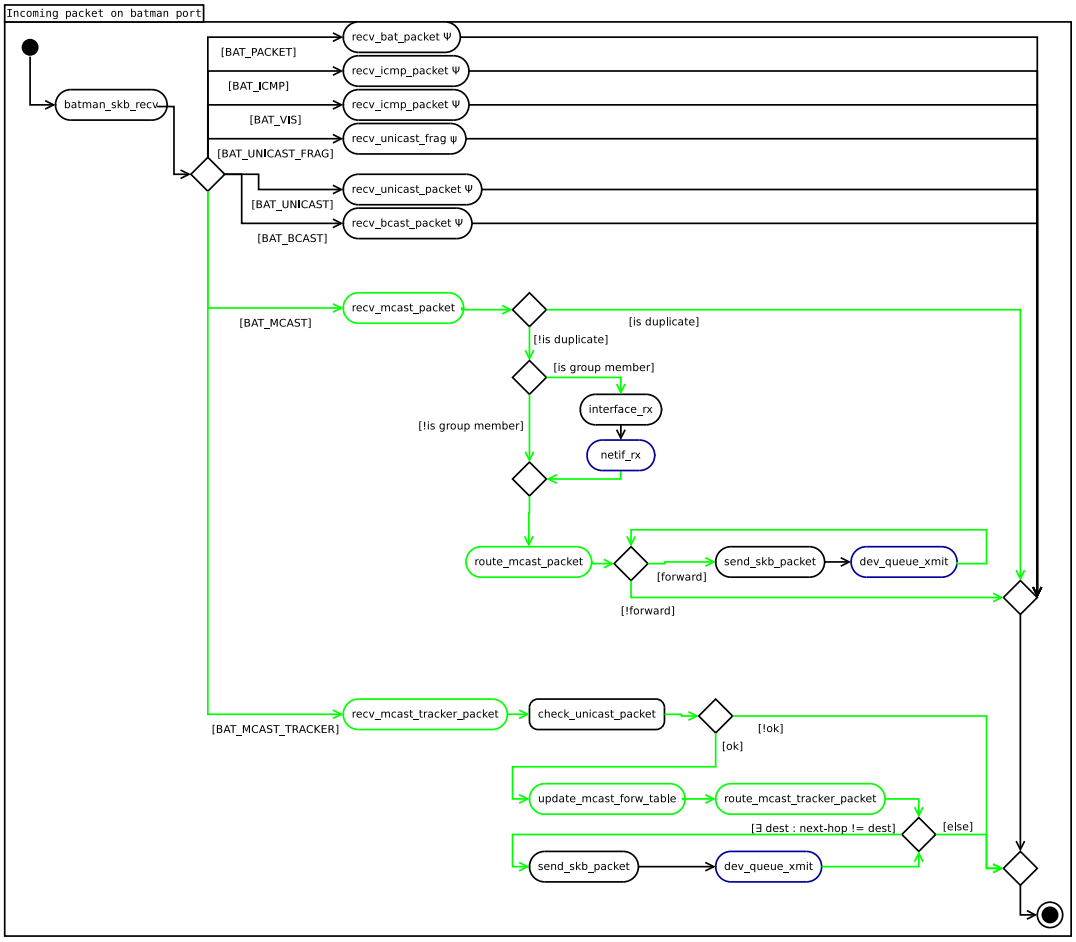
4.2 Multicast Tracker Packet Scheduling



4.3 Outgoing multicast data packet on batman interface



4.4 Incoming multicast data or multicast tracker packet on batman port



5 Implementation

5.1 Methods

This section lists the most important methods for the multicast functionality and describes their contract.

int mcast_send_skb() (multicast.h) If interface_tx() notices a multicast data packet for sending on a batman interface, multicast_send_skb() will be invoked. It is the interface for getting a common multicast data packet onto the mesh network layer. It has to:

- Encapsulate the multicast data.
- Call route_mcast_packet().

int recv_mcast_packet() (multicast.h) Processes an incoming multicast data packet. The following checks are done by this method:

- Checks, if the received packet is a duplicate or if the ethernet frame is garbage. If yes, then drop and return NET_RX_DROP, else procede.
- Call route_mcast_packet().
- Checks, if this node is a member of the same multicast group. If yes, hand a decapsulated copy to the according batman interface via interface_rx().
- Return NET_RX_SUCCESS.

static int route_mcast_packet() (multicast.h) Decide, if this node is a forwarding node for this multicast group and schedule transmission. The following steps are done by this method:

- Checks, if this node has previously been marked for forwarding for this multicast group. If not, skip the following steps.
- Check the number of (forwarding) receiver nodes on each batman port.
  - If 0, do not schedule on this interface.
  - If $\leq$ *mcast_fanout* (2.3) send a mcast_packet with unicast destination to each (forwarding) receiver node on this interface; call send_skb_packet() each.
  - If $>$ *mcast_fanout* (2.3) send a mcast_packet with broadcast destination on this interface; call send_skb_packet().

int recv_mcast_tracker_packet() (multicast.h) Checks and removes the ethernet frame layer of a multicast tracker packet and hands the pure multicast tracker packet over to the next processing function.

- Check if the ethernet header is valid for unicast transport (call check_unicast_packet for that). If not, skip the following steps and return NET_RX_DROP.
- Remove ethernet frame.
- Call route_mcast_tracker_packet() on the multicast tracker packet (without ethernet frame).
- Consume skb.
- Return NET_RX_SUCCESS.

static route_mcast_tracker_packet() (multicast.h)

- Collect a table data structure from the tracker packet for memorizing which data packets to forward later.
- Call update_mcast_forw_table() for synchronising the new table onto the old one.
- Decides to which next hop nodes to forward the/a tracker packet to.
- Splits tracker packet into one for each next hop node accordingly.
- Sends the splitted packets to the according next hop nodes.

static update_mcast_forw_table() (multicast.h) Updates the old, memorized multicast forwarding database table with the new, delivered one.

- If there is no entry for this multicast group and originator, then create one. If there is one, then reset its timestamp and next hop nodes.

static void add_own_MCA() (send.c) This method appends multicast addresses of the local node, the addresses it wants to receive multicast data for, to its primary interface's packet buffer. The number of MCAs and a batman_packet pointer to the beginning of the buffer get passed to this method. A packet buffer of a size fitting the number of MCAs needs to be allocated previously.

- Fetch currently registered multicast mac addresses from a batman interface.
- Attach these mac addresses to the current OGM.

5.2 Packet Types

BAT_MCAST So far, multicast data packets were encapsulated into packet with the type BAT_BCAST and simply flooded through the network. Multicast data packets which shall take optimized paths are now instead encapsulated into a packet with the type BAT_MCAST to allow forwarding nodes to treat such a packet special, according to the algorithm specified in this document.

BAT_MCAST_TRACKER Multicast tracker packets for marking the optimized multicast paths have the new packet type BAT_MCAST_TRACKER in their B.A.T.M.A.N. packet header. This allows a node to not only forward single unicast packets, but to send unicast packets with initially aggregated destinations as long as the next hops for a destination do not differ. It also allows a node to update their own forwarding database when noticing such a packet type.

5.3 Data Structures

Multicast Data Packet For forwarding multicast data on an optimized path, the same packet structure as for broadcasting can be used, therefore "struct mcast_packet" will have the same fields as "struct bcast_packet".

This packet structure is usable for forwarding both via unicast and broadcast. A single packet structure suitable for both broad- and unicast forwarding has been chosen to allow fast switching between the two forwarding methods for the data packet without much conversion effort. Only the ethernet header's destination address has to be modified.

```
struct mcast_packet {
        uint8_t packet_type; /* BAT_MCAST */
        uint8_t version;     /* batman version field */
        uint8_t orig[6];
        uint8_t ttl;
        uint32_t seqno;
} __attribute__((packed));
```

Multicast Tracker Packet Small unicast packets from a multicast sender to a multicast receiver are responsible for marking the path of future multicast data packets. Multicast receivers are initially being aggregated on the same mcast_tracker_packet as long as they share the same next hop.

A multicast tracker packet has the following structure:
packet = (struct mcast_tracker_packet + num_mcast_entries * [mcast_entry + num_dest * dest[6]])

```
/* marks the path for multicast streams */
struct mcast_tracker_packet {
        uint8_t  packet_type;  /* BAT_MCAST_TRACKER */
        uint8_t  version;      /* batman version field */
        uint8_t  orig[6];
        uint8_t  ttl;
        uint8_t  num_mcast_entries;
        uint8_t  align[2];
} __attribute__((packed));


struct mcast_entry {
        uint8_t  mcast_addr[6];
        uint8_t  num_dest;
```

```
}

// uint8_t  dest[6];  /* unicast destination (multicast data receiver) */
```

Multicast Forwarding Table A 'struct bat_priv' will contain a 'struct list_head mcast_forw_table' to remember its current forwarding state. Every mcast_forw_table_entry will remember the multicast addresses it is responsible for and the originators that have signed-up as multicast senders for this group (mcast_orig_list). Every mcast_forw_orig_entry will remember their multicasting sequence number status to be able to drop duplicates later, as well as the list of interfaces it has to forward multicast data packets to (mcast_if_list). Finally, every mcast_forw_if_entry remembers the nexthop destinations behind this interface (mcast_nexthop_list) and when they have last been seen (timeout) to clean up the table frequently.

Note, that this design has been chosen to be able to quickly check, if a certain, incoming multicast data packet needs to be forwarded and not traversing any list more than once in the worst case. Only the rcu protected, rather small if_list might be traversed more than once as the table is only saving the if_num and needs to look-up the according batman_if here again. However, this is a simpler and safer solution than storing a pointer to the according batman_if (which might vanish).

Furthermore, a mcast_forw_if_entry remembers its number of nexthops, so that a decision whether a multicast data packet should be forwarded via unicast(s) or one broadcast can be made quickly, without having to recount all nexthop entries.

```
struct mcast_forw_nexthop_entry {
        struct list_head list;
        uint8_t neigh_addr[6];
        unsigned long timeout; /* old jiffies value */
};

struct mcast_forw_if_entry {
        struct list_head list;
        int16_t if_num;
        int num_nexthops;
        struct list_head mcast_nexthop_list;
};

struct mcast_forw_orig_entry {
        struct list_head list;
        uint8_t orig[6];
        uint32_t last_mcast_seqno;
        TYPE_OF_WORD mcast_bits[NUM_WORDS];
        struct list_head mcast_if_list;
};

struct mcast_forw_table_entry {
        struct list_head list;
        uint8_t mcast_addr[6];
        struct list_head mcast_orig_list;
};
```