B.A.T.M.A.N Status Report

Axel Neumann, Corinna “Elektra” Aichele, Marek Lindner

June 28, 2007

Abstract

This report documents the current status of the development and implementation of the B.A.T.M.A.N (better approach to mobile ad-hoc networking) routing protocol. B.A.T.M.A.N uses a simple and robust algorithm for establishing multi-hop routes in mobile ad-hoc networks. It ensures highly adaptive and loop-free routing while causing only low processing and traffic cost.

Contents

1 Introduction

In this report we present the current status of the development and implementation of the B.A.T.M.A.N (better approach to mobile ad-hoc networking) routing protocol[1].

[1]B.A.T.M.A.N is available at http://open-mesh.net/batman

The problem with classical routing protocols is that they are typically not well suited for wireless ad-hoc networks. This is because such networks are unstructured, dynamically change their topology, and are based on an inherently unreliable medium.

OLSR, the currently most employed protocol for such scenarios, has undergone a number of changes from its original specification in order to deal with the challenges imposed by city-wide wireless mesh networks. While some of its components proved to be unsuitable in practice (like MPR and Hysterese) new mechanisms have been added (like Fish-eye and ETX). However, due to the constant growth of existing community mesh networks and because of the inherent requirement of a link-state algorithm to recalculate the whole topology-graph (a particularly challenging task for the limited capabilities of embedded router HW), the limits of this algorithm have become a challenge. Recalculating the whole topology graph once in an actual mesh with 450 nodes takes several seconds on a small embedded CPU.

The approach of the B.A.T.M.A.N algorithm is to divide the knowledge about the best end-to-end paths between nodes in the mesh to all participating nodes. Each node perceives and maintains only the information about the best next hop towards all other nodes. Thereby the need for a global knowledge about local topology changes becomes unnecessary. Additionally, an event-based but timeless[2] flooding mechanism prevents the accruement of contradicting topology information (the usual reason for the existence of routing loops) and limits the amount of topology messages flooding the mesh (thus avoiding overly overhead of control-traffic). The algorithm is designed to deal with networks that are based on unreliable links.

The protocol algorithm of B.A.T.M.A.N can be described (simplified) as follows. Each node transmits broadcast messages (we call them originator messages or OGMs) to inform the neighboring nodes about it's existence. These neighbors are re-broadcasting the OGMs according to specific rules to inform their neighbors about the existence of the original initiator of this message and so on and so forth. Thus the network is flooded with originator messages. OGMs are small, the typical raw packet size is 52 byte including IP and UDP overhead. OGMs contain at least the address of the originator, the address of the node transmitting the packet, a TTL and a sequence number. OGMs that follow a path where the quality of wireless links is poor or saturated will suffer from packetloss or delay on their way through the mesh. Therefore OGMs that travel on good routes will propagate faster and more reliable. In order to tell if a OGM has been received once or more than once it contains a sequence number, given by the originator of the OGM. Each node re-broadcasts each received OGM at most once and only those received from the neighbor which has been identified as the currently best next hop (best ranking neighbor) towards the original initiator of the OGM. This way the OGMs are flooded selectively through the mesh and inform the receiving nodes about other node's existence. A node X will learn about the existence of a node Y in the distance by receiving it's OGMs, when OGMs of node Y are rebroadcasted by it's single hop neighbors. If node X has more than one neighbor, it can tell by the number of originator messages it receives quicker and more reliable via one of its single hop neighbors, which neighbor it has to choose to send data to the distant node. The algorithm then selects this neighbor as the currently best next hop to the originator of the message and configures its routing table respectively.

2 Evolution

In contrast to many other protocols, B.A.T.M.A.N has undergone an extensive practical implementation and testing phase from the beginning of the development. Just like in natural evolution, many improvements have been developed in order to adapt the core algorithm to (the frequent

[2]timeless in the sense that B.A.T.M.A.N never schedules nor timeouts topology information for optimising it's routing decisions

discoveries of) new and challenging real-life problems. Nevertheless a number of promising performance simulations have been performed now.

In 2005 Corinna 'Elektra' Aichele and Thomas Lopatic reconsidered the widespread mesh routing protocol OLSR, with all its complexity and all the modifications (mostly truncations) that have been applied to the original algorithm in order to make it work to some level in wireless real-life setups and draw the idea of a new very simple mesh-routing approach. Since then, the core algorithm as well as its implementation have undergone a number of evolutionary changes.

In the following we have tried do differentiate this evolution in two categories. Section 2.1 gives a summary about the changes in the core flooding and routing algorithm. Section 2.2 describes the evolution of our implementation in terms of capabilities that have been integrated to make B.A.T.M.A.N more applicable and powerful for the end user as well as for mesh-network administrators. The latter has prooven to play an important role especially for communities considering a smooth migration from one technology to another.

2.1 Evolution of the Core Algorithm

2.1.1 Generation I

B.A.T.M.A.N.-I does not check for bidirectional link conditions when forwarding packets. This is an obvious design flaw. We didn't bother to add bidirectional link checks in the first experimental implementation that was meant to merely test the algorithm. The results were however promising.

2.1.2 Generation II

B.A.T.M.A.N.-II implements the bare algorithm with bi-directional link checks for meshnodes with one interface. Therefore the algorithm classifies the characteristic of a link between two neighboring nodes in unidirectional and bidirectional. A specific link is considered bidirectional for a certain time-frame if the reply (re-broadcast) of the self initiated (and broadcasted) originator messages has been received from the corresponding link neighbor. Simply speaking: If I hear you replying what I've said you must have heard me. To distinguish a re-broadcasted OGM received via a (so far) unidirectional link from an OGM received via a currently bidirectional link we introduced the unidirectional flag (UDF), marking OGMs irrelevant for nodes which are not the original initiator of the OGM.

2.1.3 Generation III and Beyond

B.A.T.M.A.N.-III now represents the current implementation of the algorithm which has definitively undergone the most changes since then. It's evolution is described in the following subsections.

2.1.4 Re-Broadcast OGMs only from Best-Ranking Neighbor

Each node re-broadcasts each received OGM at most once and only those received via the neighbor identified as the currently best-ranking neighbor towards the original initiator of the OGM. Thus, even if a node has received a yet new OGM, it is not re-broadcasted until this OGM has been also received via its best-ranking neighbor (and with a TTL greater or equal than the TTL contained in the last OGM received the first time via the best-ranking neighbor).

We identified that if a node (lets say A) with multiple potential neighbors (one best neighbor called C and one second best neighbor called B) between itself and a distant node D re-broadcasts every OGM received via any of its bidirectional neighbors, the neighbors themself may perceive a wrong picture about the best path via A to D. For example (the actually best) neighbor C might receive more OGMs via A than A actually received via any particular neighbor from D. This is particularly dangerous if node C has received more OGMs via another neighbors towards D as received by A via B, but less than totally accepted from A. In this case node C might select A as its best neighbor towards D while node A may select node C. A simplified illustration of such a scenario is given in Figure 1 with lines representing the existing links between nodes. The lines with an arrow are used to indicate distinct sequence numbers from originator D, re-broadcasted and received between particular nodes.

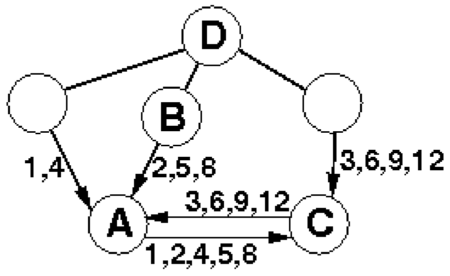

Figure 1: Illustrating of the (not) best-ranking neighbor problem

The described problem could be solved by requesting each node to re-broadcast only those OGMs received via it's currently best-ranking neighbor towards the original initiator of the OGMs. According to this rule and for the scenarios given in Figure 1 node A does only re-broadcast the OGMs received via node C.

Another benefit of this approach lies in the reduced traffic overhead from the routing protocol because only selected OGMs are re-broadcasted.

2.1.5 Time-Independent Topology Diffusion

The protocol has been purged from any timers related to the topology detection and diffusion mechanism. To describe this we will start with for what reasons timers have been used before. So far the B.A.T.M.A.N algorithm used timers for two reasons.

- Firstly, to identify a link as bidirectional by evaluation whether a re-broadcast of the recently send own OGMs have been received from a neighbor within the last BIDIRECTIONAL_LINK_TIMEOUT seconds
- Secondly, to rank a neighbor (selecting the best next hop towards a distant originator) according to the OGMs received via this neighbor first and within a certain time frame

In both cases, the reliance on timers could be eliminated in favor of (the always increasing) sequence numbers given with each OGM and corresponding originator.

Regarding the bidirectional neighbor check, whenever a node initiates the existence of a new OGM by broadcasting it to its potential neighbors, the originator remembers the last send sequence number and uses a by-one-increased number for the next broadcasted OGM. Here, the concept of a timeout value has been exchanged with a value defining the range of a sequence number frame, terminated at the upper boundary by the sequence number transmitted in the last initiated OGM. Whenever a non-self initiated OGM is received via a neighbor, the receiving node performs the bidirectional link check by validating whether the last received self-initiated OGM (re-broadcasted) from this neighbor contained a sequence number which falls within the upper and lower boundaries (lower boundary = upper boundary - BIDIRECTIONAL_SEQUENCENUMBER_RANGE) of the current sequence number frame.

Regarding the neighbor ranking the protocol defines another type of sequence-number frame (the Neighbor Ranking sequence Frame NBRF) which also has a fixed size defined by the RANKING_SEQUENCENUMBER_RANGE. An individual NBRF must be maintained for each known OG in the mesh.

The upper boundary of the NBRF (maintained for a particular other OG) is always defined by the largest sequence number received from this OG. Previous received OGMs with a sequence number lower than the lower boundary of the NBRF are not of interest any more and can be dropped. Whenever a new OGM from another node is received for the first time via a bidirectional link and containing a sequence number greater than the upper boundary of the corresponding NBRF the following tasks are performed.

- The frame boundaries are updated (as mentioned above).
- The sequence number is stored together with the neighbor via which it has been received and the IP address of the initial OG.
- The best-ranking neighbor is re-evaluated by selecting the neighbor with the most stored sequence numbers within the current NBRF.

The typical question people have raised when being presented with this approach is: "What happens with an OG (and related data and routing entries) if it just dies?" The answer is simple. Nothing happens until some kind of garbage collector comes along after quite a while (if more than an integral of the time calculated from the originator interval multiplied with the RANKING_SEQUENCENUMBER_RANGE has passed since the last OGM from this OG has been received) and completely purges this originator together with its routing-table entry and all it's data. It should be noted, that if the garbage collector purges an OG, none of the existing routes is optimized, its just removed and no processing time or traffic is wasted during this time for the defunct route to this OG.

2.1.6 Multiple Interfaces Support

The protocol supports multiple interfaces per node. Since the same message type is used for link sensing, neighbor discovery, bidirectional-link validation, and flooding a mechanism was needed to allow the initiator of an OGM to distinguish the received re-broadcast from his neighbors into two categories. Those, re-broadcasted by an neighbor on the same interface as being received and those re-broadcasted on another interface as being received. Where the rebroadcast of an OGM via all interfaces of a node is important for the flooding, the described distinction is important for the bidirectional-link check (one particular link can only be bidirectional between two particular interfaces). To achieve this separation the "is direct link flag" (IDF) has been introduced. This flag must be populated only by direct neighbors, only when re-broadcasting an OGM which has been received directly from the interface represented by this OGM, and only for those re-broadcasts transmitted on the interface via which it has been received.

2.1.7 Hiding Local Topology Information Beyond the Neighborhood

Nodes may alter (i.e. 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 OGM 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 nodes. For example a node with three interfaces (three originators) may be configured to send OGMs with a high TTL only for the first interface and a TTL of two for OGMs representing the second and third interface. This way, the node is still reachable via the IP of it's first interface but does not burden the nodes beyond its two-hop horizon with the efforts of maintaining and re-broadcasting OGMs from it's second and third interface.

Especially for (back-bone) 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, the possibility of using a small TTL for all their OGMs comes with the following advantages. 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.

2.1.8 Protocol Version Check

In order to prevent B.A.T.M.A.N instances with significant differences between their protocol (implementation) from falsely interpreting each others OGMs a protocol-version field is conveyed with each OGM. Whenever a B.A.T.M.A.N instance receives an OGM containing a protocol-version number that does not match the hardcoded number of it's own program version, the packet is silently dropped.

2.2 Maturity and Applicability Evolution

2.2.1 Flexible Interface Configuration

Alias interfaces B.A.T.M.A.N can be started on alias interfaces. Using alias interfaces allows the assignment of different IP addresses, netmasks, and broadcast addresses to the same physical network-interface device. By assigning different netmasks (like in Berlin 104.0.0.0/8 for OLSR and 105.0.0.0/8 for B.A.T.M.A.N) to the same interface, two different routing protocols can be used in parallel on the same hardware without influencing each other. This is particulary important to allow a smooth migration from an existing routing-protocol (like OLSR) to a new one (like B.A.T.M.A.N). For example, it enables node administrators to test the B.A.T.M.A.N protocol while running OLSR in parallel until the former proved to be better.

Support for equal netmasks on different interfaces B.A.T.M.A.N allows the assignment of equal netmasks for different interfaces. This is necessary to allow different interfaces to participate in the same mesh, thus releasing the administrator from the task of managing routing between different netmask (fractions) in the topology.

2.2.2 Gateway and Network Announcement

Besides the main task of making routing decisions B.A.T.M.A.N helps offering different services such as network announcement or internet connectivity which make a mesh network worthwhile.

Network announcement In certain situations it may be desired to offer dedicated infrastructure like webservers, mailservers, or similar services to the mesh network. Often this infrastructure operates outside the mesh but should be reachable to make these services available to the nodes in the mesh. In such a case a near B.A.T.M.A.N node may send out network announcements carrying the information that a particular address range (the IP addresses of the announced infrastructure) can be reached by forwarding the data to the same neighbor as selected for the node announcing it. Another purpose of announcing whole networks instead of individual host routes is to reduce the size of the routing tables.

Internet connectivity In order to provide a reliable internet connection B.A.T.M.A.N offers a set of features.

Gateway classes The B.A.T.M.A.N node which offers internet connection floods the network with information of the available internet bandwidth by setting a corresponding gateway class.

Routing classes The B.A.T.M.A.N node which wants Internet connection chooses its internet gateway based on certain criteria. The user can control these criteria by setting a routing class. B.A.T.M.A.N knows three different routing classes:

1. Use fast internet connection - B.A.T.M.A.N tries to find the best available connection by watching the uplinks throughput and the link quality.
2. Use stable connection: B.A.T.M.A.N observes the internet nodes and tries to find out which one is the most reliable. This mode is not implemented yet but will follow in batman 0.3.

3. Use best statistic internet: B.A.T.M.A.N only compares the link quality of the internet node and chooses the one with the best connection.

Preferred gateway In some cases it may be desired to override the auto selection mechanism by setting the preferred gateway.

UDP tunnel for GW traffic To encounter gateway switching B.A.T.M.A.N automatically negotiates UDP tunnels between the client and the internet gateway. The gateway distributes tunnel IPs among the clients.

2.2.3 Debugging and Observation

Demonizing batmand Like most routing implementations the routing application can be used as a daemon, operating in the background.

The daemon will log all critical information to the syslog (/var/log/syslog on most Linux systems).

Multiple debug-instances of the main daemon To observe the status of a batman-daemon operating in the background, multiple additional client instances can be started in monitor-mode to print out status information of the main daemon instance (which performs the actual routing). The batmand-client processes can specify different debug levels in order to provide various levels of verbosity to the user. The following debuglevels are currently available.

- Debug level 1 lists all known originators together with the best-ranking and further potential neighbors towards the final destination.
- Debug level 2 lists all available as well as the currently selected originator used as GW for tunneling packets to and from the internet.
- Debug level 3 instantly shows all changes to the routing table.
- Debug level 4 provides in-depth event logging and consequential decisions made by the protocol. This level is very very verbose.
- Debug level 5 provides (if enabled with corresponding defined during compilation) additional profiling and memory-allocation observations.

2.2.4 Visualization Support

Since no topology database is computed by the protocol an additional solution to create topology graphs has been implemented, the Vis-Server. Batman-daemons may send their local view about their single-hop neighbors to the Vis-server. The Vis-Server collects the information and provides data in a format similar to OLSR's topology information output. Therefore existing solutions to draw topology graphs developed for OLSR can be used to visualize mesh-clouds using B.A.T.M.A.N.

2.2.5 Policy Routing Support

As describe above, B.A.T.M.A.N. can run in parallel to other routing protocols without influencing each other. However, there is one issue where forwarding rules in the routing-table from different protocols may collide.

This happens when two protocols are configured to assign different rules for the same destination network mask, e.g. when configuring the default route. Then, normally one entry has a higher priority than the other (in case of OLSR and B.A.T.M.A.N the latter has a higher priority) with the consequence that every forwarded packet with a destination address destined to the internet will end up in the tunnel to the currently selected B.A.T.M.A.N GW.

As of version 0.3 alpha, B.A.T.M.A.N supports policy routing which allows for example the selection of routing rules depending on the source address of the to-be-forwarded packet. Therefore the new policy-routing implementation now uses its own dedicated routing tables. The default table (used by OLSR) and its default route is not shared with B.A.T.M.A.N any longer. Instead, the following tables are used (accessible by executing the linux command: ip route ls table 66)

- Table 65 for announced networks
- Table 66 for batman host routes
- Table 67 for the default route

Additionally, B.A.T.M.A.N (from the 0.3-and-beyond branch) defines rules to specify which routing table shall be used according to the source and destination address of each to-be-forwarded packet (these rules are available with the linux command: ip rule). In case of the Berlin Freifunk setup (with 104/8 and 105/8 netmasks for OLSR and B.A.T.M.A.N) these rules may be configured to always use the default route configured by OLSR for internet traffic except for packets with a source address matching the 105/8 netmask for which it would choose the default route configured in table 67. Since the implementation can not yet anticipate the exact intentions of the mesh-network administrators, these rules may be adapted to individual demands.

3 Real-Life Deployment and Experience

At the release party of B.A.T.M.A.N.-0.2 at the C-Base in Berlin, various people reported about their experience with B.A.T.M.A.N. Among the guests have been mesh-network administrators from Berlin, Goerlitz, Halle, and Magdeburg. Further experimental B.A.T.M.A.N. setups and related experience are documented in the internet by groups from Leipzig, Berlin Nord-Ost, and Weimar. An up-to-date collection of known B.A.T.M.A.N Mesh networks with lins to related websites can be found at http://open-mesh.net/batman/experience .

The typical size of these mesh networks consists of about 30 nodes running OLSR and B.A.T.M.A.N in parallel. As of today it is difficult to find conclude with a consensus between existing reports. One reason is definitely the fact, that recent releases have undergone many and major changes and the experience gained from the attendees goes back to very recent as well as already outdated and buggy revisions of our implementation. We hope that our stable release will motivate the groups to update their testbeds.

However, even groups that summarized their observations based on experiments made with the latest release candidate presented a quite different picture of B.A.T.M.A.N.s performance - ranging from the conclusion that the protocol causes more CPU load than OLSR and connects individual nodes much worse to the central GW as OLSR, to the persuasion that B.A.T.M.A.N. performs so outstandingly well that OLSR will not be enabled any more in particular mesh clouds.

The observations about higher CPU-load are also caused by the fact, that most OLSR setups used by the communities are configured with extremely slow Hello- and Topology-Update-Intervals. This is not comparable to the short Originator-Interval of one second that is the default setting in B.A.T.M.A.N. Our observation in the Berlin mesh with approximately 70 originators showed that the quality of routes computed by B.A.T.M.A.N. is comparable to OLSR as long as the latter does not loop. We didn't experience any routing loops with B.A.T.M.A.N., neither in real life setups nor in simulations. The increase of CPU-load caused by B.A.T.M.A.N. grows only linear with the number of nodes.

A snapshot from the B.A.T.M.A.N topology of the Berlin Freifunk Mesh is shown in Figure 2

4 Performance Evaluation

We have started to evaluate the performance as well as the traffic and processing overhead of our B.A.T.M.A.N implementation in a virtualized network environment (thanks to the OLSR-NG[3]

[3]The OLSR-NG project can be found at https://wiki.funkfeuer.at/index.php/Olsrd-ng

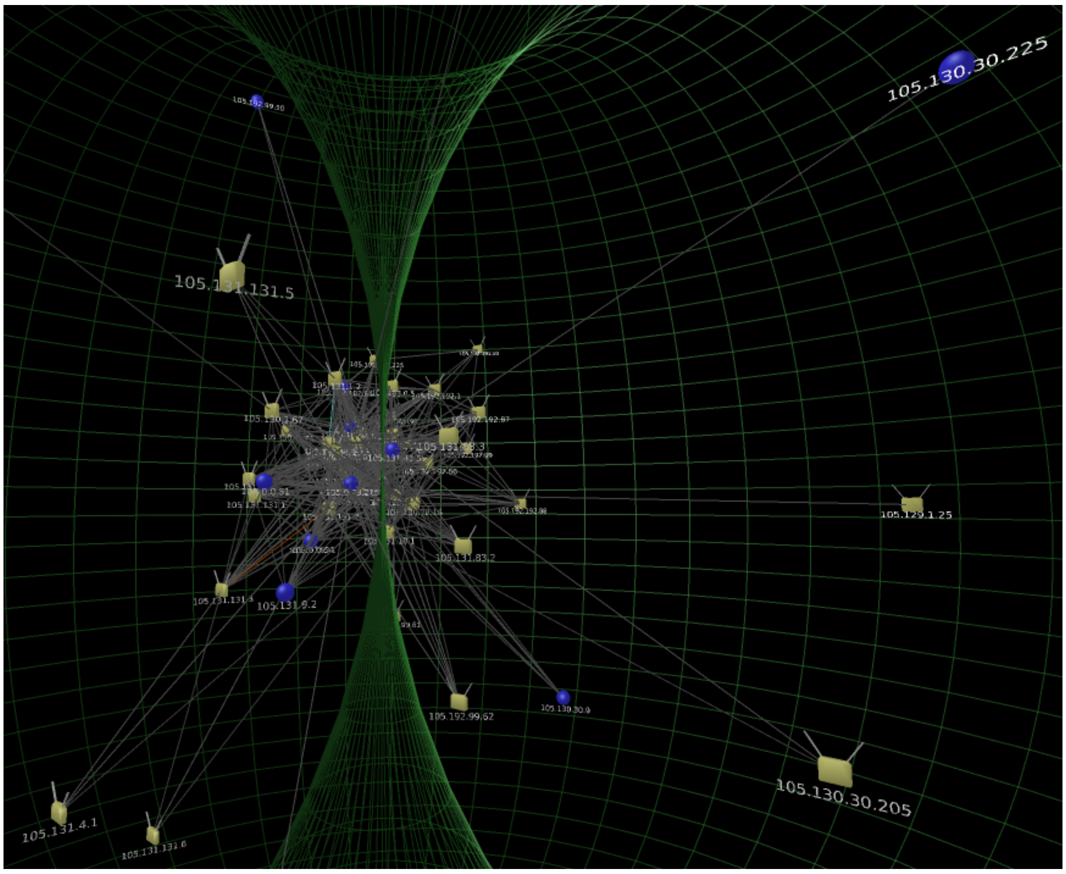

Figure 2: S3D Screenshot from 22.6.2007 of Berlin's B.A.T.M.A.N. Cloud

project for providing knowledge and resources for this task). Allthough, more effort is required to make profound statements about the B.A.T.M.A.N. algorithm in large scale networks, some preliminary results have been obtained.

The major problem regarding the applicability of the given results lies in the zero packet loss of the current emulation testbed. Because the biggest part (about 80 %) of the consumed CPU load (caused by the B.A.T.M.A.N implementation) can be assigned to the number of received and subsequently re-broadcasted packets, we expect a much lower CPU load in real-life setups.

In order to have some values for comparing the results each measurement scenario has been performed for B.A.T.M.A.N and the OLSR protocol. Thereby the following general setup and parametrization has been applied.

- The setup for the following emulation consisted of an "Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz" with a 32-bit gentoo Linux OS and using openVZ for virtualization of individual nodes.
- The network configuration consisted of a fixed topology with (except due to CPU overload) no packet loss. This has been achieved by connecting virtualized linux instances (running B.A.T.M.A.N of OLSR) to a layer 2 bridge and using ebtables for allowing only particular packets (from the virtual neighbors) to pass by)
- One single interface per virtualized node
- OLSR 0.5.0 has been used as the olsrd binary with the following configuration values.
  - Hysteresis no
  - LinkQualityLevel 2, LinkQualityWinSize 100
  - Pollrate 0.05
  - NicChgsPollInt 3.0
  - RcRedundancy 2
  - MprCoverage 5
  - HelloInterval 5.0, HelloValidityTime 200
  - TcInterval 0.5, TcValidityTime 250.0
  - MidInterval 5.0, MidValidityTime 100.0
- B.A.T.M.A.N 0.2 has been used as the batmand binary with an originator interval of 5000, a neighbor ranking sequence-number frame (NBRF) range of 10, and a TTL of 62.

4.1 CPU Load Depending on Number of Neighbors

Figure 3 shows the dependency between the average number of neighbors and resulting CPU load for B.A.T.M.A.N and OLSR. The last (6.5) value for OLSR has been omitted because the host machine was already overloaded. The measurement has been achieved with a constant value of 60 nodes. For the two-neighbor case the virtual network nodes have been configured in a line, having (except for the exteriors) one neighbor on the right and another on the left side each. The 2.9-neighbor case was given by a 30x2 grid configuration. The measurements for 3.4 and above are based on a 10x6 grid configuration with additional diagonal connections between nodes.

For the given scenarios, the measurement indicates that B.A.T.M.A.N scales lineary with the average number of neighbors and with an average gradient of 0.2.

4.2 CPU Load Depending on Number of Nodes

Figure 4 illustrates the average CPU load depending on the number of nodes in the mesh. Therefore, the virtual network has been configured as a simple grid with 10 columns (10 nodes per line) and filled up with as much lines (multiples of further 10 nodes) as needed for the corresponding number-of-nodes value on the x-axis. For the graph, the different number of average neighbors per node and measurement has not been respected and should be considered.

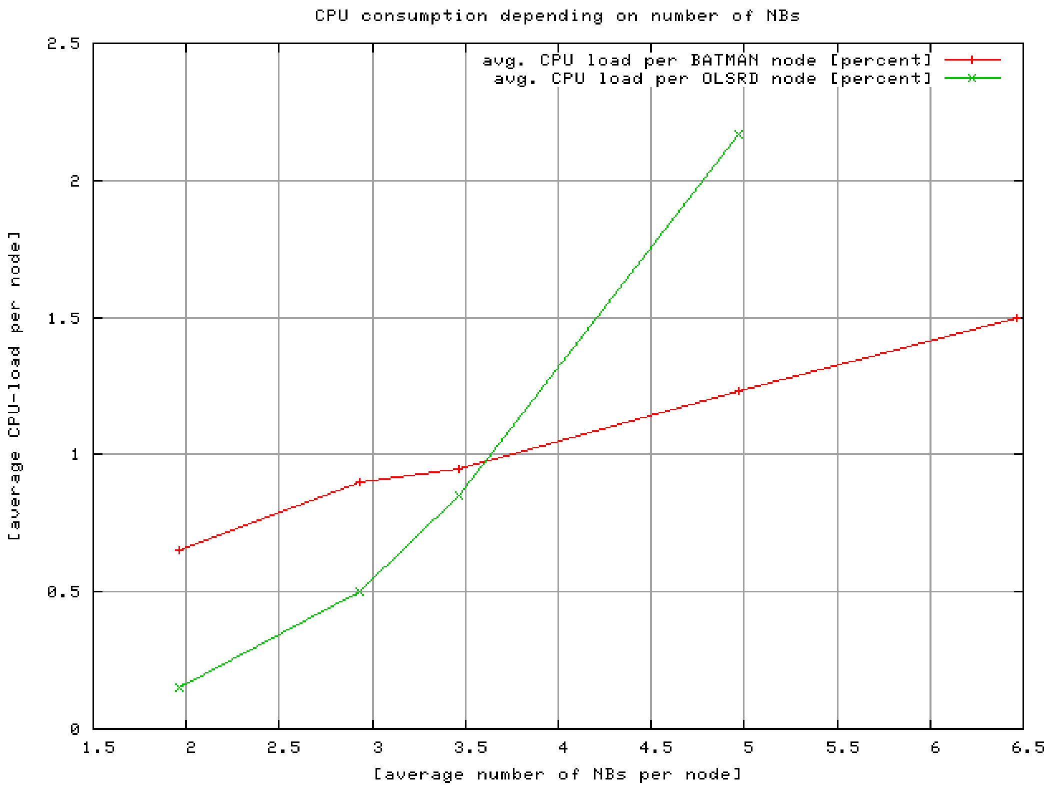

Figure 3: CPU consumption depending on number of neighbors

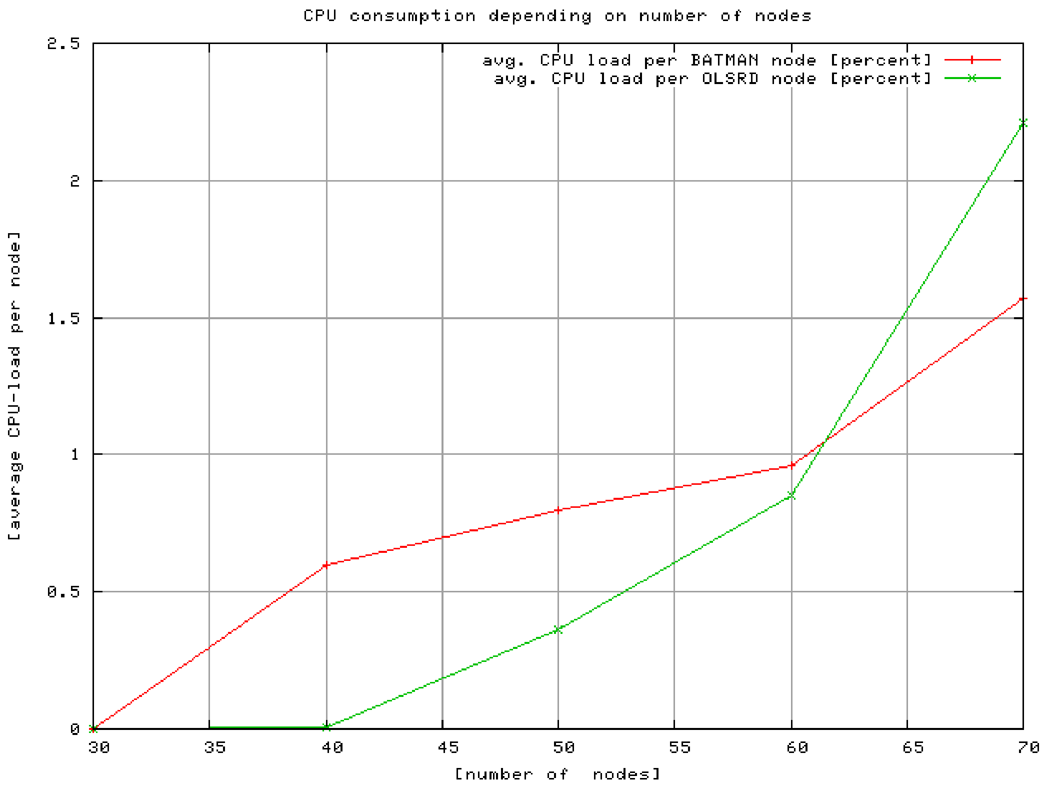

Figure 4: CPU consumption depending on number of nodes

4.3 CPU Load Depending on Originator Interval

Figure 5 shows the dependency between the originator interval and the resulting average CPU load per node in red and the reciprocal of the resulting average CPU load in green.

Thereby the measurement indicates that for the given scenario and measurement samples the originator interval has a reciprocal influence on the average CPU load per node.

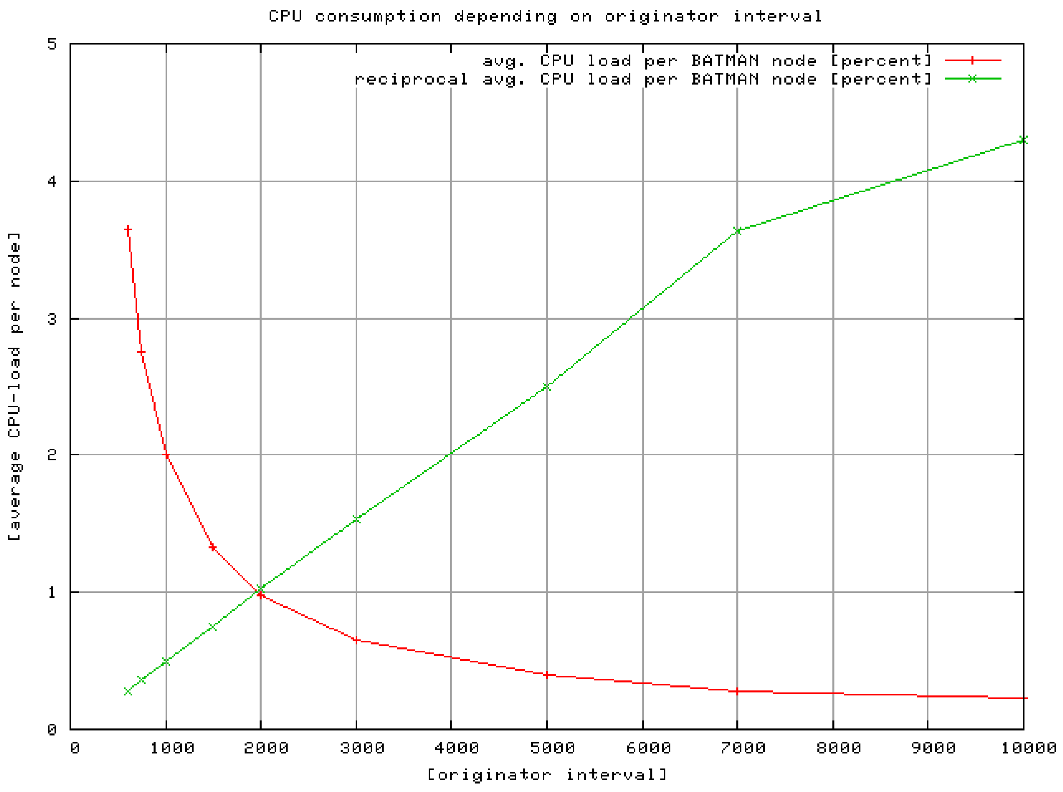

Figure 5: CPU consumption depending on originator interval