

# Reference Manual of B.A.T.M.A.N Experimental

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Related documentations and overview . . . . .	2
1.2	The path-detection algorithm . . . . .	3
<b>2</b>	<b>Debug-Level Parameters</b>	<b>4</b>
<b>3</b>	<b>Parametrizing the Core Routing Algorithm</b>	<b>6</b>
3.1	Bidirectional link timeout . . . . .	6
3.2	Neighbor ranking frame size (NBRF) . . . . .	7
3.3	Re-broadcast delay . . . . .	7
3.4	Tradeoff: Long versus short but poor-link paths . . . . .	10
3.4.1	Accept non-quickest OGMs depending on number of additional hops . . . . .	11
3.4.2	Accept non-quickest OGMs with reduced probability . . . . .	12
3.5	The impact of asymmetric links . . . . .	12
3.5.1	Reflecting link qualities with B.A.T.M.A.N . . . . .	14
3.5.2	Reflecting the link transmit quality (TQ) . . . . .	15
3.5.3	Reflecting the link receive quality (RQ) . . . . .	15
<b>4</b>	<b>Customizing the Daemon for Individual Network Requirements</b>	<b>16</b>
4.1	Achieving in-the-field protocol migration . . . . .	17
4.1.1	Customizing the protocol ports . . . . .	17
4.1.2	Customizing the used routing table and priority rules . . . . .	18
4.1.3	Customizing the tunnel IP-address ranges . . . . .	18
4.2	Reconfiguring network integration . . . . .	18
4.3	Hiding redundant topology information beyond the local neighborhood . . . . .	19
4.4	Miscellaneous . . . . .	20
<b>5</b>	<b>Proposed Parametrization Sets</b>	<b>20</b>
5.1	The <code>-bmxddefaults</code> switch . . . . .	20
5.2	The <code>-graz-2007</code> switch . . . . .	20
5.2.1	Assigned parameters and arguments . . . . .	21
5.2.2	Summary of measurement results . . . . .	22
5.3	The <code>-generation-III</code> switch . . . . .	24
<b>6</b>	<b>Tutorials</b>	<b>24</b>
6.1	Setting up a basic testbed . . . . .	24
6.2	Setting up internet gateway and client nodes . . . . .	31

## 7 Acknowledgments

38

# 1 Introduction

B.A.T.M.A.N is a proactive routing protocol targetting on networks with dynamically changing link characteristics. The algorithm is designed to deal with networks that are based on unreliable and lossy links which is typically the case in wireless mesh networks.

The batmand-experimental implementation is based on the code basis of batmand-0.3 and provides a number of experimental extensions to the batmand-0.2 and batmand-0.3<sup>1</sup> implementations. This document aims to provide the necessary background information to understand and use the features and underlying concepts of batmand-experimental.

## 1.1 Related documentations and overview

Generally, batmand-experimental can be used in the same way as the currently stable batmand-0.2 and batmand-0.3 branches. All the following documentations can be used.

- The batmand howto written by Wesley [1]
- The batmand manpage written by Wesley [2]
- The batmand INSTALL from the subversion repository [3]

The main differences for the above given documentations are:

- The exact download URL for the batmand-experimental sources [4]. Pre-compiled binaries for various linux systems can be found in the corresponding folder at:  
<http://downloads.open-mesh.net/batman/development/> .
- Executing batmand-experimental with the same parameters as used for batmand-0.2 results (except for the path detection algorithm) in the same behavior. Starting it with the *-generation-III* directive enables even the same path detection algorithm as implemented in batmand-0.2 But batmand-experimental offers a bunch of optionally switches which can be used for parametrizing the core routing algorithm and which allow a more flexible network integration. A brief summary of these switches can be obtained by executing the binary with the

```
--dangerous -H
```

switches. More about the switches is subject to the remainder of this document.

Although, batmand-experimental incorporates a number of extensions, the high-level concepts of the underlying algorithm are still the same, leaving existing technical documentations a worthwhile lecture. These are namely

- The B.A.T.M.A.N-Status Report [5].

<sup>1</sup>The full truth is: Recent efforts towards a new BATMAN-IV algorithm have lead to some different concepts in batmand-0.3 which are not implemented in batmand-experimental.

- The B.A.T.M.A.N online specification [6]

The remainder of this document is organized as follows. Section 1.2 provides some general background information about the BATMAN algorithms and its flooding mechanism. Section 2 to 4 then continues with the detailed explanations of the concepts and parametrization options implemented in batmand-experimental. And Section 5 summarizes some promising parametrization sets of batmand-experimental. You are welcome to commit your favorite parametrization set here. There are a few tutorials in Section 6 which briefly summarize the necessary steps to setup a simple mesh network and to activate, parameterize, and observe some of the optional concepts of batmand-experimental. Finally, Section 7 reveals the names of the people that are investigating so much of their time in the development of the B.A.T.M.A.N. routing protocol.

## 1.2 The path-detection algorithm

The general path-detection algorithm works as follows. Every node propagates the knowledge about its own existence over the mesh simply by flooding the network with originator messages (OGMs). Each OGM can be uniquely identified by a sequence number and the IP address of the node that initiated the message. OGMs sent via a poor or congested link will suffer from delay and packet loss. OGMs flooded via good and uncongested links will propagate further, faster, and more reliably.

Every node selects one of its neighbors as the best next hop towards a specific other node. This process of identifying the best next hop towards a distant node is called neighbor ranking. The selected neighbor is referred to as the best-ranking neighbor. Every node maintains one best-ranking neighbor for each known node (and interface) in the mesh. The neighbor-ranking algorithm simply selects the neighbor via which it received (and accepted) the most recent OGMs from the node that initiated the OGMs.

The propagation of OGMs over the mesh relies on intermediate nodes on the paths to re-broadcast a received OGM. One general rule of the batman algorithm is: Each node re-broadcasts only those OGMs received via its currently best-ranking neighbor. This way, the path that proved to be the quickest and most reliable will establish as a continuous unidirectional route from the receiver to the originator of the OGM.

By intentionally not accepting an OGM for the neighbor ranking, each node (along the propagation path of an OGM) also has the possibility to counteract on the propagation via specific links. This way, each node can influence the establishment of resulting end-to-end routes in the mesh. In batman-experimental the decision whether to accept or not accept a received OGM for the neighbor ranking is made in the configurable "acceptance-function". Parameters exist to control the acceptance-function when to accept or not to accept a received OGM for the neighbor ranking depending on local observations. Such observations may be the link quality to the neighbor via which the OGM has been received, the latency of the OGM, or the number of hops the OGM has passed.

The idea is, that each node along the propagation path of an OGM expresses negative observations about the link via which an OGM was received by simply reducing the probability with which it accepts and further propagates the received OGM. Thereby OGMs selectively flooded on good routes will propagate further, faster, and more reliably. The computation of the best neighbor towards a distant node is reduced to the counting on local observations of

the reality. The need for computation and knowledge of the complete end-to-end paths at a single node is eliminated. Instead, it is divided to all participating nodes in the mesh. Each node perceives and maintains only the information about the best next hop towards all other nodes.

## 2 Debug-Level Parameters

The main difference between the debug-level outputs of batmand-0.2 and batmand-experimental is the output of debug level one.

Parameter: `-d 1`

Debug level 1 lists all other originators (batman nodes and interfaces) and all links known by the node. The output is organized in one line per known originator. Each line is organized in columns where the first column represents the IP of the known originator. Further information is given in the following columns. Be aware that all values counting received or re-broadcasted or whatsoever OGMs only consider the most recent OGMs. "Recent" OGMs are identified by validating that their sequence numbers fall into the current neighbor-ranking-frame (NBRF) size (as can be specified with the `-window-size` parameter). The upper frame boundary of this frame is always limited by the most recent OGM received from a given originator.

- 2. column, captioned with `viaIF`** Indicates the interface via which packets are currently routed towards the originator.
- 3. column, captioned with `Router`** Indicates the IP address of the best-ranking neighbor. This is the neighbor via which packets are currently routed toward the originator given in the first column.
- 4. to 7 column, captured with (`brc rcvd`, `lseq lvid`)** Provides information about the currently best-ranking neighbor.
  - 4. column, captured with `brc`** Indicates the number of OGMs re-broadcasted for the given originator and received via the neighbor indicated in the 3. column. This number also indicated the number of OGMs considered for the neighbor-ranking to evaluate the best neighbor towards the given originator.
  - 5. column, captured with `rcvd`** Indicates the total number of OGMs received via the given neighbor. This number can be different (and usually is) form the value indicated in the 4. column because not every OGM received from the best-ranking neighbor is accepted for being re-broadcasted. The decision whether a received OGM is also accepted for for being re-broadcasted (and for the neighbor-ranking) may depend on the asymmetry of the link to this neighbor, on the number of hops it has passed, or whether a specific OGM has been received via this neighbor before it could be received from any other neighbor.
  - 6. column, captured with `lseq`** Indicates the last sequence number known from the given originator.

- 7. column, captured with `lvld`** Indicates the elapsed time (in seconds) since the last OGM has been received from the given originator.

**Optionally link-information block, captured with (`viaIF RTQ RQ TQ`)** If the given originator is also a direct neighbor of this node further information is provided for each link identified to this neighbor.

- 1. column of the link-information block, captured with `viaIF`** Indicates the interface for which a link has been identified to the given originator interface of the neighboring node. The neighboring batman interface may be seen via different local interfaces. Then, different links exist to that neighboring interface and each link is represented with its own link-information block, indicating the corresponding local interface as the `viaIF`. The local interface with the best link to the neighboring batman interface is also the `viaIF` shown in the second column.
- 2. column of the link-information block, captured with `RTQ`** Indicates the Round Trip Quality measured to the neighboring interface. This value represents the amount of "recent" own OGMs, re-broadcasted by the neighboring node, and received back by this node. In human context the `RTQ` value indicates the probability with which I hear somebody repeating what I just said.
- 3. column of the link-information block, captured with `RQ`** Provides a measure for the directed link quality from the neighboring interface to this local interface. In human context: How good do I hear my neighbor. It is similar to the `LQ` value known from OLSR-ETX. The batman-`RQ` value counts the amount of OGMs which have been "recently" received from the neighboring interface.
- 4. column of the link-information block, captured with `TQ`** Indicates the directed link quality from the local interface to the neighboring interface. This value is similar to the `NLQ` value known from OLSR-ETX. In batman it is calculated based on  $TQ = RTQ/RQ$ . In human context: How good can I expect my neighbor to hear what I am saying.

**Trailing columns** If more than one neighbor to the originator exists, each additional neighbor is listed in a separate trailing column. Each neighbor is presented with its IP address and the number of OGMs received and accepted via this neighbor.

The following block shows an example of the output.

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bmX 10.16.0.11, WindSize: 100, BLT: 20, OGI: 1500, UT: Od 7h18m
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternatives..
10.16.0.12      vlan1:bmX  10.16.1.12 (100 100 17321 1)
10.16.0.5       eth1:bmX   10.16.0.5 ( 56 78 17513 0) [ eth1:bmX 47 67 70]          10.16.1
10.16.1.12      vlan1:bmX  10.16.1.12 (100 100 17321 0) [vlan1:bmX 100 100 100]
10.16.0.13      eth1:bmX   10.16.0.13 ( 94 99 17430 1) [ eth1:bmX 82 94 87]
10.16.0.3       vlan1:bmX  10.16.1.12 ( 47 47 17544 1)          10.16.0.5 ( 26)          10.16.0
10.16.0.20      vlan1:bmX  10.16.1.12 ( 77 77 17378 2)          10.16.0.5 ( 42)          10.16.0
```

### 3 Parametrizing the Core Routing Algorithm

Batman uses a distributed algorithm. Each node link-locally promotes or counteracts on the propagation of OGMs via specific hops and thereby also promotes or counteracts on the establishment of end-to-end routes along these hops.

People, willing to do some experimentation or fine tuning on the algorithm can do so by parametrizing some of its behavior.

#### 3.1 Bidirectional link timeout

```
Parameter: --bi-link-timeout <value> : set bidirectional timeout value
Parameter: /b <value> : must be attached after an interface name
                    to set individual bidirectional-timeout value for this interface.
                    default: 2, allowed values: 1 <= value <= 100
```

**Background:** Batman only promotes OGMs received via a specific neighbor if the link to that neighbor is identified as a bidirectional link. A link is defined as a direct transmission capability between two interfaces on two different batman nodes. A link is considered bidirectional if communication via that link is possible in both directions. Each node makes its own decisions about the bidirectional-link status to neighboring batman interfaces. Each node always rebroadcasts every OGM received directly from an neighboring batman interface. It is rebroadcasted on the same interface where it has been received and with the following flags:

- The Is Direct Flag (IDF) to indicate the direct reception of the OGM.
- If the direct link via which the OGM has been received is not identified as a bidirectional-link by the receiving node with a marked Unidirectional Flag (UDF). All batman interfaces (except the originator interface of the OGM) do discard OGMs with a marked UDF flag.

A link is identified as bidirectional if:

- OGMs initiated on the behalf of a batman interface could be received back from the neighboring batman interface.
- It was received on the same interface on which behalf it was originated and send.
- It is marked with the IDF flag.

Because (wireless) link qualities are expected to change over time, the bidirectional-link status is re-evaluated whenever its status is questioned. The status is evaluated by checking that at least one of the latest X self-initiated OGMs has been directly received back (as described above) from the neighboring batman interface.

The *-bi-link-timeout* can be used to parameterize the variable X. Actually, this parameter does not really configure a timeout. We just kept this name for traditional reasons. If you want to translate it to a corresponding time value, the bi-link-timeout value must be multiplied with the originator interval.

Setting the bi-link-timeout to 1 makes the bidirectional-link check very strict because every OGM that failed to be successfully replied by the neighboring interface will result in a negative bidirectional-link status. The status persists until one of the following OGMs could be successfully replied. Setting the bi-link-timeout to a larger value does not cause the bidirectional-link check to fail until more than X subsequent OGMs failed to be successfully replied by the neighboring batman interface.

In batmand-0.2 the default value for the bi-link-timeout is 2 because the bidirectional-link check is the only means for identifying and reacting on strongly asymmetric links. In batmand-experimental other mechanisms exist to identify and to counteract on asymmetrical links. If these mechanisms are enabled a very large bi-link-timeout (e.g. 20) should be acceptable.

### 3.2 Neighbor ranking frame size (NBRF)

Parameter: `--window-size <value>` : set neighbor ranking frame (NBRF) size  
default: 128, allowed values:  $1 \leq \text{value} \leq 250$

**Background:** Batmand-0.2 selects the best neighbor towards a distant node by counting the number of "recent" OGMs received via each neighbor. The window-size parameter specifies the range of sequence numbers that fall into this "recent" category. As an example, if the window-size has been configured to 10 and the latest OGM received from a distant node has a sequence number of 100, then all previously received OGMs with a sequencenumber between 91 and 100 would be considered for the best-neighbor ranking.

As a consequence, a mesh with each node configured with a window-size of 10 will converge very fast because all routing decisions made by the algorithm are based on only the latest 10 sequence numbers. The negative side effect is, that temporary variations in packet loss have a quite huge impact on the path detection. The more measurement samples are considered by a statistic, the more reliable it becomes (in mathematical terms this can be expressed with the confidence interval). Measurement showed that for semi static mesh networks a large window-size (e.g. 128) results in much better performance and much less needless route changes.

On the other hand, if you think about setting up a mesh which allows almost seamless handover at walking speed, then you may try a rather small window-size (e.g. 10).

### 3.3 Re-broadcast delay

Parameter: `--re-brc-delay <value>` : set maximum of random re-broadcast delay in milliseconds  
default: 0, allowed values:  $0 \leq \text{value} \leq 100$

**Background:** Relieving the neighbor detection mechanism from the hidden node problem. Assuming the typical hidden node scenario with nodes A, B, and C as illustrated in Figure 1.

Whenever B broadcasts its OGMs, these messages are received simultaneously by A and C. According to the batman-0.2 algorithm these OGMs are instantly re-broadcasted by node A and node C. Because A and C can not hear each other they have no chance to coordinate their transmissions using the 802.11 carrier-sense mechanism.

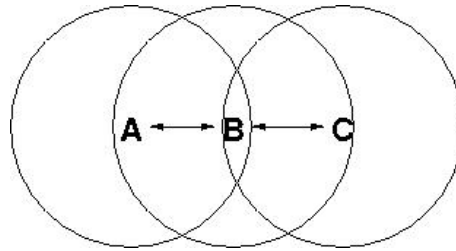


Figure 1: Illustrating a typical hidden node problem with B being in communication range with A and C. A is completely out of the communication range of C.

As a consequence (the hidden nodes) A and C will both rebroadcast Bs OGM at almost the same moment and cause a collision at (the exposed node) B. Therefore, node B will miss many of its own OGMs re-broadcasted by its neighboring nodes and will often consider these links as non-bidirectional. Subsequently, node B will ignore OGMs from its neighbors in the neighbor-ranking process due to the failed bidirectional-link check and (if exist) favor an alternative path. This happens, despite having a collusion-free unidirectional connection from B to A and from B to C.

On the other hand, node A (and same for C) is not even aware of that problem. OGMs initiated by A are only heard and re-broadcasted by B and received back by A again. Chances of A and C choosing the same moment for initiating their own OGMs are very rare because every node chooses a random delay for the moment when initiating their own OGMs.

The `-re-brc-delay` switch can be used to configure each node to wait also a random delay before re-broadcasting an OGM received from a neighboring node. This phenomenon can be directly observed as the following debug-information shows: BATMAN running only on B (105.130.0.67) and C (105.131.41.3) :

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bat 105.130.1.67, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h 4m
Originator      viaIF      Router (brc rcvd lseq lvl) [ viaIF RTQ RQ TQ].. alternatives...
105.131.41.3    eth1:bat   105.131.41.3 ( 42 42 268 0) [eth1:bat 52 50 100]
```

```
B.A.T.M.A.N. 0.3-exp rv641, MainIF/IP: eth1:bat 105.131.41.3, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h 4m
Originator      viaIF      Router (brc rcvd lseq lvl) [ viaIF RTQ RQ TQ].. alternatives...
105.130.1.67    eth1:bat   105.130.1.67 ( 62 62 283 1) [eth1:bat 45 89 50]
```

you can see, that the RTQ (the round-trip-probability), the RQ and the calculated TQ values are quite similar from node A and Bs points of view.

when I start batman also on A and wait a while BATMAN running on A (105.130.30.1), B (105.130.0.67) and C (105.131.41.3):

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bat 105.130.30.1, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h 3m
Originator      viaIF      Router (brc rcvd lseq lvl) [ viaIF RTQ RQ TQ].. alternatives...
105.131.41.3    eth1:bat   105.130.1.67 ( 12 12 635 8)
105.130.1.67    eth1:bat   105.130.1.67 ( 60 60 661 0) [eth1:bat 49 90 54]
```

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bat 105.130.1.67, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h11m
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternatives...
105.131.41.3   eth1:bat   105.131.41.3 ( 17 17 644 1) [eth1:bat 21 44 47]
105.130.30.1   eth1:bat   105.130.30.1 ( 25 25 217 0) [eth1:bat 27 50 54]
```

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bat 105.131.41.3, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h10m
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternatives...
105.130.1.67   eth1:bat   105.130.1.67 ( 56 56 666 0) [eth1:bat 40 85 47]
105.130.30.1   eth1:bat   105.130.1.67 ( 16 16 220 1)
```

It can be seen that the measured RQ values (at node A and B) did NOT relevantly change (indicating, that the link quality has not really changed). But the measured RTQ value measured at exposed node B (to node A) decreased by almost 50% due to the collisions. Because the TQ value is calculated based on  $TQ = RTQ/RQ$  it indicates a wrong value. A simple test shows that the collision problem could be almost resolved by applying a random delay of up to 15 ms. Using

```
$batmand --window-size 100 --re-brc-delay 15 eth1:bat
```

the debug-level one output now shows:

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bat 105.130.30.1, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h 3m
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternatives...
105.131.41.3   eth1:bat   105.130.1.67 ( 24 24 208 0)
105.130.1.67   eth1:bat   105.130.1.67 ( 62 62 197 0) [eth1:bat 53 83 63]
```

```
B.A.T.M.A.N. 0.3-exp, MainIF/IP: eth1:bat 105.130.1.67, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h 3m
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternatives...
105.131.41.3   eth1:bat   105.131.41.3 ( 34 34 208 1) [eth1:bat 46 49 93]
105.130.30.1   eth1:bat   105.130.30.1 ( 41 40 195 1) [eth1:bat 45 58 77]
```

```
B.A.T.M.A.N. 0.3-exp rv551, MainIF/IP: eth1:bat 105.131.41.3, WindSize: 100, BLT: 2, OGI: 1000, UT: 0d 0h 3m
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternatives...
105.130.1.67   eth1:bat   105.130.1.67 ( 61 61 197 1) [eth1:bat 47 89 52]
105.130.30.1   eth1:bat   105.130.1.67 ( 33 33 195 2)
```

By enabling the `-re-brc-delay` switch you may also consider to use the accept duplicates switches described below.

It should be mentioned that such a random delay has further side effects. The most critical is probably that the batman-0.2 algorithm considers only the first OGM (received with a new sequence number) for the neighbor ranking. This way, the batman-0.2 algorithm implements its strong favor for the quickest and shortest paths.

Applying the `-re-brc-delay` parameter, the propagation speed of OGMs is probably more related to the random delay (experienced by each OGM at each intermediate hop) than on the actual number of hops it has passed. You are welcome to continue reading for a deeper discussion about the pros and cons of this side effect.

### 3.4 Tradeoff: Long versus short but poor-link paths

In a wireless multi-hop mesh, there are typically several possible paths between two nodes. Assuming the probably most simple setup as illustrated in Figure 2, each node is in physical communication range of each other. However, with different link qualities.

For packets that need to be delivered from A to C there are two possible paths. The two-hop path from A via B to C and the direct path from A to C.

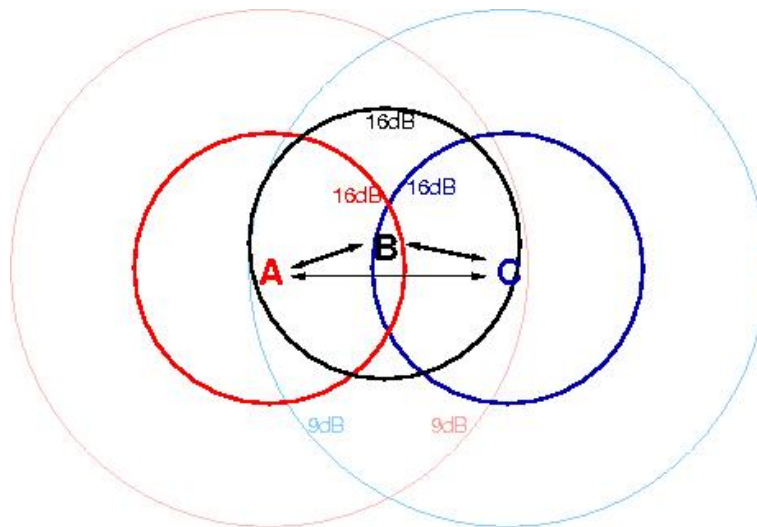


Figure 2: Illustrating the tradeoff between the long two-hop path with good link qualities versus the one-hop path with a poor link quality.

Each of the two short links ( $A-B$  and  $B-C$ ) has a better link quality than the direct link between A and C. For the first path, each end-to-end packet delivery requires two transmissions. One from A to B and another from B to C. Choosing the direct link, one transmission might be sufficient. This seems to result in a shorter transmission time for the end-to-end packet delivery and being more efficient with the overall bandwidth consumption. But (due to the lower link quality) each transmission is also more prone to errors and therefore might require casual retransmissions. Probably even more important, the 802.11 MAC layer might choose a less robust but faster data-rate for the packet transmission via the two short links. This would once again decrease the required transmission time and maybe even outperform the transmission via the direct link. According to the receiver performance requirements specified in [8] page 31, a signal degradation of 7dB may result in a data-rate degradation from 18 to 6 MBits/second. In this case, choosing the two-hop path should even pay out when considering the additional transmissions necessary for the extra hop.

The Batman-III algorithm (as implemented in batman-0.2) has a strong favor for short paths – being rather ignorant about poor links along that path. This is because:

- The end-to-end path established between two batman nodes is the result of the neighbor ranking performed on all intermediate nodes along that path.

- Only the first (quickest) OGM received via a specific neighbor is considered for the neighbor ranking.
- All OGMs are broadcasted using an equal and (depending on the WLAN driver) very robust and slow data rate. The link quality in terms of bandwidth is not considered until the link is so lousy that many OGMs get lost.<sup>2</sup>
- Therefore, an OGM received directly from its originator is always quicker than a re-broadcasted OGM received via an intermediate hop. OGMs propagated over a two-hop path are generally quicker than those propagated over a more-than-two-hop path.

Given the scenario illustrated in Figure 2, whenever an OGM from C is directly received by A, the same OGM is ignored when received via the two-hop path (from C via B to A). Even if all of C's OGMs that were propagated via the two-hop path C-B-A could be received by A, but only 60 % of C's OGMs make it via the direct link C - A, the neighbor ranking at node A would still count 60 OGMs for the direct link and only 40 OGMs for the path via neighbor B. Thus, making neighbor C the winner of the neighbor ranking and choosing the direct link for packet delivery from A to C. Drawing it even more dramatically, the decision could result in favoring a single-hop path with a data rate of 6MBit/s and a packet loss of 40 % instead of a two-hop path with a per-link data rate of up to 18 MBit/second and no packet loss.

The parameters described in the following can be used to handle this challenge.

### 3.4.1 Accept non-quickest OGMs depending on number of additional hops

A straight forward approach to relieve the strong favor for short paths would be to also accept OGMs for the neighbor ranking that propagated via an alternative and non-quickest path. Thus, also accept OGMs that could be identified as a duplicate (due to the same sequence number and originator IP) of a previously received OGM. But care must be taken to not accept duplicates that travelled a loop. This could be sorted out by limiting the number of additional hops such a duplicate is allowed to have passed compared to the previously received OGM.

Given a similar setup as illustrated in Figure 1, an OGM initiated by node A and re-broadcasted by node B, may subsequently be received and rebroadcasted by node C, and finally be received by node B for a second time. The OGM has been transmitted from A to B, to C, and back to B again. In this case, after being received by node B for the first time, the OGM has passed the smallest possible loop with two additional hops. Each additional hop an OGM is propagated along a path, the time to live (TTL) field is decremented by one. Therefore, each node can easily identify the number of additional hops a non-quickest OGM has passed (compared to the previously received OGM) by comparing contained TTL values.

The limit of additional hops, such duplicate OGMs are allowed to have passed, for being accepted for the neighbor ranking could be configured using the `-dups-ttl` parameter. Applying a value of 2 would reject every duplicate OGM that passed 2 or more additional hops than any previously received OGM – ensuring loop-free propagation of OGMs.

<sup>2</sup>Varying and thereby choosing also faster data rates for the emission of OGMs might be a promising approach to tackle this shortcoming.

Parameter: `--dups-ttl <value>`

Accept non-quickest OGMs to relieve preference for shortest path.

(< value > - 1) defines how much smaller the TTL of a non-first OGM can be compared to the largest TTL received so fare (with the same originator IP and sequencenumber).

default: 0 (disabled), allowed values: 0 <= value <= 10

I recommend to combine the `-dups-ttl` parameter with the `-re-brc-delay` parameter.

Values larger than 2 should not be applied until using also one of the parameters described in the following Section.

### 3.4.2 Accept non-quickest OGMs with reduced probability

The problem with looped OGMs is not the OGM itself but the path promoted by that message. The end-to-end path established between two nodes is the result of the neighbor ranking performed on each hop along that path. If it ever happens that the best-ranking neighbor becomes selected due to looped OGMs, the path established via the selected neighbor causes a routing loop. This should be strictly avoided. On the other hand, from the perspective of each node along that path, this path is only changed if even more recent OGMs are received and accepted via an alternative neighbor, than have been received and accepted via the currently best-ranking neighbor. By ensuring that duplicate (and potentially looped) OGMs are only partially accepted, the number of looped OGMs could never outperform the number of OGMs received via the real best-ranking neighbor.

The following two parameters can be used to accept duplicate OGMs only with a reduced probability

Parameter: `--dups-rate <value>`

Accept non-quickest OGMs to relieve preference for shortest path.

< value > defines the general probability with which non-quickest OGMs are accepted.

default: 0 (disabled), allowed values in percent: 0 <= value <= 100

Parameter: `--dups-ttl-degradation <value>`

Accept non-quickest OGMs to relieve preference for shortest path.

< value > defines the probability degradation for each additional hop

(compared to the OGM arrived via the shortest path) with which non-quickest OGMs are accepted.

default: 0 (disabled), allowed values in percent: 0 <= value <=100

## 3.5 The impact of asymmetric links

The phenomenon of an asymmetric link can be observed in every-day life. Given for example the conversation between a Diskjockey and one of his fans as illustrated in Figure 3. In this scenario the DJ, having his ears covered with booming headphones, is well understood by the fan. But in the other direction, because the DJ is exposed to the noisy music from his headphones, he can not understand anything from his fan.

Assigning an overall verdict, one may qualify the quality of the communication channel between the DJ and his fan as *average*. However, relying on such a verdict would be misleading



Figure 3: Illustration of an asymmetric communication channel.

and inefficient. From the fans point of view, choosing a redundant but oral transmission technique (for example by speaking loud and slowly, to deliver an important message, will fail. But if the fan knew about the asymmetry of the link, he may have chosen to write down the message. From the DJs point of view, choosing to speak loud and slowly is simply not necessary. It is a waste of energy because there is actually no need to speak loudly. And it is inefficient because his message would have been understood even when talking faster.

The impacts of asymmetric links in wireless mesh networks are even more versatile (and probably unknown). The following example shall illustrate this. Considering for example a very simple asymmetric links scenario as given in Figure 4.

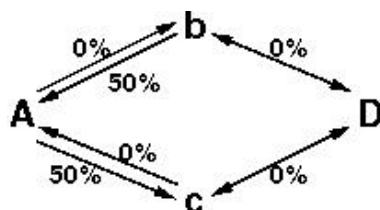


Figure 4: Illustration of a simple multi-hop asymmetric-link scenario.

The figure shows four nodes (A,b,c,D) and four existing links between them  $A - b$ ,  $b - D$ ,  $A - c$ , and  $c - D$ . There are two potential end-to-end paths between node A and D, one going via the intermediate node b and another via node c. The links  $b - D$  and  $c - D$  are symmetric, perfect links with 0 % packet loss. The links  $A - b$  and  $A - c$  are asymmetric links with 0 % packet loss for the directions  $A \rightarrow b$  and  $c \rightarrow A$ , but 50 % loss for the directions  $b \rightarrow A$  and  $A \rightarrow c$ . For the task of delivering a ping request packet from A and D and sending the ping reply back from D to A there are the following four routing options:

1. Via  $A \rightarrow b \rightarrow D \rightarrow b \rightarrow A$  with a total round-trip loss of 50 %.
2. Via  $A \rightarrow b \rightarrow D \rightarrow c \rightarrow A$  with a total round-trip loss of 0 %.
3. Via  $A \rightarrow c \rightarrow D \rightarrow b \rightarrow A$  with a total round-trip loss of 75 %.
4. Via  $A \rightarrow c \rightarrow D \rightarrow c \rightarrow A$  with a total round-trip loss of 50 %.

This example is of course just an naive fallacy compared to the complexity of real-live wireless-mesh networks but it should illustrate the potential efficiency margin that might result from properly or improperly reflecting asymmetric links.

In BATMAN, the link characteristics are differentiated in receive quality (RQ), transmit quality (TQ), and the product of both values  $RTQ = RQ * TQ$ . From node A's point of view (Figure 4), the RQ of the link to neighbor b is 50 % and the TQ to neighbor b is 100 %. From node b's point of view it's just the other way around.

The RQ of a specific link is measured by counting the amount of recently received OGMs from the corresponding link neighbor. The RTQ of a specific link is measured by counting the amount of recently received own OGMs, rebroadcasted from the corresponding link neighbor. And the TQ is calculated based on  $TQ = RTQ/RQ$ . Some further information about link qualities is also given in Section 2.

In the first place, from the sender's point of view, the TQ of a given link seems to be more relevant than the corresponding RQ. But the MAC protocol of 802.11 [7] demands that each unicast transmission from A to B is successfully acknowledged back from B to A. This demand makes a successful transmission of the acknowledgment message as important as the unicast-data message itself. On the other hand, the transmission of a MAC acknowledgement is subject to a number of advantages compared to the transmission of a unicast-data message.

- The packet size of a MAC acknowledgement is very small and usually very much smaller than the size of the corresponding data packet, leaving a much smaller risk for the Acknowledgment message to collide.
- The time slot during which the MAC acknowledgement is to be transmitted has already been reserved by the preceding data packet (with a similar result as the RTS/CTS mechanism could achieve for the data packets).
- Last but not least, the MAC acknowledgement are usually transmitted with a much more redundant modulation compared to the modulation used for the preceding data packet. This arms them with some additional resilience against interference and noise.

The above discussion provided a rationale for the differentiation of a link quality into its receive and transmit quality. However, properly weighting the relevance of the RQ and TQ characteristics for an unicast transmission is subject to further experiments. Some preliminary but promising weighting factors have been gained from the experiments at the wireless community weekend 2007 in Graz. The currently proposed parametrization of the batman-experimental implementation is described in Section 5.

### 3.5.1 Reflecting link qualities with B.A.T.M.A.N

The BATMAN-III algorithm establishes routes between nodes based on the number of OGMs received via a specific neighbor. This way, the algorithm promotes routes with good receive qualities (from the receiving nodes' point of view). But these routes are then used for sending. In other words, the OGMs initiated by each node install routing information for DOWN-link traffic which are actually optimized for UP-link traffic.

Referring to Figure 4, node A receives 100 % of the OGMs initiated by D and rebroadcasted via node c. On the other hand, node A receives only 50 % of D's OGMs via node b.

Therefore A would select c as its best-ranking neighbor towards D (obversely D would select b as its best neighbor towards A).

Thereby, the flooding mechanism of BATMAN-III inherently promotes the propagation via links with a good RQ. The reflection of the TQ characteristics could only be achieved indirectly with a very strict bidirectional link check. But even when applying the strictest possible bi-link-timeout, the total link-quality reflection only sums up to  $TQ * RQ^2$  per hop, giving the TQ a much smaller influence than the RQ.

BATMAN-IV has the capability to differentiate a link quality into its distinct directions and reflect these characteristics in the path detection algorithm. The idea is, that each node along the propagation path of an OGM expresses negative observations about the link via which an OGM was received by simply reducing the probability with which it accepts and further propagates the received OGM. By intentionally not accepting an OGM for the neighbor ranking, each node has the possibility to counteract on the propagation via specific links and influence the establishment of resulting routes in the mesh. Only accepted OGMs received via the best-ranking neighbor are reboradcasted.

### 3.5.2 Reflecting the link transmit quality (TQ)

The following parameters exist to configure the acceptance-function when to accept or not to accept a received OGM depending on the TQ to the corresponding link neighbor.

```
Parameter: --asymmetric-exp <value> : Ignore rcvd OGMs to respect asymmetric-links.  
Ignore with probability  $TQ^{\langle value \rangle}$ .  
default: 0, allowed exponent values:  $0 \leq value \leq 3$ 
```

```
Parameter: --asymmetric-weight <value> : Ignore rcvd OGMs to respect asymmetric-links.  
default: 0, allowed probability values in percent:  $0 \leq value \leq 100$ 
```

Regarding the TQ, a received OGM is accepted with probability p, according to following formula:

```
probability  $p \in [0..1]$   
neighbor ranking frame (NBRF) size  $s$   
link transmit quality  $t \in [0..s]$   
asymmetric-exp value  $e \in [0, 1, 2, 3]$   
asymmetric-weight value  $w \in [0..100]$   
 $p(t) = (w/100) * (t/s)^e$ 
```

### 3.5.3 Reflecting the link receive quality (RQ)

As described in Section 3.5.1, the reflection of the RQ characteristics (along the paths in a mesh) is an inherent characteristic of the flooding algorithm. The previous discussion in Section 3.5 and 3.5.1 has also indicated that a prevailing reflection of the link-RQs can even be contraproductive for the establishment of performant routes.

Therefore, the question is rather how this inherent behavior can be relieved than how the reflection of the RQ characteristics can be further enforced. Kind of brute-force approach to

reduce the influence of a weak RQs is to simply decrease the chance that OGMs get lost. And one way to achieve that is to simply broadcast OGMs multiple times, increasing the chance that at least one of the multiple times broadcasted OGMs could be received. Besides the obvious disadvantage of an increased protocol-traffic overhead, this approach incorporates also a number of advantages.

- By marking the first broadcasted OGM of each sequence number with a dedicated bit, each node can still measure the correct RQ, RTQ, and TQ to their direct neighbors.
- By only broadcasting OGMs twice which passed the acceptance-function and which were received via the best-ranking neighbor the amount of additional protocol traffic could be limited.
- All previously described mechanisms (for example to reflect the TQ of a link) remain functional.
- The impact of the RQs is not fully eliminated. It is still reflected but with a smaller influence.
- This approach helps to increase the propagation range (in terms of hops) of the OGMs.

The following parameter exist to configure the flooding mechanism to broadcast the same OGMs even multiple times.

```
--send-clones <value> : (re-)broadcast OGMs with given probability
/c <value> : attached after an interface name
to specify an individual re-broadcast probability for this interface.
default: 100, allowed probability values in percent: 0 <= value <= 300
```

The parameter takes a probability argument in the range of zero to several hundred. An argument in the range of zero to hundred specifies the probability in % with which a to-be-propagated OGM is broadcasted once. An argument in of 150 specifies that each to be propagated OGM is broadcasted at least once and with a probability of 50 % even twice. An argument of 200 specifies that each to-be-broadcasted OGM is broadcasted exactly twice!

The `/c` parameter can be used to configure an individual (re-)broadcast probability for a specific interface. This makes particularly sense for wired lan interfaces because such links are usually not showing the huge packet-loss which is typical for wireless links.

## 4 Customizing the Daemon for Individual Network Requiremets

Batmand-experimental supports a number of parameters to customize the routing daemon for various network environments.

## 4.1 Achieving in-the-field protocol migration

A typical desire for an already existing mesh network is the possibility to test and compare a new routing protocol before any of the functionality needed by the currently operating protocol is disabled. A straight forward approach to achieve this desire is to execute the new protocol in parallel to the already existing protocol and ensure that none of the two protocols is influencing the other.

The batmand-0.2 and 0.3 implementations already provided all necessary functionality to operate in parallel to OLSR. The following batmand-experimental parameters can be used to simultaneously run two (or even more) batman versions on the same devices. Therefore, the following system attributes should be decoupled.

- The IP-address ranges (or netmasks) used for each routing protocol should be non-overlapping. Achievable by equipping the already used network interfaces with an additional alias address and a non-overlapping address range.
- The port numbers (used for transferring protocol data) and the unix-socket (used to obtain runtime-debug informations) must be different. A corresponding parameter is described below.
- The routing table used by the different protocols. A corresponding parameter is described below.
- The routing-priority rules used for assigning dedicated routing tables to selected address ranges. A corresponding parameter is described below.
- The tunnel IP-address ranges used to tunnel internet traffic between the internet-GW and client nodes. A corresponding parameter is described below.
- Optionally, to simplify monitoring of cpu-load and memory-consumption, the binary name of the daemons should be different. Achievable by copying the corresponding binary file to an new name before executing it.

Related examples are also given in Section 6.

### 4.1.1 Customizing the protocol ports

By default, batman uses the UDP port 4305 (as assigned by IANA [?]) for the flooding of OGMs. This port is referred as batmands base-port. Optionally, the two sequently following port numbers are used for gateway connectivity and visualization. The port number (base-port + 1) is used for tunneling packets between internet-gateway nodes and client nodes. The port number (base-port + 2) is used for sending visualization data to a dedicated visualization server. The base-port can be configured with the following parameter.

```
--base-port <value> : set base udp port used by batmand.  
    <value> for OGMs, <value+1> for GW tunnels, <value+2> for visualization server.  
    default: 4305, allowed values: 1025 <= value <= 60000
```

Whenever applying the `-base-port` parameter during the execution of the main daemon, this parameter must be applied again and with the same value for obtaining the correct debug informations. A simple example is given below.

```
root@ng1e:~# ln -s /usr/sbin/batmand /usr/sbin/bmxd
root@ng1e:~# bmxd --base-port 24305 --rt-table-offset 77 --prio-rules-offset 26600 ath0:bmxd
WARNING: You are using the experimental batman branch!
...
Long option: base-port with argument: 24305
Long option: rt-table-offset with argument: 77
Long option: prio-rules-offset with argument: 26600
Using interface ath0:bmxd with address 103.130.30.201 and broadcast address 103.255.255.255
root@ng1e:~#
root@ng1e:~# bmxd -c --base-port 24305 -d 1 -b
WARNING: You are using the experimental batman branch!
Long option: base-port with argument: 24305
B.A.T.M.A.N. 0.3-exp rv687, MainIF/IP: ath0:bmxd 103.130.30.201, WindSize: 128, BLT: 2, OGI: 1000, UT: 0
Originator          viaIF          Router (brc rcvd lseq lvl) [   viaIF RTQ  RQ  TQ].. alternatives..
103.130.30.202  ath0:bmxd 103.130.30.202 ( 26 26   95   0) [ ath0:bmxd 26 95 35]
root@ng1e:~#
```

#### 4.1.2 Customizing the used routing table and priority rules

The following switches can be used to change the default routing tables and the priority rules.

```
--rt-table-offset <value> : set base routing table used by batmand.
    Configures table <value> to be used for HNA routes, <value+1> for host routes,
    <value+2> for unreachable routes, and <value+3> for the default tunnel route.
    default: 65, allowed values: 2 <= value <= 250

--prio-rules-offset <value> : set base ip-rules priority used by batmand.
    default: 6600, allowed values: 3 <= value <= 32765
```

#### 4.1.3 Customizing the tunnel IP-address ranges

The following switch can be used to change the default IP-address range used for establishing tunnels between internet-GW and client nodes.

```
--gw-tunnel-network <ip-address/netmask> : set tunnel IP-address range leased out by GW nodes.
```

### 4.2 Reconfiguring network integration

```
--no-unreachable-rule : does not set the unreachable rule for host routes.

--no-prio-rules : does not set the default priority rules.

--no-throw-rules : does not set the default throw rules.

--resist-blocked-send : lets daemon survive if firewall blocks outgoing OGMs.
```

### 4.3 Hiding redundant topology information beyond the local neighborhood

Nodes may reduce the default TTL of their own OGMs to limit the number of hops that these OGMs are propagated through the mesh. This can be done for all OGMs or just for OGMs propagating the existence of particular interfaces. This does not affect the routing between other nodes in the mesh, but may be used to limit the range of presence (existence) of individual interfaces. For example, a node with three interfaces may be configured to send OGMs with a high TTL only for the first (primary) interface and a small TTL for OGMs representing the second and third (all non-primary) interfaces. This way, the node is always reachable via the IP of it's first interface. But it does not burden the nodes beyond its neighbor horizon with the efforts of processing, maintaining, and re-broadcasting OGMs from it's second and third interface.

One side effect of the one-TTL OGMs is that data-packets generated on these nodes may leave the node with a source address of such a secondary interface. This has the consequence that non-neighboring nodes could not reply to this source address, simply because the OGMs for this source address have never been propagated that fare.

The Solution to counter the above problem is that each multi-homed node automatically makes an HNA announcement for all non-primary (hidden) interfaces and propagates this HNAs with the OGM representing its primary interface. This way the IP addresses of the non-primary interfaces are also reachable beyond the local neighbor horizon. All HNA announcements end up in routing table 65. It tells the network stack to route packets with a destination address of a non-primary interface towards the IP address of the corresponding primary interface.

This is now the default behavior in 0.3

This behavior is can be controlled using the interface-specific `/a` and `/Aswitch`.

The following parameters can be used to reduce the TTL of OGMs representing specific interfaces.

```
--t <value> : change default TTL of originator packets.  
/t <value> : attached after an interface name  
to change the TTL only for the OGMs representing a specific interface  
default: 50, allowed values: 1 <= value <= 63  
  
/i : attached after an interface name  
to broadcast the OGMs representing this interface only via this interface,  
also reduces the TTL for OGMs representing this interface to 1.  
  
/a : attached after an interface name  
to add the IP address of this interface to the HNA list. Also  
reduces the TTL for OGMs representing this interface to 1 and  
broadcasts the OGMs representing this interface only via this interface  
  
/A : attached after an interface name  
to remove the IP address of this interface from the HNA list.
```

A small TTL for all self-initiated OGMs has some additional advantages for back-bone nodes. Nodes, which are not supposed to generate pay-load traffic itself and which were only

installed to improve the connectivity and coverage of the mesh by relaying other nodes traffic. Firstly, the topology and even the existence of the back-bone nodes could be completely hidden beyond their local neighbor horizon and secondly, the number of back-bone nodes (and resulting coverage) can be increased to any size with virtually no side affects to the overall traffic and processing cost.

#### 4.4 Miscellaneous

`--no-unresp-gw-check` : disables the unresponsive-GW check.

`--gw-change-hysteresis <value>`: Use hysteresis for fast-switch gw connections (-r 3).  
`<value>` for number additional rcvd OGMs before changing to more stable GW.  
default: 1, allowed values: 1 <= value <= 65

## 5 Proposed Parametrization Sets

Batmand-experimental implements a number of proconfigured parametrization sets. These parametrization sets have been gained from (subjectively) successfull experiments and deployments. The parameters applied by a specific set can be overwritten by subsequent parameters. The parametrization applied by a specific set is always revealed during startup sequence. If none of the available parametrization sets is specified explicitly, the default behavior of the daemon will conform to the `-bmx-defaults` switch.

### 5.1 The `-bmx-defaults` switch

The `-bmx-defaults` switch is supposed to always enable the latest and most promising mechanisms that are available in the current implementation. Therefore, configured parameters and values may change with future code versions.

When writing this document, this directive assigned the same parameters as described in Section 5.2.1.

See the daemons initialization printout to obtain information about the performed parametrization.

### 5.2 The `-graz-2007` switch

This switch reflects the experience gained at the wireless community weekend in Graz 2007. During the preparations of the event an experimental mesh network has been setup at the exhibition area. The network consisted of twenty nodes distributed over the 4 floors of the exhibition building. The nodes have been configured with a very small transmission power to achieve rather small cells. The setup resulted in routes with up to 7 hops. Each node was running the freifunk firmware [?] version 1.6.2 and three mesh routing protocol daemons in parallel, namely the firmware-buildin olsrd version 0.5.4pre [?], batmand-0.2, and batmand-experimental revision 623. The parametrization used for the experimental batman daemon are described in Section 5.2.1. A short summary about the gained experience is given in

Section 5.2.2. Some further informations about the experiment (including a topology snapshot and some smoke-ping based performance measurements) are available at [9].

### 5.2.1 Assigned parameters and arguments

As described, related parametrizations can be obtained from the daemons initialization print-out.

```
root@ng2e:~# batmand --graz-2007 ath0:bat eth0.0:bat
WARNING: You are using the experimental batman branch!
Applying graz-2007 !
This parametrization expects the first given interface argument to be a wireless interface !
Parametrization based on experience gained from the Wireless Community Weekend in Graz 2007!
Short option: o with argument: 1500
Long option: bi-link-timeout with argument: 20
Long option: window-size with argument: 100
Long option: dups-ttl with argument: 2
Long option: dups-rate with argument: 100
Long option: dups-ttl-degradation with argument: 2
Long option: send-clones with argument: 200
Long option: asymmetric-weight with argument: 100
Long option: asymmetric-exp with argument: 1
Long option: re-brc-delay with argument: 35
Using interface ath0:bat with address 10.10.0.2 and broadcast address 10.10.255.255
Using interface eth0.0:bat with address 10.10.1.2 and broadcast address 10.10.255.255
Interface eth0.0:bat specific option: /a
Interface eth0.0:bat specific option: /i
Interface eth0.0:bat specific option: /t 1
Interface eth0.0:bat specific option: /c 100
root@ng2e:~#
```

The selection of the given parameters was driven by the following considerations.

- One clear objective was to test the mechanisms to reflect asymmetric links as described in Section 3.5.
  - The *-asymmetric-weight 100 -asymmetric-exp 1* parameters are supposed to improve the reflection of the link transmit qualities during the path detection. They ensure that received OGMs are accepted only with a probability equivalent to the TQ measured for the incoming link (see Section 3.5.2).
  - The reasons for having a bidirectional link check becomes redundant because of the new mechanisms to reflect asymmetric links. Choosing a bi-link-timeout of 20 almost disables its behavior.
  - The *-send-clones 200* parameter is supposed to relieve the strong impact of the link-receive qualities as described in Section 3.5.3. It causes the daemon to re-broadcast all accepted OGMs (received via the best-ranking neighbor) twice.
- Another objective was to test the mechanisms to pay more attention on paths with many hops but good links (see Section 3.5.1). Related parameters have been configured

as *-dups-ttl 2 -dups-rate 100 -dups-ttl-degradation 2*. Ensuring, that also non-quickest received OGMs are accepted for the neighbor-ranking as long as these OGMs have not passed two or more additional hops. Also, each additional traversed hop is penalized with a 2%-acceptance degradation.

- *-o 1500* sets the originator interval to 1500. and *-window-size 100* sets the neighbor ranking frame (NBRF) size to 100. These values have been chosen rather arbitrarily. The selection was driven by the following considerations.
  - Choosing a smaller window size (than the default of 128) reduces the convergence latency – the protocol latency to react on topology changes. There have also been experiments with a window size of 64 and even 10 with reasonable results. However, a larger value clearly provides more information for the calculation of representative link and path statistics. On the other hand, choosing a window size of 10 allowed to even walk around on the exhibition side and experience seamless handover and path adaption. But it also resulted in lots of superfluous routing changes and less performant routes. Finally we stick to the window size of 100 because of its simple interpretation character which is equal to the percent unit.
  - Choosing a larger originator interval (than the default of 1000 ms) is a simple way to reduce the resource consumption as described in [5] Section 4.3. The side effect of an increased originator interval is an increased convergence latency. It is equivalent to the product of originator interval and window size. Therefore the increased window size could almost be balanced by the reduced window size.
- The *-re-brc-delay 35* parametrization evinced to adequately handle the hidden node problematic in scenarios with up to 5 neighbors. More background is given in Section 3.3.
- Finally, the *-graz-2007* parametrization is tailored for nodes with one wireless and zero or more wired interfaces. The parametrization automatically hides the existence of all non-primary interfaces beyond the direct neighbor horizon. Therefore each non-primary interface is configured with the */a /i /t 1* parameters (see Section 4.3).

Additionally, the previously *-send-clones 200* parameter is reset to its default value of 100 with the */c 100* parameter. This ensures that no efforts are undertaken to counter the OGM packet loss on a usually loss-less wired link (see Section 3.5.2).

### 5.2.2 Summary of measurement results

Due to the short time, the observations made about the three routing protocols do not reflect a profound and in-depth analysis. Nevertheless, a few interesting numbers about the performance and resource-consumption have been captured.

**The CPU-load** has been captured with the linux command `top` over a period of 70 seconds and with one measurement sample per second. The average CPU load per node is given in Table 1.

	olsrd 0.5.4pre	batmand-0.2	batmand-exp –graz-2007
avg. CPU load in percent:	0.51	3.85	1.81

Table 1: Average CPU consumption

The results show that CPU consumption of the parametrized experimental batmand is less than half of the batmand-0.2 implementation. This is probably mostly caused by the hidden secondary interfaces as described in Section 4.3. However, the numbers reveal that despite broadcasting OGMs twice, the overall CPU consumption can be decreased. The numbers also show that the CPU load of olsrd is much less than the load caused by batmand. It would be interesting to evaluate how these numbers scale in larger networks.

**The protocol traffic overhead** has been measured with a independent wireless card in monitor mode . The Graph illustrated in Figure 5 has been generated with wireshark [10]. It shows the protocol traffic generated by each of the three routing protocols (identified by its protocol port number) over a period of 200 seconds.

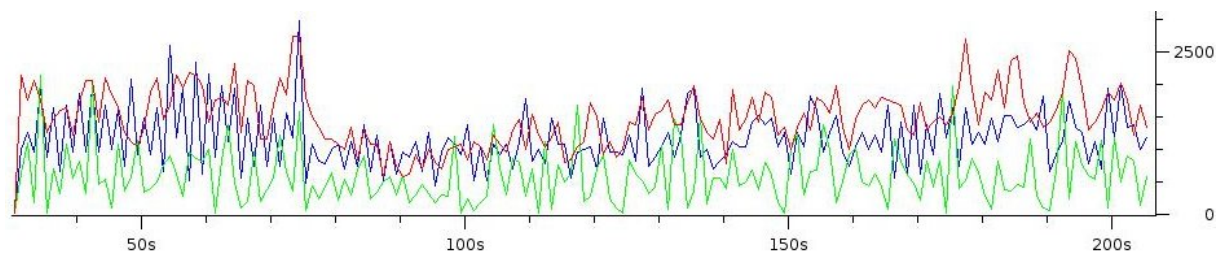


Figure 5: Graph showing the wireless IP traffic in bytes per second as generated by olsrd (green), batmand (red), and batmand-experimental (blue).

**The OGM flooding range** is another interesting aspect of the B.A.T.M.A.N protocol. The quality of the established end-to-end routes depends on the number of recently received OGMs that can be used for calculating representative path statistics. The more recent OGMs could be received from a distant node, the more representative should be the identified best path. If not even one OGM could be received, no end-to-path could be established.

Table 2 summarizes the number of received and accepted OGMs by node 01 from a subset of other (rather distant) nodes.

rcvd OGMs from node:	03	04	15	16	17	18	19	20
batmand-0.2	7	9	8	5	8	7	8	33
batmand-exp –graz-2007	48	34	26	16	22	24	29	56

Table 2: Snapshot of received and accepted OGMs by node 01 from other nodes

The table shows that the number of accepted OGMs using the batmand-experimental implementation is always greater than the number of OGMs received with the batmand-0.2 implementation.

**The path quality** has been tested by sending ping packets from one node to a subset of other nodes over a period of 40 minutes using smokeping [11]. The packet size of each ping request and reply was 1400 bytes. Table 3 illustrates the measured packet loss from one node to a subset of other nodes and Table 4 illustrates the average measured round trip time.

from node 02 to node:	05	06	08	10	12	15	18	total avg.
olsrd 0.5.4pre	14.84	5.74	0.70	2.77	65.63	27.87	26.58	20.59
batmand-0.2	28.21	12.53	0.59	2.68	35.52	29.36	59.45	24.05
batmand-exp -graz-2007	12.85	1.94	0.35	2.16	32.25	39.13	26.06	16.39

Table 3: Snapshot of average ping packet loss

from node 02 to node:	05	06	08	10	12	15	18	total avg.
olsrd 0.5.4pre	64.62	164.42	123.25	157.04	244.29	335.05	335.14	203.41
batmand-0.2	81.64	143.34	106.31	179.11	354.70	400.33	446.07	244.50
batmand-exp -graz-2007	123.66	18.00	12.04	114.58	365.72	188.09	339.69	165.97

Table 4: Snapshot of average ping round trip time

### 5.3 The -generation-III switch

This switch configures the behavior known from B.A.T.M.A.N generation III as implemented in batmand-0.2.

## 6 Tutorials

### 6.1 Setting up a basic testbed

This Section provides a brief summary how to setup a simple three-nodes mesh network using the batmand-experimental routing daemon. The tutorial describes all necessary (and some optionally) steps in an enumerated order. The following description assumes a basic setup as illustrated in Figure 6. Starting with two and working towards three nodes, each having one wireless lan and one wired lan interface. The interfaces and links between the nodes are discussed as needed.

1. The following libraries and kernel functionality must be available:

- wireless extensions, iwconfig, ifconfig, TUN and IP\_ADVANCED\_ROUTER support in the kernel and the iproute2 tool. Usually this is already installed on your

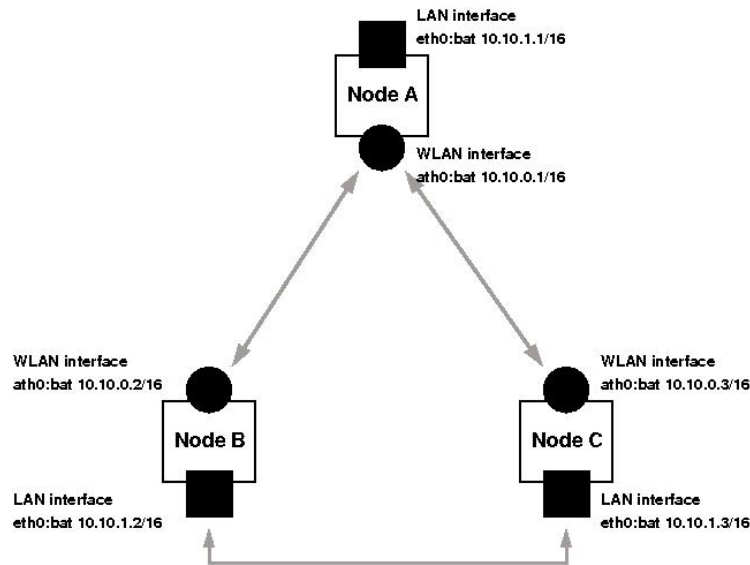


Figure 6: Default setup used for the tutorial. Grey lines indicate links that are enabled when needed.

system. You can simply test this by executing the ip or iwconfig binaries or grep for the loaded tun module as done in the following example:

```
root@ng1e:~# ip rule
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default

root@ng1e:~#root@ng1e:~# iwconfig
lo      no wireless extensions.

eth0    no wireless extensions.

ath0    IEEE 802.11g  ESSID:"batman-test"  Nickname:""
Mode:Ad-Hoc  Frequency:2.412 GHz  Cell: 02:CA:FF:EE:BA:BE
Bit Rate:0 kb/s  Tx-Power=15 dBm  Sensitivity=1/1
Retry:off  RTS thr:off  Fragment thr:off
Encryption key:off
Power Management:off
Link Quality=60/70  Signal level=-36 dBm  Noise level=-96 dBm
Rx invalid nwid:547  Rx invalid crypt:0  Rx invalid frag:0
Tx excessive retries:0  Invalid misc:0  Missed beacon:0

root@ng1e:~# lsmod | grep tun
tun          6496  0

root@ng1e:~#
```

- On a mipsel-openWrt-based system like freifunk libpthread, tun support, iwconfig,

and the batmand-experimental (installed at /usr/sbin/batmand) binaries can be installed with:

```
root@ng1e:~# ipkg install kmod-tun libpthread freifunk-openwrt-compat
root@ng1e:~# ipkg install http://downloads.open-mesh.net/batman/development/wrt-freifunk/batmand-exp_*-current_mipsel-wr-elf-32-lsb-dynamic.ipk
root@ng1e:~# modprobe kmod-tun
```

- For a i386-based system pre-compiled binaries can be downloaded and installed at /usr/sbin/batmand with:

```
root@ng1e:~# wget http://downloads.open-mesh.net/batman/development/i386/batmand-exp_*-current_i386-gc-elf-32-lsb-static.tgz
root@ng1e:~# tar xvzf batmand-exp_*-current_i386-gc-elf-32-lsb-static.tgz
root@ng1e:~# mv batmand-exp_*_i386-gc-elf-32-lsb-static /usr/sbin/batmand
```

- If you have a firewall it must be opened respectively. The easiest way to do so is to accept all IP packets with the example netmask of 10.10.0.0/16:

```
root@ng1e:~# iptables -I INPUT 1 -s 10.10.0.0/16 -j ACCEPT
root@ng1e:~# iptables -I INPUT 1 -d 10.10.0.0/16 -j ACCEPT
root@ng1e:~# iptables -I OUTPUT 1 -s 10.10.0.0/16 -j ACCEPT
root@ng1e:~# iptables -I OUTPUT 1 -d 10.10.0.0/16 -j ACCEPT
root@ng1e:~# iptables -I FORWARD 1 -s 10.10.0.0/16 -j ACCEPT
root@ng1e:~# iptables -I FORWARD 1 -d 10.10.0.0/16 -j ACCEPT
```

2. After having all the necessary tools available, the involved network interfaces on each node must be configured. Starting with only two nodes, each running batmand on one wireless interface. Assuming your wireless interfaces are labelled ath0. The wireless parameters of all nodes must be configured with the same essid (e.g. batman-test), channel (e.g 10), and mode (ad-hoc). Using iwconfig

```
root@ng1e:~# iwconfig ath0 mode ad-hoc essid batman-test channel 1
```

We are using a 10.10.0.0/16 network where all wireless interfaces shall become a 10.10.0.x address and the optionally wired interfaces get a 10.10.1.x address. Using alias interfaces, node A is configured like:

```
root@ng1e:~# ifconfig ath0:bat 10.10.0.1 netmask 255.255.0.0 broadcast 10.10.255.255
```

The wireless interface on node B is configured using:

```
root@ng1e:~# ifconfig ath0:bat 10.10.0.2 netmask 255.255.0.0 broadcast 10.10.255.255
```

Finally it should be possible to do a first connectivity test. Because the wlan interfaces of both nodes are in communication range of each other (I am assuming that the testbed has been setup in the same room or so) and have the same netmask they should be link local. A successful ping on node A to node B should verify this.

```
root@ng1e:~# ping 10.10.0.2
PING 10.10.0.2 (10.10.0.2): 56 data bytes
64 bytes from 10.10.0.2: icmp_seq=0 ttl=64 time=5.1 ms
64 bytes from 10.10.0.2: icmp_seq=1 ttl=64 time=2.2 ms
```

```
--- 10.10.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 2.2/3.6/5.1 ms
root@ng1e:~#
```

- Now everything is ready to start batmand. Simply execute the daemon on both nodes with the *-generation-III* directive and the name of the configured wlan interface as its only parameter. The output on node B will show something like:

```
root@ng2e:~# batmand --generation-III ath0:bat
WARNING: You are using the experimental batman branch!
...
Using interface ath0:bat with address 10.10.0.2 and broadcast address 10.10.255.255
root@ng2e:~#
```

- After executing the binaries they just fork to the background and the prompt is available again. Launching the *ps* command should show the process running in the background (The reason for seeing the process several times is a sideeffect of the threaded processing of the daemon. It can be ignored). The first feedback from the routing daemon may be obtained with the debug-level-one output. Therefore execute batmand a second time with the arguments *-c -d 1*. The argument *-c* tells batmand to connect to a running batman daemon. The argument *-d 1* tells the batmand to printout debug-level-one informations and *-b* tells batmand to provide this information only once. If *-b* is omitted, the debug information will be updated every second until the process is terminated using *ctrl-c*.

```
root@ng2e:~# batmand -c -d 1 -b
WARNING: You are using the experimental batman branch!
B.A.T.M.A.N. 0.3-exp rv686, MainIF/IP: ath0:bat 10.10.0.2, WindSize: 128, BLT: 2, OGI: 1000, UT: 0
Originator          viaIF          Router (brc rcvd lseq lvlld) [ viaIF RTQ RQ TQ].. alternati
10.10.0.1           ath0:bat      10.10.0.1 (112 112 785 0) [ ath0:bat 112 127 112]
root@ng2e:~#
```

The debug-level-one output reveals a number of informations. The first line shows the batmand branch, the revision, the label and ip address of the first interface parameter, the window size, the bidirectional link timeout, the originator interval and the amount of time passed since this process was started.

The next line shows the headline of a table. In our case the table has only one entry which indicates that node B has learned about the existence of node A. The line starts with the IP of the other batman node, followed by the interface used for routing towards that node, and the best-ranking neighbor used as gateway towards that node. In this case the best-ranking neighbor towards node A is already node A. For more informations about the informations revealed with debug-level-one see Section 2.

- You can use the *iproute2* tool to investigate the changes batmand has caused to the networking stack.

```
root@ng2e:~# ip rule
0:      from all lookup local
6600:   from all to 10.10.0.0/16 lookup 66
6699:   from all lookup 65
```

```
6700:   from all to 10.10.0.0/16 lookup 67
32766:  from all lookup main
32767:  from all lookup default
root@ng2e:~# ip route list table 65
root@ng2e:~# ip route list table 66
10.10.0.1 dev ath0 proto static scope link src 10.10.0.2
root@ng2e:~# ip route list table 67
unreachable default proto static
root@ng2e:~#
```

Comparing to the first time we started the *ip rule* command (when we checked if the *iproute2* tool is properly working), the output of this command shows three new lines. These new lines are priority rules. They are telling the networking stack to look out for routes from and towards certain IP ranges in dedicated routing tables – namely table 65, 66, 67, and 68. Table 65 is used for HNA routes, 66 for host routes, 67 for unreachable routes. Later on in the tutorial, also routing table 68 will be used for the default-tunnel route to forward internet traffic between client and GW nodes. Using the command *ip route list table 66* the content of table 66 can be further investigated.

For example the *ip rule* command reveals that the ip stack will look out for all kinds of target-ip addresses in table 65. This table is used for network announcements as we will see soon.

6. The batman daemons can be killed with *killall batmand*. Afterwards all configurations made to the networking stack should be cleaned up again.
7. Next, we introduce one additional link and involve the node C into the mesh. Therefore we configure the lan interfaces on the three nodes and plug an ethernet cable between the lan interface of node B and node C. Assuming the lan interfaces are labelled *eth0*, one of the following lines must be executed on each corresponding node.

```
root@ng1e:~# ifconfig eth0:bat 10.10.1.1 netmask 255.255.0.0 broadcast 10.10.255.255
```

```
root@ng2e:~# ifconfig eth0:bat 10.10.1.2 netmask 255.255.0.0 broadcast 10.10.255.255
```

```
root@ng3e:~# ifconfig eth0:bat 10.10.1.3 netmask 255.255.0.0 broadcast 10.10.255.255
```

Be aware, that now node A and B have two batman interfaces (the lan and the wlan interface) but node C has only the lan interface configured.

8. This time the batman daemon on node A and B must be executed with two trailing interface arguments (*ath0:bat eth0:bat*) where on node C only the single lan interface argument *eth0:bat* must be applied. On node B the procedure will look like this:

```
root@ng2e:~# batmand --generation-III ath0:bat eth0:bat
```

```
WARNING: You are using the experimental batman branch!
```

```
...
```

```
Using interface ath0:bat with address 10.10.0.2 and broadcast address 10.10.255.255
```

Using interface eth0:bat with address 10.10.1.2 and broadcast address 10.10.255.255

```
root@ng2e:~# batmand -c -d 1 -b
WARNING: You are using the experimental batman branch!
B.A.T.M.A.N. 0.3-exp rv686, MainIF/IP: ath0:bat 10.10.0.2, WindSize: 128, BLT: 2, OGI: 1000, UT: 0
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternati
10.10.1.3       eth0:bat   10.10.1.3 ( 70 70 178 0) [ eth0:bat 70 70 128]
10.10.0.1       ath0:bat   10.10.0.1 (100 100 3044 45) [ ath0:bat 83 128 83]
10.10.1.1       ath0:bat   10.10.0.1 ( 99 99 3041 45)
root@ng2e:~#
```

The debug-level-one output on node B now also shows the outgoing interface and best-ranking neighbor towards the lan interface of node A and the wlan interface of node C. The debug-level-one output on node C should show:

```
root@ng3e:~# batmand -c -d 1 -b
WARNING: You are using the experimental batman branch!
B.A.T.M.A.N. 0.3-exp rv686, MainIF/IP: eth0:bat 10.10.1.3, WindSize: 128, BLT: 2, OGI: 1000, UT: 0
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternati
10.10.0.2       eth0:bat   10.10.1.2 (128 128 3581 0)
10.10.1.2       eth0:bat   10.10.1.2 (128 128 3583 0) [ eth0:bat 128 128 128]
10.10.0.1       eth0:bat   10.10.1.2 ( 24 24 3465 52)
10.10.1.1       eth0:bat   10.10.1.2 ( 24 24 3461 52)
root@ng3e:~#
```

9. So fare (despite the different looking debug-level-one output) everything behaved just like with batmand-0.2 Now it is time to make the first experiments with some batmand-experimental concepts. Therefore we once again *killall batmand* processes and relaunch them with the *-bmx-defaults* argument as shown for node A:

```
root@ngle:~# batmand -c -d 1 -b
WARNING: You are using the experimental batman branch!
B.A.T.M.A.N. 0.3-exp rv687, MainIF/IP: ath0:bat 10.10.0.1, WindSize: 128, BLT: 2, OGI: 1000, UT: 0
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternati
10.10.0.2       ath0:bat   10.10.0.2 ( 38 38 63 0) [ ath0:bat 38 39 124]
10.10.1.2       ath0:bat   10.10.0.2 ( 39 39 64 0)
10.10.1.3       ath0:bat   10.10.0.2 ( 37 37 47 0)
root@ngle:~# killall batmand
root@ngle:~#
root@ngle:~# batmand --bmx-defaults ath0:bat eth0.0:bat
WARNING: You are using the experimental batman branch!
Applying bmx-defaults !
This parametrization expects the first given interface argument to be a wireless interface !
Short option: o with argument: 1500
Long option: bi-link-timeout with argument: 20
Long option: window-size with argument: 100
Long option: gw-change-hysteresis with argument: 2
Long option: dups-ttl with argument: 2
Long option: dups-rate with argument: 100
Long option: dups-ttl-degradation with argument: 2
```

```

Long option: send-clones with argument: 200
Long option: asymmetric-weight with argument: 100
Long option: asymmetric-exp with argument: 1
Long option: re-brc-delay with argument: 35
Using interface ath0:bat with address 10.10.0.1 and broadcast address 10.10.255.255
Using interface eth0:bat with address 10.10.1.1 and broadcast address 10.10.255.255
Interface eth0.0:bat specific option: /a
Interface eth0.0:bat specific option: /i
Interface eth0.0:bat specific option: /t 1
Interface eth0.0:bat specific option: /c 100
root@ngle:~#
root@ngle:~# echo dont forget to start batmand --bmx-defaults... also on the other nodes before co
dont forget to start batmand --bmx-defaults... also on the other nodes before continueing :)
root@ngle:~#
root@ngle:~# batmand -c -d 1 -b
WARNING: You are using the experimental batman branch!
B.A.T.M.A.N. 0.3-exp rv687, MainIF/IP: ath0:bat 10.10.0.1, WindSize: 100, BLT: 20, OGI: 1500, UT:
Originator      viaIF      Router (brc rcvd lseq lvld) [ viaIF RTQ RQ TQ].. alternati
10.10.0.2       ath0:bat   10.10.0.2 ( 4 5 739 0) [ ath0:bat 5 6 83]
10.10.1.3       ath0:bat   10.10.0.2 ( 3 4 743 1)
root@ngle:~#

```

The parametrization applied with the `-bmx-defaults` switch is described in Section 5.1. In newer versions, this switch can also be omitted because it is the default parametrization of `batmand-experimental`. One of the activated features is that all non-primary interfaces do not announce their existence beyond their local neighbor horizon. Instead, the IP address of secondary and further interfaces are announced by host-network announcement (HNA) messages. The HNA messages are conveyed with the OGMs of the primary interface. This reduces the protocol-traffic overhead and the CPU load of the daemon. Compared to the debug-level-one output before killing `batmand`, the output now does not show the originator IP of 10.10.1.2 anymore. This is because related OGMs are not propagated beyond the direct-link neighborhood of node Bs secondary interface and Node A is not connected to that link. Nonetheless, node Bs secondary interface IP is still reachable because it has been announced using HNA messages. Thereby, node A has learned that all packets for 10.10.1.2 must be forwarded towards the IP of node Bs primary interface. Therefore, `batman` uses the routing table 65 (which has been unused in previous examples).

```

root@ngle:~# ip route list table 65
throw 10.10.1.1 proto static
10.10.1.2 via 10.10.0.2 dev ath0 proto static
root@ngle:~#
root@ngle:~# ping -c 1 10.10.1.2
PING 10.10.1.2 (10.10.1.2): 56 data bytes
64 bytes from 10.10.1.2: icmp_seq=0 ttl=64 time=6.8 ms

--- 10.10.1.2 ping statistics ---

```

```
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 6.8/6.8/6.8 ms
root@ng1e:~#
```

## 6.2 Setting up internet gateway and client nodes

This Section follows the track of the previous tutorial but focuses on how to enable internet connectivity for the nodes in the mesh. Therefore, the involved nodes are separated into two categories. The category of nodes which are offering internet connectivity is referred as gateway (GW) nodes and the other category is referred as client nodes. Client nodes are looking for internet connectivity. Because of the few number of node we have, there will be only one GW node in this example, namely node A. This GW node needs a real internet connection which can be used to forward traffic.

The following steps describe step by step how to configure one GW and two client nodes so that the GW node announces its internet connectivity and that the clients automatically detect the existence of an available GW and setup tunnels to the GW to forward their internet traffic.

1. Starting with the GW node, we use the lan interface of node A for accessing a dhcp server, obtain an IP address and a default route. But first, the running batmand should be terminated and the `eth0:bat` alias interface should be disabled. In the following capture, the program udhcpc is used to contact the dhcp server, any other dhcp client programm should be fine as well. Afterwards, a ping to the famous open-mesh.net web site is used to verify that the internet connectivity from the GW node is up and running.

```
root@ng1e:~# killall batmand
root@ng1e:~# ifconfig eth0:bat down
root@ng1e:~#
root@ng1e:~# udhcpc -i eth0
udhcpc (v1.4.2) started
Sending discover...
Sending select for 192.168.3.35...
Lease of 192.168.3.35 obtained, lease time 60000
adding router 192.168.3.6
route: SIOC[ADD|DEL]RT: File exists
deleting old routes
route: resolving 104.130.30.205;
adding dns 213.191.74.11
root@ng1e:~#
root@ng1e:~# ping open-mesh.net
PING open-mesh.net (88.198.145.69): 56 data bytes
64 bytes from 88.198.145.69: icmp_seq=0 ttl=53 time=31.2 ms
64 bytes from 88.198.145.69: icmp_seq=1 ttl=53 time=29.5 ms

--- open-mesh.net ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 29.5/30.3/31.2 ms
root@ng1e:~#
```

- Next, node A must be configured to share its internet connection with other client nodes in the mesh. Because all the nodes in the internet do not know anything about the IP addresses of the client nodes in our mesh there must be a network address translation (NAT) on the lan interface of node A. On a linux system, NAT is achieved with iptables (which is hopefully installed on your system) and the easiest way to activate NAT is to configure a MASQUERADE rule for the lan interface of node A.

```

root@ng1e:~# iptables -t nat -I POSTROUTING 1 -o eth0 -j MASQUERADE
root@ng1e:~# iptables -L -t nat -n -vv
Chain PREROUTING (policy ACCEPT 5408 packets, 1635K bytes)
  pkts bytes target     prot opt in     out     source         destination
Chain POSTROUTING (policy ACCEPT 39 packets, 4554 bytes)
  pkts bytes target     prot opt in     out     source         destination
    0    0 MASQUERADE  all  --  *      eth0    0.0.0.0/0      0.0.0.0/0
Chain OUTPUT (policy ACCEPT 63 packets, 8014 bytes)
  pkts bytes target     prot opt in     out     source         destination

```

- Relaunching batmand on node A with the *-g bandwidth* switch causes the daemon to announce its GW services to other nodes in the mesh. The daemon will create a tunnel devices labelled as batX for each interface participating in the mesh. The tunnel devices are used for forwarding internet traffic between the GW node and the connected client nodes. (In order to use the tunnel functionality of the linux kernel it is mandatory to have the tun module loaded. Check this for example with the command *lsmod* . If the command does not list a line showing tun then load it with *modprobe tun* or *insmod tun* on a openWrt based system.)

```

root@ng1e:~# batmand --bmx-default -g 5000 ath0:bat

```

- Finally, batmand must be restarted on the client nodes with the *-r routing-class* directive. The routing class must be a value between 1 and 3 to define a policy for the GW selection. Routing class 1 causes the client to prefer fast connections (focusing on the announced bandwidth of the GW) and routing class 2 prefers stable connections. Routing class 3 also prefers stable connections but changes the currently selected GW if another GW shows up with a more stable connection between the GW and the client. For the setup of this tutorial the routing class does not matter because there is only one available GW. So lets try routing class 3. After restarting batmand the daemon waits some time to gather some statistics about the path qualities to the available internet GWs. Use debug-level two (*-c -d 2 -b*) to monitor the available GW nodes and see when the client mode has decided for one of them. Et voila, now also the clients can access internet.

```

root@ng3e:~# killall batmand
root@ng3e:~# batmand --bmx-defaults -r 3 eth0.0:bat
WARNING: You are using the experimental batman branch!

```

```
Applying bmx-defaults !
...
Using interface eth0.0:bat with address 10.10.1.3 and broadcast address 10.10.255.255
root@ng3e:~#
root@ng3e:~# batmand -c -d 2 -b
WARNING: You are using the experimental batman branch!
      Gateway          Router (#/100)
      10.10.0.1        10.10.1.2 ( 13), gw_class 49 - 4MBit/1024KBit, reliability: 0
root@ng3e:~#
root@ng3e:~# echo wait some time until debug-level two shows that the client has connected
echo wait some time until debug-level two shows that the client has connected
root@ng3e:~#
root@ng3e:~# batmand -c -d 2 -b
WARNING: You are using the experimental batman branch!
      Gateway          Router (#/100)
=> 10.10.0.1          10.10.1.2 ( 98), gw_class 49 - 4MBit/1024KBit, reliability: 0
root@ng3e:~#
root@ng3e:~# ping open-mesh.net
PING open-mesh.net (88.198.145.69): 56 data bytes
64 bytes from 88.198.145.69: icmp_seq=0 ttl=52 time=34.3 ms
64 bytes from 88.198.145.69: icmp_seq=1 ttl=52 time=31.0 ms
64 bytes from 88.198.145.69: icmp_seq=2 ttl=52 time=31.2 ms

--- open-mesh.net ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 31.0/32.1/34.3 ms
root@ng3e:~#
```

The first packet often takes more time or even gets lost because the client node needs to obtain an IP address from the GW node to configure its tunnel correctly (Be aware that the first packet is usually the DNS request to resolve the requested domain name. Also ensure that `/etc/resolv.conf` contains a valid nameserver). Often, there are problems with the firewall configuration on the client as well as on the GW node resulting in packets to simply get dropped. This can become quite tricky to debug, especially because the client node tries to observe the traffic send via the tunnel and disconnects from the currently selected GW if no packets do return via the tunnel. This is feature is referred as unresponsive GW check and it can be quite annoying when searching for the actual reasons for the broken connections. Therefore, there is also a `-no-unresp-gw-check` switch to disable the GW monitoring and automatic disconnection.

Simply flushing the firewall might be a good idea as well (using `iptables -F`; `iptables -t nat -F`) but remember to reassign the MASQUERADING rule afterwards. Additional tools to debug a broken internet connectivity are of course `tcpdump` and `debug level 3`.

- Probably the most common configuration is to enable the client nodes of a mesh for offering internet connectivity via `dhcp` to ordinary computers. This way, any computer supporting the `dhcp-client` protocol can use the infrastructure of the mesh without knowing anything about the internals of the involved protocols. Because of the limited number of nodes and interfaces assumed for this tutorial we once again have to

reconfigure the interfaces of the client nodes.

The following examples shows this for client node C. First, the running batman daemon must be terminated and the lan interface must be reconfigured with a static, side-local, and yet unused netmask and IP address (it is important to do this before restarting the batman daemon because otherwise batmand will falsley detect your interface configuration and catch all packets towards your side-local lan addresses). If there is a dhcp client or server already running in the background it should be killed. Then, any kind of dhcp-server daemon must be installed, configured, and started to offer dhcp service on the new configured lan interface. I've choosen the busybox version of udhcpd because it was shipped with the openWrt distribution used for this tutorial (udhcpd also needs a configuration file as indicated):

```
root@ng3e:~# killall batmand
root@ng3e:~#
root@ng3e:~# ifconfig eth0:bat down
root@ng3e:~# killall udhcp
root@ng3e:~# ifconfig eth0 192.168.123.1
root@ng3e:~# cat /etc/udhcpd.conf
start          192.168.123.2
end            192.168.123.12
max_leases    10
opt           router 192.168.123.1
opt           subnet 255.255.255.0
opt           lease 43200
opt dns 141.1.1.1
interface     eth0
lease_file    /var/run/udhcpd.leases
root@ng3e:~# killall udhcpd
root@ng3e:~# udhcpd /etc/udhcpd.conf
udhcpd (v1.4.2) started
udhcpd: cannot open /var/run/udhcpd.leases for reading
root@ng3e:~#
```

For node C, the wireless interface must be configured for accessing the mesh network. There need to be a second network address translation (NAT) on the tunnel interface of the client nodes because the GW node, at the other end of the tunnel, does not know anything about the IP addresses leased out to the dhcp-clients (the ordinary computers) of node C.

Optionally you may do SNAT on the batman interfaces to allow the ordinary-dhcp clients to access the nodes in the mesh.<sup>3</sup>.

```
root@ng3e:~# iwconfig ath0 essid batman-test channel 1
root@ng3e:~# ifconfig ath0:bat 10.10.0.3 netmask 255.255.0.0 broadcast 10.10.255.255
root@ng3e:~#
root@ng3e:~# iptables -t nat -A POSTROUTING -o bat0 -j MASQUERADE
root@ng3e:~# #iptables -t nat -A POSTROUTING -o ath0 \
```

<sup>3</sup>Then the -o, -s, -d, and -to arguments must be customized for each node and interface. If batmand is running on several interfaces you need one SNAT rule per interface.

```
-s 192.168.123.0/24 -d 10.10.0.0/16 -j SNAT --to 10.10.0.3 #optional, needs customization
root@ng3e:~#
root@ng3e:~# batmand --bmx-defaults -r 3 ath0:bat
WARNING: You are using the experimental batman branch!
Applying bmx-defaults !
...
root@ng3e:~#
```

That's it! Now it should be possible to connect the network interface of an ordinary computer system with the lan interface of node C and lease an IP address for accessing the internet. The following capture shows this procedure from my notebook. Using the `-R` directive for ping it is possible to track the path taken by the packets. Unfortunately the `-R` directive is not implemented in many embedded ping implementations. If you want to have a full working ping on your embedded device you may check [13] for a precompiled binary that fits for your embedded devices.

```
smart ~ # killall dhcpcd
smart ~ #
smart ~ # dhcpcd eth0
smart ~ # ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:90:F5:55:5B:EB
          inet addr:192.168.123.2  Bcast:192.168.123.255  Mask:255.255.255.0
          inet6 addr: fe80::290:f5ff:fe55:5beb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:269068 errors:0 dropped:0 overruns:0 frame:0
          TX packets:255602 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:60394298 (57.5 Mb)  TX bytes:27671725 (26.3 Mb)
          Interrupt:16

smart ~ # ping -n -R open-mesh.net
PING open-mesh.net (88.198.145.69) 56(124) bytes of data.
64 bytes from 88.198.145.69: icmp_seq=1 ttl=49 time=77.6 ms
RR:      192.168.123.2
         169.254.0.1
         192.168.2.3
...

64 bytes from 88.198.145.69: icmp_seq=2 ttl=49 time=60.5 ms      (same route)
64 bytes from 88.198.145.69: icmp_seq=3 ttl=49 time=89.4 ms      (same route)

--- open-mesh.net ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 60.548/75.875/89.447/11.867 ms

smart ~ # ping -R 10.10.0.1
PING 10.10.0.1 (10.10.0.1) 56(124) bytes of data.
64 bytes from 10.10.0.1: icmp_seq=1 ttl=63 time=2.81 ms
```

```
RR:      192.168.123.2
         10.10.0.3
         10.10.0.1
         10.10.0.1
         192.168.123.1
         192.168.123.2
```

```
64 bytes from 10.10.0.1: icmp_seq=2 ttl=63 time=2.26 ms (same route)
```

```
64 bytes from 10.10.0.1: icmp_seq=3 ttl=63 time=2.40 ms (same route)
```

```
--- 10.10.0.1 ping statistics ---
```

```
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
```

```
rtt min/avg/max/mdev = 2.264/2.494/2.811/0.238 ms
```

```
root@ng1e:~# killall udhcpc udhcpd batmand bmxd
root@ng1e:~# udhcpc -i eth0.0
root@ng1e:~# iptables -F
root@ng1e:~# iptables -t mangle -F
root@ng1e:~# iptables -t nat -F
root@ng1e:~# iptables -t nat -A POSTROUTING -o eth0.0 -j MASQUERADE # eth0.0 is default
root@ng1e:~# ifconfig ath0      104.130.30.201 netmask 255.0.0.0
root@ng1e:~# ifconfig ath0:bat 105.130.30.201 netmask 255.0.0.0
root@ng1e:~# ifconfig ath0:bmx 103.130.30.201 netmask 255.0.0.0
root@ng1e:~# iwconfig ath0 channel 10 essid olsr.freifunk.net ap 02:CA:FF:EE:BA:BE
root@ng1e:~# batmand -g 5000 ath0:bat
```

```
root@ng2e:~# killall udhcpc udhcpd batmand bmxd
root@ng2e:~# iptables -F
root@ng2e:~# iptables -t mangle -F
root@ng2e:~# iptables -t nat -F
root@ng2e:~# ifconfig eth0.0    192.168.123.1 netmask 255.255.255.0
root@ng2e:~# vi /etc/udhcpd.conf # control interface, router, and ip range
root@ng2e:~# udhcpd /etc/udhcpd.conf
```

```
root@ng2e:~# ifconfig ath0      104.130.30.202 netmask 255.0.0.0
root@ng2e:~# ifconfig ath0:bat 105.130.30.202 netmask 255.0.0.0
root@ng2e:~# ifconfig ath0:bmx 103.130.30.202 netmask 255.0.0.0
root@ng2e:~# ifconfig eth0.1    104.130.30.212 netmask 255.0.0.0
root@ng2e:~# ifconfig eth0.1:bat 105.130.30.212 netmask 255.0.0.0
root@ng2e:~# ifconfig eth0.1:bmx 103.130.30.212 netmask 255.0.0.0
root@ng2e:~# iwconfig ath0 channel 10 essid olsr.freifunk.net ap 02:CA:FF:EE:BA:BE
#make shure all interfaces are configured before batmand starts (even eth0.0)
root@ng2e:~# batmand -r 3 ath0:bat eth0.1:bat
```

```
root@ng2e:~# iptables -t nat -A POSTROUTING -o bat0 -j MASQUERADE
root@ng2e:~# iptables -t nat -A POSTROUTING -o ath0 -s 192.168.123.0/24 -d 103.0.0.0/8 -j SNAT --to 103.0.0.0
root@ng2e:~# iptables -t nat -A POSTROUTING -o ath0 -s 192.168.123.0/24 -d 104.0.0.0/8 -j SNAT --to 104.0.0.0
root@ng2e:~# iptables -t nat -A POSTROUTING -o ath0 -s 192.168.123.0/24 -d 105.0.0.0/8 -j SNAT --to 105.0.0.0
root@ng2e:~# iptables -t nat -A POSTROUTING -o eth0.1 -s 192.168.123.0/24 -d 103.0.0.0/8 -j SNAT --to 103.0.0.0
root@ng2e:~# iptables -t nat -A POSTROUTING -o eth0.1 -s 192.168.123.0/24 -d 104.0.0.0/8 -j SNAT --to 104.0.0.0
root@ng2e:~# iptables -t nat -A POSTROUTING -o eth0.1 -s 192.168.123.0/24 -d 105.0.0.0/8 -j SNAT --to 105.0.0.0
```



```
# now you may connect a dhcp-client to eth0.0 of node B and surfe the web as well as access all 103/104
# do the same for node C
```

```
root@ng1e:~# bmxd --base-port 14305 --rt-table-offset 165 --prio-rules-offset 16600 --gw-tunnel-network
```

```
root@ng2e:~# bmxd --base-port 14305 --rt-table-offset 165 --prio-rules-offset 16600 --window-size 10 atl
```

```
root@ng3e:~# bmxd --base-port 14305 --rt-table-offset 165 --prio-rules-offset 16600 --window-size 10 atl
```

```
root@ng3e:~# bmxd -c --base-port 14305 -d 1 -b
```

```
WARNING: You are using the experimental batman branch!
```

```
--base-port 14305 \
```

```
BatMan-eXperimental 0.3-alpha rv751, IF: ath0:bmx 103.130.30.203, WindSize: 10, BLT: 20, OGI: 1500, UT:
```

```
Neighbor      outgoingIF      bestLink (brc rcvd lseq lvl) [ viaIF RTQ RQ TQ]..
```

```
103.130.30.201 ath0:bmx 103.130.30.201 ( 10 10 15 0) [ ath0:bmx 10 10 10]
```

```
103.130.30.212 eth0.1:bmx 103.130.30.212 ( 10 10 18 1) [eth0.1:bmx 10 10 10]
```

```
103.130.30.202 ath0:bmx 103.130.30.202 ( 10 10 18 1) [ ath0:bmx 10 10 10]
```

```
Originator      outgoingIF      bestNextHop (brc rcvd lseq lvl) alternative next hops...
```

```
103.130.30.201 ath0:bmx 103.130.30.201 ( 10 10 15 0) 103.130.30.202 ( 9) 103.130.30.212 (
```

```
103.130.30.212 eth0.1:bmx 103.130.30.212 ( 10 10 18 1)
```

```
103.130.30.202 ath0:bmx 103.130.30.202 ( 10 10 18 1) 103.130.30.212 ( 10) 103.130.30.201 (
```

```
Originator      HNAs...
```

```
103.130.30.202 103.130.30.212/32
```

```
root@ng3e:~#
```

```
root@ng3e:~# bmxd -c -d 1 -b
```

```
WARNING: You are using the experimental batman branch!
```

```
BatMan-eXperimental 0.3-alpha rv751, IF: ath0:bat 105.130.30.203, WindSize: 100, BLT: 20, OGI: 1500, UT:
```

```
Neighbor      outgoingIF      bestLink (brc rcvd lseq lvl) [ viaIF RTQ RQ TQ]..
```

```
105.130.30.212 eth0.1:bat 105.130.30.212 (100 100 1045 1) [eth0.1:bat 100 100 100]
```

```
105.130.30.202 eth0.1:bat 105.130.30.212 (100 100 1045 0) [ ath0:bat 97 99 97]
```

```
105.130.30.201 ath0:bat 105.130.30.201 ( 99 100 2816 0) [ ath0:bat 99 98 100]
```

```
Originator      outgoingIF      bestNextHop (brc rcvd lseq lvl) alternative next hops...
```

```
105.130.30.212 eth0.1:bat 105.130.30.212 (100 100 1045 1)
```

```
105.130.30.202 eth0.1:bat 105.130.30.212 (100 100 1045 0) 105.130.30.202 ( 98) 105.130.30.201 (
```

```
105.130.30.201 ath0:bat 105.130.30.201 ( 99 100 2816 0) 105.130.30.212 ( 97) 105.130.30.202 (
```

```
Originator      HNAs...
```

```
105.130.30.202 105.130.30.212/32
```

```
root@ng3e:~#
```

Do some extensive testing now. All nodes can be contacted using the 103/105 netmask.

```
root@ng5e:~# real-ping -f -c 100 -s 1300 105.130.30.201
```

```
PING 105.130.30.201 (105.130.30.201) 1300(1328) bytes of data.
```

```
--- 105.130.30.201 ping statistics ---
```



```
100 packets transmitted, 100 received, 0% packet loss, time 765ms
rtt min/avg/max/mdev = 5.494/6.703/16.865/2.146 ms, pipe 2, ipg/ewma 7.729/6.875 ms
```

```
root@ng5e:~# real-ping -f -c 100 -s 1300 103.130.30.201
PING 103.130.30.201 (103.130.30.201) 1300(1328) bytes of data.
--- 103.130.30.201 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 913ms
rtt min/avg/max/mdev = 6.475/8.378/20.180/1.881 ms, pipe 2, ipg/ewma 9.228/8.751 ms
root@ng5e:~#
```

## 7 Acknowledgments

The B.A.T.M.A.N. mesh routing protocol is currently developed and documented by:

Andreas Langer (a.langer-at-q-dsl.de)  
Axel Neumann (axel-at-open-mesh.net)  
Corinna 'Elektra' Aichele (onelektra-at-gmx.net)  
Ludger Schmudde (lui-at-schmudde.com)  
Marek Lindner (lindner\_marek-at-yahoo.de)  
Simon Wunderlich (siwu-at-hrz.tu-chemnitz.de)  
Stefan Sperling (stsp-at-stsp.in-berlin.de)  
Wesley Tsai (wesleyboy-at-gmail.com)

## References

- [1] Wesley Tsai, *B.A.T.M.A.N Daemon HowTo*, August 8, 2007, [http://open-mesh.net/batman/doc/batmand\\_howto.pdf](http://open-mesh.net/batman/doc/batmand_howto.pdf)
- [2] Wesley Tsai, *BATMAND manpage*, [http://downloads.open-mesh.net/batman/development/sources/batmand\\_\\*-current\\_sources/man/batmand.8.html](http://downloads.open-mesh.net/batman/development/sources/batmand_*-current_sources/man/batmand.8.html)
- [3] batmand subversion repository, <https://dev.open-mesh.net/svn/batman/trunk/batman/INSTALL>
- [4] batmand-experimental download URL, [http://downloads.open-mesh.net/batman/development/sources/batmand-exp\\_\\*-current\\_sources.tgz](http://downloads.open-mesh.net/batman/development/sources/batmand-exp_*-current_sources.tgz)
- [5] Axel Neumann, Corinna 'Elektra' Aichele, Marek Lindner, *B.A.T.M.A.N Status Report*, June 28, 2007, <http://open-mesh.net/batman/doc/batman-status.pdf>
- [6] *B.A.T.M.A.N Protocol Specification*, online wiki, <http://open-mesh.net/batman/doc/specification>

- [7] *IEEE 802.11-1999, 1999 Edition (R2003): "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, June, 12 2003, <http://standards.ieee.org/getieee802/802.11.html>
- [8] *IEEE 802.11a-1999(R2003): "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, High-speed Physical Layer in the 5GHz Band* , June, 12 2003, <http://standards.ieee.org/getieee802/download/802.11a-1999.pdf>
- [9] <http://open-mesh.net/batman/experience/WCWGraz2007>
- [10] <http://www.wireshark.org/>
- [11] <http://oss.oetiker.ch/smokeping/>
- [12] <http://openwrt.org/>
- [13] <http://downloads.open-mesh.net/misc/handy-tools/>

## Licence

Copyright (C) 2007 Axel Neumann (axel-at-open-mesh.net)

Last updated December 10, 2007

Licensed under the Creative Commons BY-NC-SA-2.0

<http://creativecommons.org/licenses/by-nc-sa/2.0/>