University of Dortmund
Department of Computer Science
Project Group 439

BEEhive
an Energy-Aware Scheduling and Routing Framework

Final Report

Instructors:

Prof. Dr. Horst F. Wedde (wedde@ls3.cs.uni-dortmund.de)

ME Muddassar Farooq (muddassar.farooq@cs.uni-dortmund.de)

Project Manager:

ME Muddassar Farooq (muddassar.farooq@cs.uni-dortmund.de)

Students:

Lars Bensmann (lars@almosthappy.de)

Thomas Büning (thomas.buening@udo.edu)

Mike Duhm (mike.duhm@udo.edu)

René Jeruschkat (jeruschkat@web.de)

Gero Kathagen (gero.kathagen@uni-dortmund.de)

Johannes Meth (J.Meth@LANdata.de)

Kai Moritz (kai.m.moritz@gmx.de)

Christian Müller (christian.mueller@uni-dortmund.de)

Thorsten Pannenbäcker (thorsten.pannenbaecker@uni-dortmund.de)

Björn Vogel (BjoernVogel@gmx.de)

Rene Zeglin (rene.zeglin@udo.edu)

Contents

List of Tables

List of Figures

Listings

Part I.

Nature Based Routing — A Framework for Ad–Hoc Networks

1. The Wisdom of BeeHive – An Introduction to Honey Bees

By Thorsten Pannenbäcker (thorsten.pannenbaecker@uni-dortmund.de)

1.1. Introduction

"Cooperation in foraging has evolved in many species of group-living organisms, including insects, spiders, colonial invertebrates, fishes, birds, and mammals (...). One of the most sophisticated forms of cooperative foraging occurs in a social insect, the honeybee (*Apis mellifera*). The thousands of foragers within a typical honeybee colony work together in harmony, forming an ensemble which can monitor an area of 100 square kilometres for flower patches, choose among these patches to focus the colony's foraging labor on the riches ones, and adjust its patch selectivity in relation to forage abundance and colony need (...)." [SV88]

But how is this cooperation possible? In the following some processes are discussed, which primarily illustrate how this information is spread and processed throughout the honey bee colony.

1.2. Communication of Honey Bees

Honey bees are communicating by using special movements, the so called dances, which were first extensively examined by Karl von Frisch and described in his recommendable book [Fri65]. A special dance is existing for every kind of information interchange, especially for the announcement of food sources.

In a beehive, thousands of worker bees are fulfilling certain jobs. The most interesting worker bees are the ones, who are responsible for collecting and processing the nectar: scouts, foragers and storer bees. The storer bees take the collected nectar from the foragers and store it in the combs of the hive. The foragers tasks are limited to find the nectar sources (the flowers), collect the nectar and bring it home to the hive where it is taken by the storer bees. Finally, the scouts have the job to find new nectar sources.

Basically all worker bees are the same at the time of birth and can do all jobs. Normally the first days of their short life they are working inside the hive, before they go and work outside. Actually, there is no real difference between scouts and foragers. They are only loyal to a certain nectar source in an individual degree. The great majority remain loyal to their nectar source over a very long period (some days) even if the source is run dry. During this time they rest inside the hive and will only perform a few flights from time to time, examining for changes in their source. Only very few bees become disloyal to their source more often and become scouts. After the discovery of another source, they will begin to collect nectar and become normal foragers again.

1.3. Information Interchange

Not all foragers are searching for nectar sources on their own. Most of them will fly to places described by other foragers. To find these places the foragers basically only need to know about the distance and the direction.

To exploit each nectar source optimally the active foragers on this source will hire new comrades through performing dances. But this hiring through dances does not happen automatically and steady, but depends on different ascendancies.

1.3.1. Influencing Ascendancies

The dances performed on the dance floor (a certain but not exact marked-off region of the hive, normally near the flight hole) convey different information and vary in liveliness. The decision to perform a dance depends on the factors discussed below in detail.

The most important one is the sweetness of the nectar, which isn't necessarily correlated to its sustenance. This most important ascendancy determines the dance threshold for a forager. However, this stimulus isn't sufficient. Additionally the nectar must be profitable and easy to reach to let more foragers exploit it. The distance from the hive, the steady flow of nectar, the general situation of nutrition, the relative changes of quality, the weather situation, and also the daytime are some other factors influencing the foragers to dance. (following [Fri65], pages 240ff.)

1.3.2. The Direction Information

Since other foragers have to find a (newly discovered) foraging site on their own, it is necessary to inform them about its direction. The forager uses different stimuli to ascertain the correct direction. The most important stimulus is the sun. With its

faceted eye the bee is able to determine the angle between its own trajectory and the sun. Even if the sun is covered, the bee is able to determine its direction using the polarized blue skylight that depends on the suns position. Growing experience puts the bees in position to orientate towards striking landmarks and interpolate the direction when the sky is completely covered.

Generally, only the flight from the hive to the nectar source plays a major role in calculation of direction. It is sensible because they can "think of" the way back by themselves by inverting the first way. If the nectar source moves, the forager can't find its way back correctly, but that should never happen in nature. Only when the forager should have returned to its hive but, in fact, isn't, it starts looking for it. In this case also the way back will be considered when calculating the direction. Hence the new direction indicated during the dance is the bisectors of the angle between the place of arrival and departure of the source (hive).

The direction information is passed to other bees by the dancer during a special passage of the dance by moving in the corresponding direction. On horizontal dance floors the dancer can orientate itself by looking at the sun or a part of the sky analogue to the navigation during the flight. But since the combs in the dark hive only provides a vertical dance floor the bee is able to transpose the angle to the sun into an angle to the plumb. That means that a flight directly into the sun leads to a waggle run straight upwards and for deviations in corresponding directions. This will even work for slanting planes until the well developed sense of gravity are limited on a nearly horizontal plane. (following [Fri65], page 127ff.)

1.3.3. The Distance Information

Researchers thought that the bee's measurement of the distance to a destination is based on the energy consumption. For instance, von Frisch excluded the absolute distance in metres and the flight duration at a constant velocity (30 km/h by the way) as basis. Different experiments with following and head wind, down- and uphill flights, or additional plumb weights seemed to support this thesis. However, he didn't exclude optical stimuli, since flight over areas with few possibilities of orientation (for instance a smooth water surface) showed smaller announced distances. After experiments of Esch and others the optical stimuli are considered to be the primary (or even the only) source for the distance measurement. This thesis doesn't necessarily contradict the energy consumption thesis, since bees are flying in different altitudes (according to their payload), what leads to different perceived optical flows (a flight with great altitude above a certain pattern seems slower than with small altitude).

During the dance, information about the distance is passed in a certain phase (the same as the direction information). If the perceived distance increases, this phase lasts longer and the liveliness, the speed (in rounds per minute), decreases. (following [Fri65], page 65ff.; [EB95]; [EB96])

To allow a short insight to this exciting field of honeybees behaviour some special descriptions will follow. To understand the dependencies and mechanisms more exactly it is recommended to read the books mentioned in the bibliography, specially [Fri65] by Karl von Frisch and [See95] by Thomas D. Seeley.

1.4. The Dances of Honey Bees

Honey bees do different types of dances. Eeach one conveys special information to comrade bees. The three most important types are: the round dance, the waggle dance and the tremble dance, which all are closely related to the forager bees.

Round and waggle dances serve the recruitment and reactivation of foragers. Interestingly it is distinguished between two distance ranges. The round dance is for food sources near the hive and doesn't contain information about direction or distance. If the food source is farther, this information is passed by doing waggle dances. The exact identification of a certain food source is done by the bee's sense of smell. On the one hand bees smell the special flower scent adhered to the dancer and on the other hand they smell foragers already collecting nectar at the food source, which are secreting a special odour helping nearby foragers flying around finding the source.

1.4.1. The Round Dance

If a rich nectar source near the hive is found, then inactive foragers are asked to share exploitation of this source by the so called round dance. Passing a part of it's nectar to storer bees the forager keeps still. "Now the round dance begins. With quick, tripling steps the forager is moving around in a circle so narrow, that mostly only a single comb cell is lying inside this circle. She runs around on the six adjacent cells in which she soon turns around in a sudden turn and runs on in the opposite direction, to turn around once again in a new swing and run in the earlier direction again and so on. Often one or two whole rounds are done between two swings, often also only three-quarter or a half of a round." ([Fri65], page 29) After the dance is interrupted or after the dance lasted for several rounds, the forager cleans itself, loads fuel for the next flight and starts its next forage flight quickly.

"It is never danced on an empty or poor populated comb, but only in dense crowd. Thus, during its rounds, the dancer is in direct contact with other bees, which – in right mood – are tripping after her, putting their feelers on her rear body." ([Fri65], page 30).

If a forager already knows the destination indicated by the scent, she will take the usual short preparations and will take up her collect flights to the remembered source again. A certain forager will only fly to a single food source and if the flights doesn't appear worthy, because of missing nectar for example, she won't look for a new source (neither on her own nor by following some dances). Instead, she flies home again and rests near her comrades exploiting the same source. Comrades of the same group are possible to be reactivated only by the scent of an active forager, without a dance, though only in 40% of the cases. Contact to a dancing group comrade has a success rate of 90%.

exploiting the same source. In 40% of all cases, it is possible to reactivate comrades of the same group only by emmitting the scent of an active food source, without a dance. Direct contact to a dancing group comrade has a success rate of 90%.

Does an inactive, recruited bee not know the destination yet, she will leave for a flight only knowing the scent of the nectar source. The forager searches the whole area in all directions around the hive for the special scent. The more often and the more lively the dances are, the more foragers are being recruited.

1.4.2. The Waggle Dance

The normal flight area of bees of the Krainer breed (*Apis mellifera carnica*) for example, is about six kilometres in every direction from the hive, even more in special cases. For this distances it's obviously not possible to communicate the nectar sources through round dances because the searched area is large.

so, how are distance and direction passed to a potential recruit in this dance? First, a description of this dance: "During the typical waggle dance, the bee runs a short distance straight ahead, returns to the other side in a half circle, runs the straight distance again and returns in another half circle in the direction and so forth in a regular change. The run straight ahead receives a special emphasis through the lively waggling. This arises from quick deflections to the side of the whole body which are greatest at the tip of the rear body and smallest at the head: The axis around which the side swinging is done, must be thought of as been short in front of the bees head and vertical to the subsoil. The back-and-forth-moving is repeated 13 to 15-times per second; expressed different: the movement has a frequency of 13-15 Hz." ([Fri65], page 56f.) The waggle phase is emphased additionally by a sound, which the bees, for lack of a sense of hearing, only notice as vibrations transmitted through the floor. Even if the sound is assumed to

support the effectiveness of the dance it doesn't seem to contribute any information.

The distance instruction arises from the waggle time, i.e. the time which the dancer takes for the straight distance during the waggle dance. The waggle time is not constant during a whole dance but varies in narrow bounds in which the mean value of the waggle time is correlated closely to the distance from hive to source.

The direction to the destination is plainly shown by the direction of the straight waggle distance. The direction is determined during the flight and passed as described above.

If the food source can't be reached on a direct beeline, the values for it are passed anyway. Recruited bees are looking for a roundabout route themselves when they come across an obstacle. Afterwards they take the roundabout route directly.

1.4.3. The Tremble Dance

"It is as if they suddenly acquired the disease St. Vitus's dance [chorea]. While they run about the combs in an irregular manner and with a slow tempo, their bodies, as a result of quivering movements of the legs, constantly make trembling movements forward and backward, and right and left. During this process they move about on four legs, with the forelegs, themselves trembling and shaking, held aloft approximately in the position in which a begging dog holds it forepaws." ([Fri23], page 90, quoted from [See92], page 375). Karl von Frisch had no explanation for this dance, except as reaction to unwellness. Thomas D. Seeley did examine the dance again later and found an other explanation.

Therefore this dance has another meaning as the earlier ones, because it helps organization within the hive: "This suggests that the *message* of the tremble dance is "I have visited a rich nectar source worthy of greater exploitation, but already we have more nectar coming into the hive than we can handle." It is also shown experimentally that the performance of tremble dances is followed quickly by a rise in a colony's nectar processing capacity and (...) by a drop in a colony's recruitment of additional bees to nectar sources. These findings suggest that the tremble dance has multiple *meanings*. For bees working inside the hive, its meaning is apparently "I should switch to the task of processing nectar," while for bees working outside the hive (gathering nectar), its meaning is apparently "I should refrain from recruiting additional foragers to my nectar source."" ([See92], page 375). Through this dance, a match between collecting activity and processing capacity is adjusted and a bottleneck is avoided. Since the tremble dance isn't performed only on the dance floor but in other areas of the hive too, bees resting or doing unimpatient tasks switch to the role of nectar processing, to increase nectar processing capacity.

1.4.4. Other Dances

Many other forms of dances do exist. Information about them and their meaning are hardly available in literature.

Of special interest could be, that bees are dancing too, if they look for a new nesting place. If a colony divides itself, one part swarms and settles somewhere nearby in the open air. Scouts are searching the neighbourhood for suitable nesting places and perform waggle dances pointing to the found location. Over some days these dances can be performed until only dances for a single location are left. First when all dancers "are of the same opinion" the swarm moves into the new nesting place.

1.5. Formalized Models of Honey Bees

The described decentralized decision making in honey bee colonies, where foragers work without the help of any central authority and organize their activities through dances. In this way a colony distributes its work force to different food sources according to their quality.

Some models describing honey bees and their behaviour can be found in literature. The emphasis is on the nectar collecting through communication. A model based on differential equations can be found in [See95] and an agend-based model in [Sum00].

2. The BeeHive Routing Algorithm

By Thorsten Pannenbäcker (thorsten.pannenbaecker@uni-dortmund.de)

2.1. Introduction

The BeeHive routing algorithm was developed for energy efficient routing in wireless ad hoc networks. The other standard algorithms for such an environment are DSR (Dynamic Source Routing), AODV (Ad-hoc On-Demand Vector Routing) and DSDV (Destination Sequenced Distance Vector). It is a layer 3 protocol following the ISO/OSI standard and is completely independent from higher (nearly) and lower layers. Like DSR it uses the strict source routing option of IP, this means the complete route for a packet is part of the header. Additionally, some packets used by BeeHive contain other information in the optional part of the IP header as well. A reference implementation was developed for the network simulator ns-2 (see chapter 3 for details) and it was also implemented in Linux (see section II for details). This chapter refers to the reference implementation and should help to understand the principles of BeeHive.

The algorithm was inspired from natural honey bees and their behaviour for collecting nectar. Like them, it has no global information about state of the network, instead bees communicate with each other to organize the routing framework on similar bee principles, as described in the last chapter.

BeeHive is based on three main thoughts:

- information feedback from the routes
- load balancing and adjusted capacities for each route, according to their quality
- specialized routing behaviours for different optimizations

Abstractly, all packets sent over the network are assumed to be bees. Each node is a hive, where the routing layer consists of the three logical parts: entrance, dance floor and packing floor, as shown in figure 2.1. These three parts do different jobs. The entrance is the interface to the lower layers, the packing floor is the interface to higher transport layers, like TCP or UDP and the dance floor is the main routing instance.

Like it's natural example, BeeHive is kept as simple as possible to reduce the used computing power and also, to avoid complicated interdependencies between the parts. Each bee can be treated without the knowledge of all other bees. Only very little not totally local (bee local) information is needed inside a node.

2.1.1. Analogies Between Natural Honey Bees And The BeeHive Routing Algorithm

As natural honey bees appear to organize their work very efficiently (refer to chapter 1) it was tried to copy the behaviour of honey bees to the algorithm of BeeHive.

Like in nature bee colonies there exist different kinds of bees which have different jobs. In BeeHive different kinds of workers exist, but the most important are the foragers responsible of transporting the payload. In nature they fly from the hive to a exploitation site, collect the nectar there, and return back to the hive with it. In contrast to this natural example the payload in computer networks don't need to be brought to the home hive, but transported from there to other places, so this process is inverted. Every outgoing forager takes a data packet and brings it to its destination. The transmission of data is limited to the presence of foragers. If there aren't foragers available, no data packets are transmitted.

On its way to the destination a forager collects information about the quality or the costs of its route to be able to judge the quality of the route, when it arrives back at destination (refer to chapter 1.3.1). The judgment of the nectar quality itself is replaced by the amount of data waiting to be delivered to a certain destination as the main indicator for dancing (refer to chapter 1.4). The dancing again is been abstracted to a number of possible clones, which is calculated by an evaluation function. During the duration of a dance, the forager is copied as often as this number allows, if it is requested. Through this mechanism the bees are able to distribute the delivering of the data packets to different routes according to their capacity and quality.

Normally, foragers are flying two ways: one to their nectar source and one home to their hive again. For BeeHive this would mean doubling the traffic. As this doesn't seem to be acceptable, foragers only fly to their destination and stay there. But as many communication links are bi-directional, the destination node is also a source node sending packets to the original source node, becoming the destination. So such bi-directional links enable a two way flight for the foragers. If the number of packets in both directions is equal (like TCP, which acknowledges every packet), it will not cause any problems. If there are great differences between the two streams (like UDP, which has a major stream in one direction and a very small control stream in the other direction), a special

swarming technique is used (see 2.2.3).

2.2. Types of Bees

In this sub chapter the different bee types are described that will help the reader in understanding the routing and working behaviour of BeeHive. For technical details of a real implementation please refer to the according chapter in the section "BeeHive inside the Linux kernel" (section II).

For the routing (assumed, that a route is already known) the normal IP header is extended with the standard optional part. This optional part exists for normal source routing option provided by IP. For example, this is also used in DSR. Additionally, the BeeHive header contains some bytes with some special BeeHive information. How these bytes are used will be described later. So, all bees (packets) transferred between the nodes on the net conform to a standard IP header.

On its way from the source to a destination this form of representation is kept as well. The packets are treated as normal source routed packets except, that the special information bytes of BeeHive must be updated at these nodes.

When a packet arrives at its destination, its data is passed to the higher layers. From the route and the information bytes a struct is generated, representing a bee. It contains the route and some other information needed. What this additional information is exactly will be described later on.

The BeeHive algorithm uses three types of bees. Normal foragers transport the data from one node to another. Scouts are used to retrieve routes if they aren't available yet. Packers are not sent at all. They are used to receive data from application layers and pass it to the foragers, or if no foragers are available to buffer these data.

2.2.1. Scouts

Scouts are used to find connections between two nodes, if no route is already available. Technically, it is a broadcast packet with a TTL (time to live) mechanism, very similar to the route discovering in DSR, for example.

In the BeeHive header only the route taken so far and an ID is saved. The destination and the TTL values are part of the regular IP header. If the TTL value isn't exceeded, a scout is broadcasted to all neighbours of a node and their address is added to the scouts route, until it has reached its destination. The ID helps to identify each scout uniquely.

Scouts are created in the packing floor if a packer bee can't find an appropriate forager and some other conditions are met. After it has found it's destination it is sent back

as a normal source route packet and passed to the dance floor, where a forager is built from it.

2.2.2. Foragers

The foragers are the real workers in BeeHive. Foragers are bees that transport data. There are different kinds of foragers helping to provide adaptive routing behaviour with respect to different need of communication links, like delay or throughput. Of course, the BeeHive header consists of a route and a code for the foragers kind. Different to most other protocols, it also provides a field for gathering information about the condition of a route. This information is provided by the nodes and helps to have a feedback from the routes in order to route data packets over the best available routes.

Foragers are stored in the dance floor when they have finished a flight and are waiting for the next data packet to be transported.

Here, the three main ideas appear. As information is collected for every forager kind, all routes can be evaluated in terms of a special criterion, like delay. This information helps to find the best routes for certain packets so they aren't sent through randomly chosen routes but the best ones. As this information will get worse because more traffic is on a certain route, less bees will be recruited for this route. Other routes, originally not so good, may become evaluated better, because their is less traffic. So, the capacity of each route varies through time, not flooding one link as soon as it's available. This behaviour helps to balance the load and to avoid congestions, since the transport is limited to the available foragers.

Basic Foragers

They are not specialized at all. They don't collect data and their dancing behaviour is only determined by the amount of waiting data packets. In real applications this kind of foragers should not be used.

Delay Foragers

The aim of the delay foragers is to reduce the delay. For that purpose it stores its departure time at the source node. When it arrives at the destination the difference between arrival and departure time is calculated. This value is compared to a mean value of the proceeding delay values and together with the number of waiting packers it leads to a dancing number.

The limitation to available foragers might seem quite contra productive since all packets could be send immediately once a route is known, it shows that the delay isn't worse then with other routing algorithms or even better. One reason for that might be, that the load balancing and limitation of the traffic helps avoiding congestions.

Throughput Foragers

The throughput foragers are not implemented at this moment. They should contain a mixture of the minimal throughput rate during the route and the delay. The throughput rates must be obtained from the network interface.

Energy Foragers

Energy foragers aren't implemented, too. They should contain information about the total energy consumpting caused by all transmission, receivings, and computing. These values must be obtained from the network interface.

Lifetime Foragers

Lifetime foragers tries to improve the lifetime of an ad hoc network by avoiding the nodes with low batteries. For this purpose all battery levels between the source and destination are stored in some bits in the information bytes. Together with the number of waiting packers the minimum and the average of these values lead to a dancing number.

2.2.3. Swarms

Swarms are control packets. They are needed when a communication link isn't balanced in both direction in terms of the amount of data packets. This will lead to the sending node running out of foragers all the time and a flooding of the destination by incoming foragers. To avoid this the swarming technique is developed.

If the difference between incoming and outgoing foragers reaches a certain threshold a swarm of foragers is flying back to their originating node, controlled by the dance floor (see 2.3.3). To avoid a quadruplicating of the necessary control overhead, a forager is chosen as swarm leader, while the others are put into a data part of this swarm leader, represented by their routes, their kind, and their last route information. When this swarm, sent only as one control packet, arrives at its destination, the data part of the swarm leader is evaluated and the foragers are rebuilt from the contained information and added to the dance floor like they had arrived normally.

2.2.4. Packers

Unlike the other bees, packers are only consisting inside the nodes and not sent over the network. They represent data packets received from higher layers. It is their job to find a matching forager, which can transport the data to its destination. Packers are stored in the packing floor until they find a forager.

2.3. Architecture of BeeHive

Each node in a wireless ad hoc network represents a hive. The hives again are containing the bees and through that, the routing information. The nodes are independent from each other and do not need to interchange control packets to be able to route. All the information necessary to route a packet is generated locally or its retrieval is initiated locally.

As an ISO/OSI level 3 protocol, BeeHive provides interfaces to the levels 2 (MAC) and 4 (connection securing, like TCP or UDP). These interfaces are called entrance (interface to the MAC layer) and packing floor (interface to the transport layer). Between these two entities the dance floor is positioned where the routing information is stored.

All the layers and actions underneath level 3 are seen as outside a hive (the world) and packets entering from or going to there must pass the entrance, like in natural honey bee hives. The entrance must control the acceptance, refusing and forwarding of packets.

The layers above layer 3 are seen as the local part (the home hive). The job of BeeHive is to transport packets from its home hive to their destinations and so, there is an interface where the data is accepted from application layers and distributed to the workers, the foragers. This packing floor is an instance to coordinate the packers (bees containing data from the application to be sent) and the foragers from the dance floor.

2.3.1. Entrance

As seen in figure 2.1 the entrance is the interface to the network layers, especially the MAC layer. Although BeeHive was developed and implemented for and simulated with IEEE 802.11 ad hoc networks, it is completely independent from certain MAC protocols. It doesn't take advantage of information available from for example the IEEE 802.11 protocols information like signal strength etc., although there would be some possibilities of improving the performance in terms of energy efficiency and other performance issues.

The entrance must handle all incoming packets as it's the interface to the MAC layer. In terms of BeeHive that will be scouts and foragers. And of course it must sent the

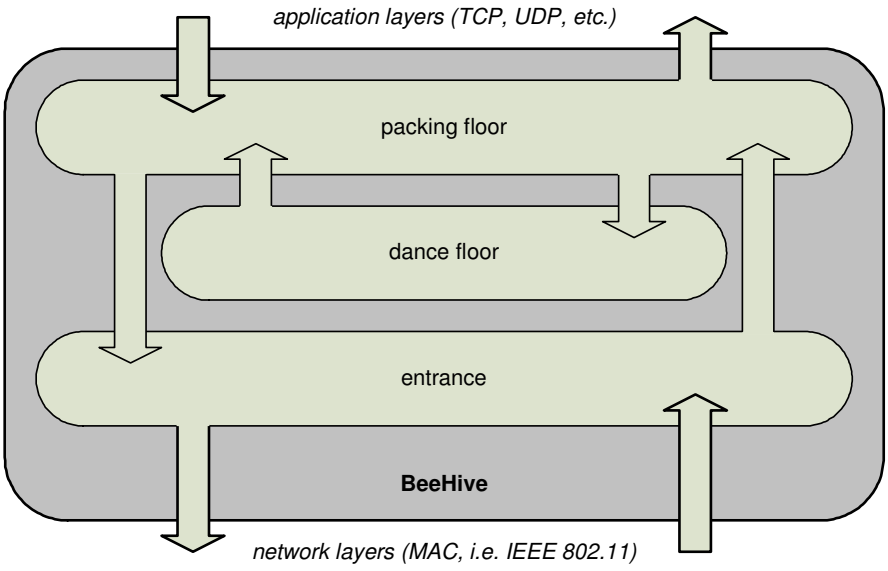


Figure 2.1.: Overview of the BeeHive architecture

outgoing packets of a hive.

The scouting mechanism of BeeHive is very similar to for example DSR. So the handling of scouts in the entrance isn't very different, too. The forwarding is done with respect to the scout's TTL (refer to chapter 2.2.1). They are broadcasted until they have reached their destination and are sent back to their source from there.

Scouts with exceeded TTL are deleted. So are scouts that have been seen already. The source and ID of every incoming forager are compared to a list of already seen scouts and added if not in this list already. This helps avoid broadcast storms but has some disadvantages in terms of route diversity. If there is only a single connection between to parts of a network, only the first scout is forwarded to the other part. All following scouts are deleted, even if they have taken different routes in the first part. The only exception is the arrival of scouts at the destination. All arriving scouts are sent back from there.

Before a scout is broadcasted to the neighbours, each hive looks if it maybe has a route to the destination already by demanding a forager from the dance floor. If so, the route is completed and the scout sent back immediately. This mechanism helps improving the route diversity an saves a lot of broadcasted scouts.

The handling of foragers is more interesting as BeeHive has one main difference to other protocols. Of course they are handled like in every other source routing protocol but additionally, they are provided information of their routes quality. This is a major

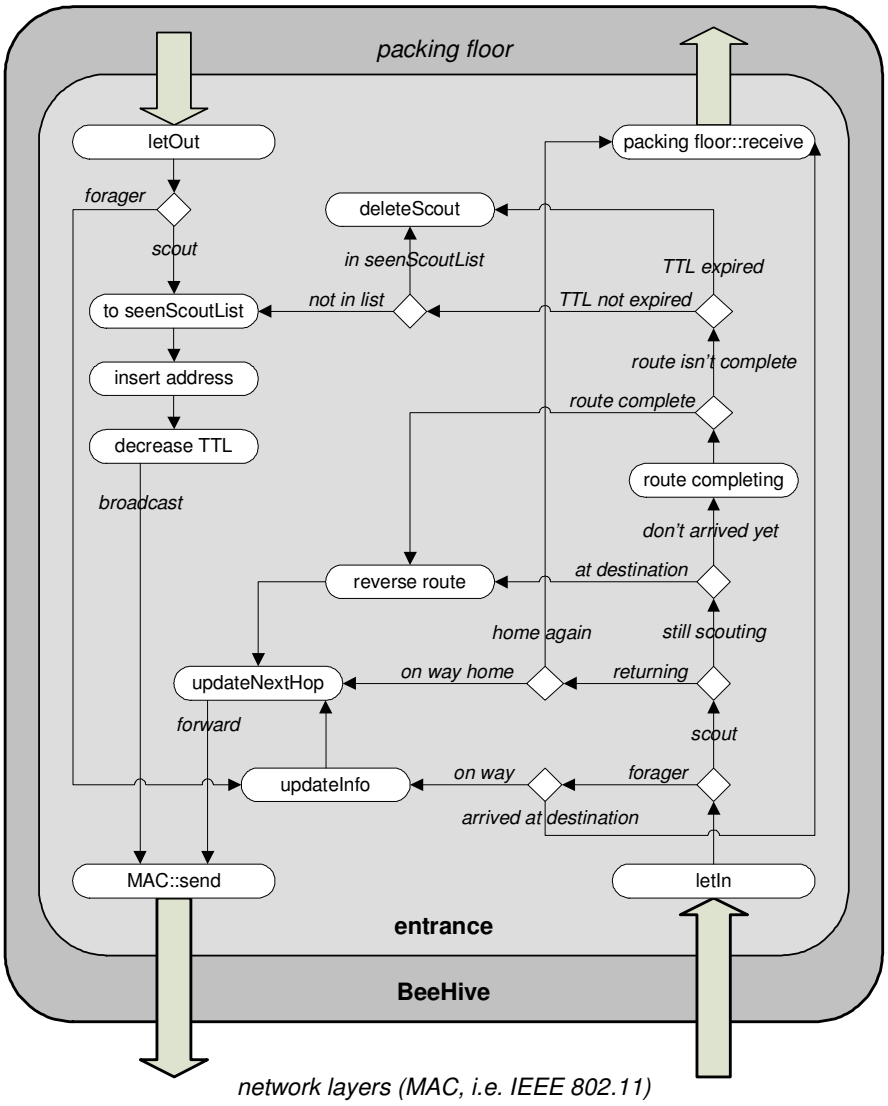


Figure 2.2.: The entrance

condition for the working of BeeHive. Dependent on the foragers type (refer to chapter 2.2.2), for example the nodes battery level or the bandwidth of the last taken hop is written into the BeeHive header. By evaluating this information after a flight, the forager is able to judge its routes quality and decide if it's promising to recruit more foragers for it.

Even if the information provided can be difficult to get technically, it shouldn't be to great a problem in theory. All information is based on local states and there is no need for global information at all. The effort to provide this information is payed back by specialized foragers able to transport the data specialized, having the edge on the other protocols.

2.3.2. The Packing Floor

Like the entrance is the interface to the layer 2, the packing floor is the interface to layer 4 the transport layer, like TCP or UDP. It takes all the packets from this layer and handles them as data packets. This means, it builds packers (refer to chapter 2.2.4) each time a packet arrives. They try to find a matching forager, which can transport the data to its destination.

It must be stressed here, that BeeHive takes one assumption; the only point in which it is not completely independent from other layers. To work properly, it's necessary that all data packets arrive at the routing layer, that means at the packing floor, as soon as they are available. BeeHive puts them into an internal buffer and uses this buffer filling level as one criterion for the evaluating of the dance number of the foragers, leading to an adaptation of the capacity for a link as needed. With UDP this is no problem at all, since it sends packets without any form of acknowledgment. But TCP will acknowledge every or every few packets. In blocking mode TCP will wait for these acknowledgments before it sends the next packet to the routing layer. This will lead to an empty buffer all the time and hence, the returning foragers think that there is no need to dance at all. So, there are very few foragers available leading to very poor performance. So the non blocking mode of TCP is absolutely recommendable for good performance with BeeHive. This mode does not wait for the acknowledgments before sending the next packets.

Now, there are two possibilities of entering the packing floor. Either from the internal part of the hive through the arrival of a data packet from the transport layer, or external from the world through the entrance if a forager or scout is returning home.

The handling of scouts and foragers is limited to the providing of the buffers fill level, the number of packers waiting and for the foragers the forwarding of its transported data to the higher layer. This was already described in the paragraph above, so it should only

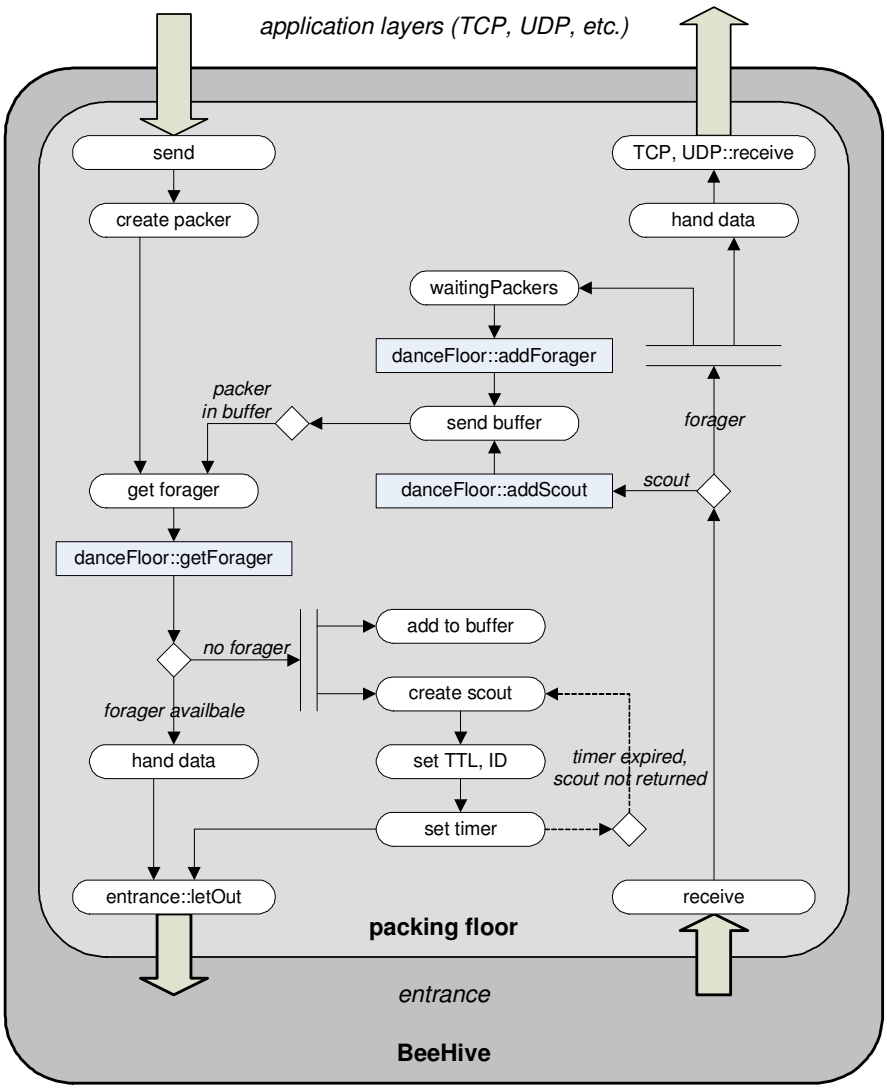


Figure 2.3.: The packing floor

be pointed out that this information is extremely important for the dancing of scouts and foragers.

The packing floor's real task is the placement of suitable foragers to the received packers. For that purpose it also handles the generation of scouts if destinations are not available yet. This is a bit more challenging than in other protocols because BeeHive can run out of foragers for a certain destination. So the packing floor must be aware of the possibility that foragers return home soon and not sent scouts immediately.

But if enough foragers are available after the first few transmission because of their dancing, it should be no problem to find some. So the demanded destination and optimization criterion are passed to the dance floor which returns a matching forager, if available. The packers data is passed to the forager and it's deleted, while the forager is forwarded to the entrance where it can start its flight.

2.3.3. The Dance Floor

The dance floor is the heart of BeeHive. Here the actual routing decisions are taken, the real job of BeeHive. Like in nature, inside BeeHive the foragers are trying to recruit new foragers for their route or rest while they are waiting for the next flight.

There are two main possibilities of using the dance floor in BeeHive: either in adding a forager after a flight or in demanding a forager for a flight.

Adding a forager is normally very simple, even if one thing is extremely important here. Before the forager is stored the collected information must be evaluated, leading to a amount of dancing. The number of dances is the base for the recruitment of new foragers. So if it's not determined very good, all routing decisions may lead to wrong or at least bad decisions.

For every type of forager a own evaluating function is existing. The first thing this function does, is judging the quality of the route through the information, which the forager has collected on its flight. So, a forager which likes to optimize the network lifetime for example will somehow look at the mean battery level, the minimum battery level on that route and also the total route length. If these values are good the evaluating function will return a high value, otherwise a low one.

After the judgment of the routes quality, the value is scaled by looking at the number of waiting packers. If very many packers are waiting, a low quality route may as well lead to relative high dance numbers. Because many packets are waiting it's possible that no better routes are available and they must be sent using a less good route. The other way around, when no packers are waiting it's also possible that a forager with a very good route and a good evaluation doesn't dance at all, because many other foragers are

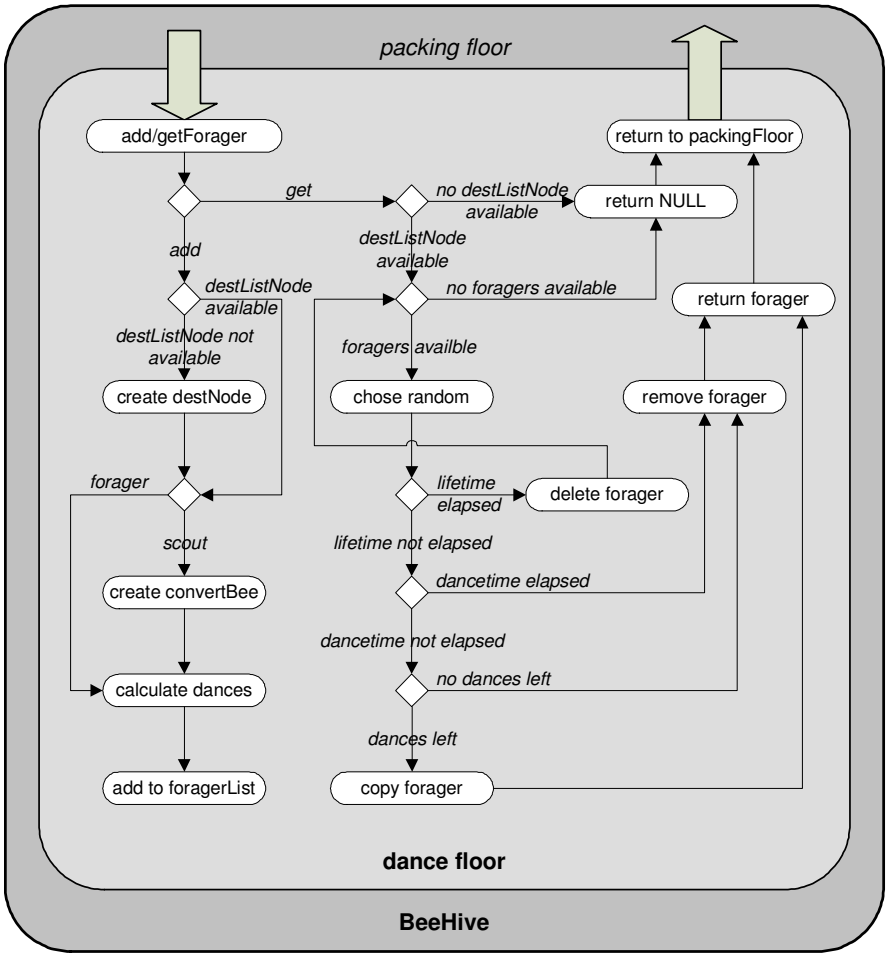


Figure 2.4.: The dance floor

available. This dancing and recruiting procedure should copy the natural example of the waggle dances, see 1.4.2. This mechanism is responsible for adjusting the capacity (the number if available foragers) for each route.

The last paragraph has shown the importance of the dance numbers. The evaluation functions calculating them are therefore very important. They should of course prefer good routes. Additionally, they should converge against a bound, so that there aren't too many foragers lying around in the dance floor being of no use. But they should have enough clearance to allow the replacement of already available foragers by newer ones, because it is possible that available routes become much worse with time and vice versa. Last, they shouldn't be too costly to calculate because they are performed for every returning foragers and con potentially waste a lot of computing time.

If a forager is demanded by the packing floor to send the data of a waiting packer, the dance floor looks for a matching one. The first criterion is of course the destination and afterwards the optimization criterion. If available, a foragers is chosen randomly of all matching foragers. If not, a alternative forager with another optimization is returned.

The chosen foragers is then examined with respect to its age and the dance number. If it's older than the specified life time of a forager it's deleted and BeeHive will choose another one. This corresponds to becoming disloyal foragers in nature (compare to chapter 1.2). If the forager is so young that the specified dance time hasn't expired, the dance number is examined. Dance numbers greater than zero will lead to a copy of that forager which is forwarded to the packing floor while the original forager is stored in the dance floor again with a decreased number of dances. If the dance number is zero, the forager itself is forwarded to the packing floor and completely deleted from the dance floor.

In terms of routing in ad hoc networks this mechanism means a potentially extension of good routes which are quite young and accordingly should still exist. Good routes should be considered by this mechanism, because they will lead to high dance numbers and can be copied often. When they are copied often, plenty of foragers for this good route exists in the dance floor, once they've returned from their flights. If there are many of them they will amount to a great fraction of the available foragers and hence it is very probable that they are chosen randomly of all the available foragers. So, good routes will take a great load of the transmitted data.

If the last forager is leaving, BeeHive has no routing information to this destination anymore and must wait for returning bees. At first view, this might seem quite stupid. But by really deleting the foragers from the dance floor, an external route maintenance is obsolete. If a forager leaves the hive and doesn't return, it might be an indication that

this route is broken. Otherwise, a returning forager guaranties a working route. Only in the beginning phase of a link that might lead to increased delay because the number of foragers is to small. After this phase there should always be enough foragers inside the hive, guarantying an immediate transport of data packets.

As foragers with a broken route are lost, it's necessary that higher layers are aware of that. If the data is absolutely necessary for an application, a higher layer like TCP must take care of this. On the other hand, BeeHive only tries to sent packets over a broken route as long as it still has foragers, so the loss will be small.

2.4. Conclusion

It was tried to follow the natural example as close as possible, although there had to be made some concessions because of the obvious differences between natural honey bees and a wireless ad hoc network and the traffic on it. But even then the result is a quite competitive algorithm for these networks, at least as far as the results of the reference implementation are showing (see chapter 5). At the end the main advantages and disadvantages of BeeHive are repeated.

2.4.1. Advantages

- The most important advantage of BeeHive is the distribution of the traffic to different routes proportional to their quality and capacity. This is done by a very simple mechanism, without wasting much computing time and energy.
- The second great advantage is the absence of many control packets compared to other algorithms. The control messages are limited to the scouts which obviously are necessary and the swarms for not balanced bidirectional or even unidirectional communication links.
- Another point is the abdiction of global information, which should be self understood normally. The is no such global information needed, all decisions are taken locally.
- As the BeeHive algorithm doesn't take advantage of any information provided by other layers, it is completely independent from these and should work with all underlying and above protocols, except the small assumption from 2.3.2.

2.4.2. Disadvantages

- The most important disadvantage of BeeHive is the use of source routing. This also appears quite unnatural compared to real bees, because they use a vector guidance. But even they remember special landmarks on their flights. The disadvantage in computer networks comes from the control overload per packet and the limitation of the maximal route length.
- Another real disadvantage is the higher memory use for storing every forager. Although they are really small it is more than storing every route only once.
- The artificial limitation to available foragers can appear as a disadvantage because packets are not sent, respectively must wait some time even if a route is known because all matching foragers have left the hive for a flight. But as BeeHive needs this behaviour for its functioning and it also has advantages it seems to be acceptable. This is indeed true if the simulation results (see chapter 5) are taken into account.

2.5. Psedo Code

BeeHives code is quite large even if the algorithm is quite simple. To provide a overview of BeeHives working, a short pseudo code description is provided here, corresponding to figures 2.1 to 2.4.

2.5.1. BeeHive

```
BeeHive::send(packet) {
  packingFloor::send(packet);
}

BeeHive::receive(packet) {
  entrance::letIn(packet);
}
```

2.5.2. Entrance

```
Entrance::letIn(packet) {
  buildBeeFromPacket();
  if(forager) {
```

```
  if(arrived)
    packingFloor::receive(forager);
  else {
    updateInformation(optimization);
    buildPacketFromForager(forager);
    MAC::send(packet);
}}
if(scout) {
  if(returningScout) {
    updateNextHop();
    buildPacketFromScout(scout);
    MAC::send(packet);
  }
  if(scouting) {
    if(arrived) {
      reverseFoundRoute();
      updateNextHop();
      buildPacketFromScout(scout);
      MAC::send(packet);
    }
    else {
      if(danceFloor::getForager(destination)!=NULL) {
        completeRouteFromForager();
        reverseFoundRoute();
        updateNextHop();
        buildPacketFromScout(scout);
        MAC::send(packet);
      }
      else {
        if(TTL==0)
         delete scout;
        else {
          if(isInSeenScoutList(source, ID)
            delete scout;
          else {
            addToSeenScoutList(source, ID);
```

```
              insertOwnAddress();
              buildPacketFromScout(scout);
              MAC::broadcast(packet);
}}}}}}}}

Entrance::letOut(bee) {
  if(scout) {
    addToSeenScoutList(source, ID);
    insertOwnAddress();
    buildPacketFromScout(scout);
    MAC::broadcast(packet);
  }
  if(forager) {
    updateInformation(optimization);
    buildPacketFromForager(forager);
    MAC::send(packet);
}}
```

2.5.3. Packing floor

```
PackingFloor::send(packet) {
  buildPackerFromPacket(packet);
  if(danceFloor::getForager(destination, optimization)!=NULL) {
    handDataFromPacketToForager(packer, forager);
    delete packer;
    Entrance::letOut(forager);
  }
  else {
    addPackerToWaitingPackers();
    createScout(destination, initialTTL, ID);
    setScoutTimer(initialTTL*delay);
    entrance::letOut(scout);
}}

PackingFloor::scoutTimerExpired(destination, lastTTL) {
  if(danceFloor::getForager(destination) == NULL) {
    createScout(destination, newTTL, newID);
```

```
    setScoutTimer(initialTTL*delay);
    entrance::letOut(scout);
}}

PackingFloor::receive(bee) {
  if(forager) {
    passDataToTransportLayer(data);
    updateWaitingPackers(destination)
    danceFloor::addForager(forager);
  }
  if(scout) {
    updateWaitingPackers(destination)
    danceFloor::addScout(scout);
  }

  while(waitingPackers>0 && danceFloor::getForager(destination)!=NULL) {
      handDataFromPacketToForager(packer, forager);
      delete packer;
      Entrance::letOut(forager);
}}
```

2.5.4. Dance floor

```
DanceFloor::addBee(bee) {
  if(destListNode(destination)==NULL)
    createDestListNode(destination);
  if(scout)
    createForagerFromScout(scout);
  calculateDances(forager);
}

DanceFloor::getBee(destination, optimization) {
  if(destListNode(destination)==NULL)
    return NULL;
  else {
    forager=NULL;
    if(foragerAvailable(destination) {
```

```
      while(forager==NULL && foragerAvailable(optimization)) {
        chooseForager(random);
        if(lastFlight+lifeTime<now) {
          if(lastFlight+danceTime<now && danceNumer>0)
            forager=copyForager();
          else
            forager=removeForagerFromDanceFloor();
        }
        else
          delete forager;
      }
      forager=getYoungestForager(destination);
    }
    return forager;
}}
```

3. Implementation of Beehive in ns-2

By Christian Mueller (christian.mueller@uni-dortmund.de)

3.1. Introduction

As the development and evaluation of routing protocols with real world hardware is expensive and very difficult to do, we made heavy use of network simulations. A key to successfully achieve the desired results in reality, is to have a good simulation model of the targeted environment. Several network simulators are available, such as ns-2 [ns2] and Omnet++ [omn]. They aided us in evolving the Beehive routing algorithm.

3.1.1. History

The concrete development of the Beehive algorithm began using the Omnet++ simulator together with AdHocSim [adh]. It provided the necessary tools to implement the basic version of Beehive and to test various enhancements like scout and forager caching. Unfortunately at the end of the first semester it became clear, that we could not get reliable results in terms of a real world usage. Our competitors at this point of time were AODV and DSR. The biggest Problem was the framework itself, which wasn't developed to fully implement the various layers, like a compliant 802.11 MAC, a full TCP/IP stack or the physics behind wireless connections.
Luckily enough the design of the already written code allowed us to switch the simulator without rewriting everything, since a single class (beehive.cc) handles nearly all simulator specific details.
In the beginning of the second semester the decision was made to use ns-2 for further evaluation. Although it was not that trivial to convert the Beehive code, due to a lack of documentation about ns-2 routing internals (the ns-2 manual gets overhauled at the time of this writing), the final result was a nearly complete simulation environment with tested algorithms and methods.

3.1.2. Ns-2

> Ns is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. [ns2]

The features of ns-2 regarding our simulation are:

- extensive TCP/IP stack (FullTCP, similar to a 4.x BSD stack)
- various traffic generators (CBR, VBR, FTP, HTTP, stochastic)
- Visualization of nodes and data flows [nam]
- Mobile Nodes with programmable trajectories
- Complete implementation of the IEEE 802.11 DCF MAC protocol
- Complete implementation of the Address Resolution Procotol (ARP)
- Implementation of DSR, DSDV, TORA and AODV
- Wireless network interface modeling the Lucent WaveLAN DSSS radio
- Modeling of signal attenuation, collision, and capture
- Two Ray Ground Reflection radio propagation model
- Simple energy model
- The scripting language OTcl to set-up the scenarios

3.2. OTcl

Ns-2 makes a twofold approach: the core is written in C++ for speed and efficiency, while the scenario configurations are described in OTcl [otc]. Furthermore the complete class hierarchy is available to both parts, which lets the user easily manipulate aspects of C++ objects in OTcl, or extend the simulator with rapidly written scripts, with the penalty of execution speed and memory requirements. Using a programming language to control the simulation eases the set-up in complex cases, but a little effort has to be done to glue both worlds together.

A short (non-working) TCL batch script to run a Beehive simulation may look like this:

```
set val(rp) Beehive
Agent/Beehive set VERBOSE 0
set ns_ [new Simulator]
set tracefd [open "$val(rp).tr" w]
$ns_ use-newtrace
$ns_ node-config -adhocRouting $val(rp)
$ns_ run
```

(This batch script won't work, since we have to set a lot more options, see tcl/beehive/s-cenario/results.tcl for reference.)
After setting up variables (line 1), an option is set in the Beehive Agent. Line 3 instances the global ns object, the simulator itself. The next line opens a file descriptor (the last part of the command) and binds it to the variable tracefd. In line 4 ns is told to generate a trace file in the new trace format, as shown in chapter (parsing a tracefile). The following line is one command, which sets up the mobile node framework. Finally the event scheduler starts and the simulation is launched. The command `ns sample.tcl` results in a tracefile called Beehive.tr, a NAM visualization file called Beehive.nam and a lot of debugging messages on stdout (and hopefully nothing on stderr). The tracefile can be analyzed with `parser.pl < Beehive.tr` and the scenario can be seen with executing `nam Beehive.nam`.

3.3. Scenario generation

Although there are tools to generate movement and traffic patterns, we decided to use ns-2's scripting abilities. The tools we found generated incorrect files or were just not usable. Since our goal was not to test out various movement patterns but to evaluate routing algorithms, the simple but generally applicable random waypoint method was used. Our implementation is straightforward and written into the TCL scripts. We made use of the fact, that ns-2 has a built-in PRNG (pseudo random number generator), which will generate the same sequence of numbers when fed with the same seed. The PRNG is independent on the machines hardware and the OS, so that a run with the same seed and options is exactly reproducible. Furthermore is ns-2 capable of instancing several independent PRNG streams, so that we do not have to take care if other parts of the simulator would disturb the contingency of the sequence. This sequence is used to initially position every node to a (pseudo) random (x, y) value. After that a function is called for every node, which does the following:

```
proc move_node {n} {
```

```
(...)
#set new x-value destination
set rnd_pos_x [expr [$rnd value] * $val(x)]
#set new y-value destination
set rnd_pos_y [expr [$rnd value] * $val(y)]
#the speed with which the node should move, \
# in the range between rwpmin and rwpmax
set rnd_speed [expr [$rnd value]]
set rnd_pos_s [($rnd_speed * $val(rwpmin)) + \
 ((1 - $rnd_speed) * $val(rwpmax))]
#execute the movement
$ns_ at [$ns_ now] "$node_($n) setdest \
 $rnd_pos_x $rnd_pos_y $rnd_pos_s"
#wait 0 to rwppause seconds and start movement again, \
# eventually interrupting an ongoing movement
$ns_ at [expr [$rnd value] * $val(rwppause) + \
 [$ns_ now]] "move_node $n"
}
```

The rnd variables are uniformly distributed floating point numbers between 0.0 and 1.0.

To generate traffic we used the built-in CBR traffic agent over full TCP in most of our evaluations. FTP traffic generation and UDP were also tested.
The nodes are connected in the following way: The first node 0 sends its data to the last node n, the second node 1 to n-1 and so on, up to m nodes generating the traffic. In our results m was set equal to n. This scenario ensured that we did not generate any artificial hotspots, equally to a peer-to-peer network. That has the advantage of a comparable bandwidth of every node and a standard deviation which is meaningful concerning the routing itself.

3.4. Beehive implementation

The main implementation of beehive consists on the following files:

- b_beeDefinitions.h, defines constants and the structures of the agents. Most of the constants have been made available to the TCL hierarchy as variables for convenience.

- hdr_beehive.cc .h, links the beehive packet header into the ns2 structures that is the source routing and IP options data fields
- beehive.cc, the main class, responsible to handle most of the simulator specific details to enable easy porting of beehive
- beehive.h, beehive's global definition file with data structures based on STL [stl] and class definitions
- beehive.tcl, sets the TCL variables of Beehive to default values
- b_entrance.cc, the entrance, which is explained in detail in chapter 1
- b_packingfloor.cc, the packingfloor, see above
- b_beefloor.cc .h, the dancefloor, see above
- b_gridfloor.cc .h, an alternative dancefloor based on graphs, explained in [PGb] (unused)

To integrate Beehive into ns-2, some sources of the simulator had to be expanded, with mostly one line changes:

- Makefile.in, of course our sources should be compiled too
- common/packet.cc, adds the Beehive header, defined in hdr_beehive to the core
- queue/priqueue.cc, to add the prefer routing protocols flag, although unused in our simulations
- tcl/lib/ns-lib.tcl, this adds the Beehive routing agent to the TCL hierarchy
- tcl/lib/ns-packet.tcl, which makes our packet header available

Furthermore a few bug fixes (taken from the mailing lists) have been applied to aodv/aodv.cc, mac/mac-802_11.cc and tora/tora.cc.

The class beehive.cc provides, as mentioned, an interface to the simulator. It instantiates upon creation the packetfloor, entrance and dancefloor, and binds the TCL variables. Beehive::command accepts the TCL events send by the core, as setting the local address per node, or attaching objects like the port demux, but most of the events are handled by beehive's parent class, the ns-2 agent. Beehive::recv takes a packet as an argument and forwards it, depending on the header, to handleFromApp, handleScout or

handleForager. These and other functions in beehive.cc are helper functions to transform ns-2 packets in beehive agents and vice versa. Since these are simple pointer operations or API specific details (or voodoo because of the lack of documentation about ns-2's internals), there is no reason to explain this further. Finally the packets will traverse into the packingfloor, if they are to be sent; into the entrance, if they are coming from the net; or send/broadcasted out into it.

One detail of the simulation is noteworthy: a problem arose, when we broadcasted our scouts into the neighborhood, a fast queue built up was experienced. Since the transmitting time and delays between the wireless nodes are exactly the same when no problem occurs, all neighbor nodes broadcasted their scouts again at the same time, which led to a massive collision of the scouts. This would not happen in reality, so we added a small random jitter upon receiving and forwarding a scout on every node. It seems that the DSR implementation does the same.

4. Parsing a tracefile

By Rene Jeruschkat (rene.jeruschkat@uni-dortmund.de)

4.1. Introduction

After implementing our routing algorithm the main question was how to compare all those algorithms available in ns-2? Getting results only for beehive would be a rather easy task as we did it before in omnet. We would modify some of our classes and gather statistics during execution time. But this approach would require us to modify all other algorithms. These algorithms look as if they all were coded by different people although most of them were implemented within the same university project. There is no unified approach, no framework, no general picture where to start editing these algorithms. Even proper comments were missing, variable names are very often cryptic shortcuts and documentation as usual in ns-2 is really poor. (That doesn't go for using ns-2, but it's definetely true for editing it)

After complaining so much, there was obviously another solution and that's the one we followed: All those algorithms produce tracefiles while executing. These tracefiles have a common structure and contain a lot of information. They don't contain real performance indicators though. This is where the parser comes into play.

4.2. Tracefile size / Simulation sequence

Within simulation ns-2 creates tracefiles roughly 200MB in size. In code editing cycles the usual approach is executing the algorithm, parsing the tracefile and see whether its an improvement or not. These two steps executing and parsing can't be split up in a large simulation. We are simulating round about 1000 different network conditions and therefore this usual sequence is not possible anymore due to diskspace limitation.

Our first attempt to solve this little problem was to compress the output everytime one of the 1000 simulation runs completes. Zipped files are one order of magnitude smaller than their original textfile counterparts. It is still too much and therefore no

proper solution. The most important fact to skip the compression is the attribute of ns-2 to write its output chronologically. Enabling the parser to read from standard input (STDIN) solves the space problem entirely as every tracefile is deleted after parsing ns-2 output (extracted information contains everything needed).

4.3. Parser usage

Given a typical tracefile Beehive-40-50000.tr (Algorithm Beehive, 40 nodes, 50000: seed for random number generator) a parser call would look like

```
./parser.pl < Beehive-40-50000.tr
```

space efficiency can be improved by reading from STDIN and passing the output from ns-2 directly into the parser. Trace files aren't generated any more. Therefore the real parser call is inside a ns-2 specific tcl script which is responsible for starting all of our simulations. A tcl-parser call looks like

```
set tracefd [open "|../parser.pl > \$val(filename).txt" w]
```

4.4. Tracefile structure

So, what is this tracefile all about? To explain the tracefile structure we cut out a small part of a real file, cleaned it up and made some changes to improve readability. Leading # lines are comments made by us and are not inherent in any tracefile. Take a look at the legend to get all those abbreviations in figure **??** explained.

4.4.1. Legend

```
s       alternative of [s,r,f,d] = send,receive,forward,dropped
-t      current time
-Ni     current node ID
-Ne     current batterylevel
-Nl     networklayer [AGT,RTR,MAC] = Agent, Routing, MediumAccessControl
-Nw     in this example always "---" sometimes collisions are reported
        here by COL=collisions
-Mi     packettype as seen by MAC [ack, RTS, CTS, beehive, DSR, ...] =
        Acknowledge, RequestToSend, ClearToSend, beehive, DSR, and other
        routing algorithms
```

```
-Uid    UniqueId for following packetflow
-Md     MacDestination the destination the MAC wants the packet to send to
-Is     IPsource A.B - A is the initiatial node, B shows the port used to
        send the packet
-Id     IPdestination A.B - A is nexthop, B the port (similar to -Is)
-It     IPtype similar to -Mi
-Il     packetsize in bytes
```

```
  #node 1001
s -t 46.046121273 -Ni 1001 -Ne 98.923240 -Nl AGT -Nw --- -Mi ack -Uid 3709 -Md 0 -Is 1001.1 -Id 1008.0 -It ack -Il 40
r -t 46.046121273 -Ni 1001 -Ne 98.923240 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 0 -Is 1001.1 -Id 1008.0 -It ack -Il 40
s -t 46.046121273 -Ni 1001 -Ne 98.923240 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 0 -Is 1001.1 -Id 1001.255 -It ack -Il 104
s -t 46.046620273 -Ni 1001 -Ne 98.923240 -Nl MAC -Nw --- -Mi RTS -Uid 3709 -Md 1002

#establish connection 1001<->1002
r -t 46.046972505 -Ni 1002 -Ne 98.998550 -Nl MAC -Nw --- -Mi RTS -Uid 3709 -Md 1002
s -t 46.046982505 -Ni 1002 -Ne 98.998550 -Nl MAC -Nw --- -Mi CTS -Uid 3709 -Md 1001
r -t 46.047286737 -Ni 1001 -Ne 98.923125 -Nl MAC -Nw --- -Mi CTS -Uid 3709 -Md 1001
s -t 46.047296737 -Ni 1001 -Ne 98.923125 -Nl MAC -Nw --- -Mi ack -Uid 3709 -Md 1002 -Is 1001.1 -Id 1001.255 -It ack -Il 156

#node 1002
r -t 46.048544969 -Ni 1002 -Ne 98.998278 -Nl MAC -Nw --- -Mi ack -Uid 3709 -Md 1002 -Is 1001.1 -Id 1001.255 -It ack -Il 104
r -t 46.048569969 -Ni 1002 -Ne 98.998225 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 1002 -Is 1001.1 -Id 1001.255 -It ack -Il 104
f -t 46.048569969 -Ni 1002 -Ne 98.998225 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 1002 -Is 1001.1 -Id 1002.255 -It ack -Il 104
s -t 46.049228969 -Ni 1002 -Ne 98.998225 -Nl MAC -Nw --- -Mi RTS -Uid 3709 -Md 1003

#establish connection 1002<->1003
r -t 46.049581474 -Ni 1003 -Ne 98.259148 -Nl MAC -Nw --- -Mi RTS -Uid 3709 -Md 1003
s -t 46.049591474 -Ni 1003 -Ne 98.259148 -Nl MAC -Nw --- -Mi CTS -Uid 3709 -Md 1002
r -t 46.049895979 -Ni 1002 -Ne 98.998110 -Nl MAC -Nw --- -Mi CTS -Uid 3709 -Md 1002
s -t 46.049905979 -Ni 1002 -Ne 98.998110 -Nl MAC -Nw --- -Mi ack -Uid 3709 -Md 1003 -Is 1001.1 -Id 1002.255 -It ack -Il 156

#node 1003
r -t 46.051154483 -Ni 1003 -Ne 98.258877 -Nl MAC -Nw --- -Mi ack -Uid 3709 -Md 1003 -Is 1001.1 -Id 1002.255 -It ack -Il 104
r -t 46.051179483 -Ni 1003 -Ne 98.258824 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 1003 -Is 1001.1 -Id 1002.255 -It ack -Il 104
f -t 46.051179483 -Ni 1003 -Ne 98.258824 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 1003 -Is 1001.1 -Id 1003.255 -It ack -Il 104
s -t 46.051778483 -Ni 1003 -Ne 98.258824 -Nl MAC -Nw --- -Mi RTS -Uid 3709 -Md 1004

...

#establish connection 1007<->1008
r -t 46.070580817 -Ni 1008 -Ne 98.317063 -Nl MAC -Nw --- -Mi RTS -Uid 3709 -Md 1008
s -t 46.070590817 -Ni 1008 -Ne 98.317063 -Nl MAC -Nw --- -Mi CTS -Uid 3709 -Md 1007
r -t 46.070895445 -Ni 1007 -Ne 98.115469 -Nl MAC -Nw --- -Mi CTS -Uid 3709 -Md 1007
s -t 46.070905445 -Ni 1007 -Ne 98.115469 -Nl MAC -Nw --- -Mi ack -Uid 3709 -Md 1008 -Is 1001.1 -Id 1007.255 -It ack -Il 156

#node 1008
r -t 46.072154072 -Ni 1008 -Ne 98.316792 -Nl MAC -Nw --- -Mi ack -Uid 3709 -Md 1008 -Is 1001.1 -Id 1007.255 -It ack -Il 104
r -t 46.072179072 -Ni 1008 -Ne 98.316739 -Nl RTR -Nw --- -Mi ack -Uid 3709 -Md 1008 -Is 1001.1 -Id 1007.255 -It ack -Il 104
r -t 46.072179072 -Ni 1008 -Ne 98.316739 -Nl AGT -Nw --- -Mi ack -Uid 3709 -Md 1008 -Is 1001.1 -Id 1008.0 -It ack -Il 40
```

This "small" extract describes the whole communication needed to send a single packet from node 1001 to node 1008. The agent at node 1001 initializes the transmission by deciding to send a packet to node 1008. This decision is communicated to the Routing layer (line number #2) which receives(#3) the wish, forwards it to the mac layer(#4) which tests the availability of the transportation medium by sending a RequestToSend(#5) message. Nexthop 1002 receives the call(#8) is ready and sends a ClearToSend packet(#9) back to the origin. Node 1001 receives the message(#10) and backs it up by sending an acknowledge(#11). Finally node 1002 receives(#14) the ack and completes the handshake. Node 1002 examines the packet, forwards it to node 1003(#16) and another handshake takes place. Even more nodes forward the packet until finally node 1008 is reached. The last handshake between node 1007 and 1008 provide the medium and presents the data to the destination agent.

Above of this example we mentioned that we cleaned the extract up. The reason for that lies in another fact we have already stated: Chronological writing in ns-2. Since ns-2 has to keep track of many connections simultaneously therefore overlapping is a natural result and causes headaches reading output unformatted. In the example only one connection is present, therefore obviously no interleaving is possible. Connection trackability is achieved by assigning unique IDs to each of them - packets get distinguishable. Considering parsing the tracefile that comes in handy.

4.5. Parsing

Analyzing line by line the parser stores valuable information needed for future calculations, ignoring and forgetting redundant or unnecessary data.

As seen in the example the most important lines are the ones containing AGT information. These initiate and close connections. In the first case a data structure that is supposed to track connections is tested against the parsed unique ID. As every connection start occures only once, a positive test indicates a corrupt tracefile (should never happen). Usually there is no problem here and data gathered from this AGT start line is inserted into the data structure indexed by the unique ID. Following information is stored: The amount of data sent, the current time in tracefile (needed for delay), the number of hops the route has seen until now (thats always 0) and the energy needed to send this information.

When reading another line containing this unique ID the associated record in the data structure is updated. Different data categories have different update frequencies. The delay for example is updated at the concluding AGT receive, intermediate RTR

and MAC layers have no effect on it. It is calculated as receive time (current time) subtracted by send time (gathered at initiation). Hop count and energydata have to be updated more often, RTR layer information affect them directly. Every occurence of a unique ID in a RTR layer increases the corresponding hop count by one. That's of course quite intuitive, energy on the other hand is a little more complex. Energy is influenced by the packet size and the hopcount: Some constant and some value proportional to the packet size has to be added every hop. One could assume it is possible to multiply the constant with hopcount and add proportionalized packetsize at the finalization of the connection. That's not the case though. Packetsize varies over time as sourcerouting implies increasing headersize at every node visited.

Once the Agent receives its data all important facts can be evaluated and stored in different overall data structures. On this receive, the delay is calculated, the hopcount finalized and so is the Send- ReceiveEnergy. Most of these facts will be stored in different lists each containing one criteria. At the end of every simulation a statistic method is called upon each list giving back the average, minimum, maximum, standard deviation and sum of the elements. Although not all of these calculations make sense for all criteria nevertheless it's no problem to calculate them. Therefore data can be handled uniformly and one method fits them all.

Erasing the connection record right now ends this connection. All relevant data was already analyzed or collected for later analyzation, there is no more need for it.

4.6. Throughput remarks

There are two ways handling throughput, the common (in our opinion rather poor) one is to divide the amount of received data by the simulation time. This ratio is dominated by the amount of bytes received, since simulation time is fixed for all competing algorithms. Bytes received is for sure an important criteria to measure an algorithms performance but doesn't really apply to the term throughput as used in common applications.

Let's take a look at an example: Assuming a simulation is set to 10 seconds simulation time and two different algorithms competing. Both algorithms should deliver 10 MB from node A to node B. For simplicity reasons let's assume there is no packetloss and both will deliver 100% of the data. Let's further assume the first algorithm needs 1 second to deliver the 10 MB to node B and the second algorithm needs 10 seconds for the same task. Obviously the first algorithm finishes its task faster than the second one and should therefore be rewarded with a higher throughput. Nevertheless both algorithms get a rating of 1 MB/s.

Our suggestion is to ignore the idle time and to reward the first algorithm with a throughput of 10 MB/s while the second one is one order of magnitude slower. Therefore the parser calculates both values "throughput(simTime)" which is the common one - "Throughput" the min,max,average, stddev and sum of throughput a single connection has.

4.7. Parser output

typical output:

100s simulation time,
40 nodes,
Beehive,
100s network lifetime,
5.8814 µJ/kB,
217.32 kbit/s throughput(simTime)

Optimization Criteria:	MEAN	MIN	MAX	STDDEV	SUM
Data:					
sendDataEnergy (µJ)	2096.189	496.000	25054.400	3131.595	11359250.400
receiveDataEnergy (µJ)	801.480	346.800	7909.920	1027.716	4142848.320
Ctrl:					
sendCtrlEnergy (µJ)	366.006	310.800	556.800	80.348	919406.000
rcvCtrlEnergy (µJ)	88.586	72.000	360.240	58.116	857954.640
sendCtrlP2PEnergy (µJ)	524.558	488.400	556.800	25.008	257558.000
rcvCtrlP2PEnergy (µJ)	353.558	345.120	360.240	5.433	156272.640
sendCtrlBCEnergy (µJ)	327.485	310.800	379.200	16.763	661848.000
rcvCtrlBCEnergy (µJ)	75.915	72.000	90.000	4.426	701682.000
Etc:					
batteryLevel (%)	95.773	93.713	99.457	1.621	3830.903
Delay (ms)	114.915	2.000	4013.297	226.619	590776.328
Throughput(Kbit/s)	140.718	0.082	834.437	218.660	723429.489
Hopcount	1.417	1	10	1.603	7284

		SENT volume (number)		RECEIVED volume (number)		RATIO (rcvd/sent) volume (number)		LOST volume (number)		DROPPED volume (number)	
APP:	Wish	2929kB (	5419)	2716kB (	5141)	92.7% (	94.9%)	213kB (	278)	NaN (	NaN)
	Data	2924kB (	5246)	2865kB (	5141)	98.0% (	98.0%)	59kB (	105)	NaN (	173)
ROUTE:											
	NetData	4181kB (	7481)	4122kB (	7376)	98.6% (	98.6%)	59kB (	105)	NaN (	NaN)
	NetCtrl	106kB (	2512)	383kB (	9685)	359.0% (	385.5%)	NaN (	NaN)	NaN (	NaN)
	- P2P	26kB (	491)	24kB (	442)	91.8% (	90.0%)	2kB (	49)	NaN (	NaN)
	- BC	80kB (	2021)	359kB (	9243)	446.7% (	457.3%)	-279kB (	-7222)	NaN (	NaN)
MAC:											
	NetSum	4288kB (	9993)	4505kB (	17061)	105.1% (	170.7%)	NaN (	-7068)	196kB (	1314)

Figure 4.1.: some typical parser output

5. Simulation Results

By Johannes Meth (J.Meth@landata.de) and Björn Vogel (BjoernVogel@gmx.de)

5.1. Introduction

This chapter deals with the evaluation and simulation of the developed algorithm. The BeeHive algorithm was developed with the intention to be used in Wireless Networks with up to 200 nodes. Since it takes enormous efforts and costs to evaluate such large networks in reality it was decided to do simulation and evaluation with a software-simulator. In the early days of algorithm development the omnet++ simulator was used to test the fitness of the algorithm. So some early results were evaluated with the use of omnet++ but soon it turned out that this simulator does not have the capability to reflect a real wireless environment in a satisfactorily manner. So in the final phase the algorithm was migrated to the ns-2 simulator which has more realistic ways to simulate an environment (see chapter 3 of Part I).

5.1.1. Tested algorithms

Beside BeeHive there were simulated some more algorithms which are the following:

AODV - On-Demand Distance Vector Routing Protocol

The AODV protocol is a next-hop routing algorithm therefore every node has its own routing-table [CEPD02]. This table is updated on demand only. When a source node desires to send a message to some destination node and does not have a valid route to that destination the route discovery is as follows: The Source S sends a Route-Request (RREQ) as a kind of broadcast to all its neighbours. These neighbours forward the RREQ to all reachable nodes, until the destination D has been reached. Has the RREQ packet arrived at its destination, the destination node sends a Route-Reply packet back to the source. If an intermediate node on the way of an RREQ messages has an up-to-date route to the destination, it could also sent an RREP on behalf of the destination. This RREP packet uses the reverse path of the RREQ message. Nodes along this

reverse path set up forward route entries in their route tables which point to the node from which the RREP came. If no RREP packet arrives at the source node after a fixed period of time, the node repeats sending an RREQ packet with a higher TTL. In case of this RREQ also not being successful the destination is marked as unreachable. AODV utilizes destination sequence numbers to ensure all routes are loop-free and contain the most recent route information. Each node maintains its own sequence number, as well as a broadcast ID. The broadcast ID is incremented for every RREQ the node initiates, and together with the node's IP address, uniquely identifies an RREQ. Along with its own sequence number and the broadcast ID, the source node includes in the RREQ its most recent sequence number for the destination. Intermediate nodes can reply to the RREQ only if they have a route to the destination whose corresponding destination sequence number is greater than or equal to that contained in the RREQ.
The route maintenance process is as follows: When a node detects a link failure, it sends a Route-Error message (RRER) to each neighbour for which it is forwarding traffic through the link. This RRER is thus propagated to each source for which traffic is being routed through the link failed link, causing if necessary the route discovery process to be reinitiated. An additional aspect of the protocol is the use of `hello` messages. These messages are periodic local broadcasts to inform each mobile node about other nodes in its neighbourhood. This can help the nodes to maintain there local connectivity's and detect failed links fast.

DSR - Dynamic Source Routing Algorithm

The DSR protocol [DBJH04] is an on-demand routing protocol that is based on the concept of source routing. The route discovery process of DSR is much similar to the behaviour of AODV. The source sends an RREQ packet via broadcast to all reachable nodes. These nodes add their node IDs into the route header and forward the packet to all its neighbour nodes. If the packet reaches the destination the header contains a complete route to the source, which is inserted in the route header of the RREP. To avoid loops every node checks the route header of the RREQ packet for its own node ID. If the own node ID is found the packet has already been forwarded by this node and has to be discarded. When the source receives a route reply, it caches the source route and includes it in the header of each data packet. Intermediate nodes forward the packet according to the route specified in the header and also cache the route of the route header in their route cache. The nodes are responsible for the accurate transmission of a packet. Therefore acknowledges are send from every node after receiving a data or RREP message. If a node does not receive an acknowledge after sending a

packet, the node will resend the packet. Route error packets are generated at a node when the data link layer encounters a fatal transmission problem. When a route error messages is received, the hop in error is removed from the node's route cache and all routes containing this hop are truncated at that point.
Many optimizations for DSR passed on aggressive caching and analysis of topology information are incorporated into this scheme. From the source route included in each data and RREP packet, each intermediate node can trivially extract routes to all downstream nodes. Additional topology information can be reduced by combining information about several routes. Further information can be obtained by nodes operating their network interface in promiscuous mode. In this mode a node can overhear the transmitted packets between its neighbours and can additionally add the routes of these packets to its route cache. This aggressive caching can lead to a high cache hit rate that reduces the expensive route discovery and finds routes more quickly. On the other hand it can also increase the risk of stale route information being injected into the network. An advantage of the DSR protocol compared to AODV is that the route cache can contain multiple routes to one destination and that it uses much less messages for network maintenance. On the opposite the demand of memory in the packet for route header and in the nodes for the route cache is a lot higher than for AODV.

DSDV - Destination-Sequenced Distance-Vector

In the opposite to the on-demand DSR and AODV protocols the DSDV algorithm is a proactive table-driven protocol in which every node has its own routing table and the nodes exchange there information about active routes via messages to update these tables. Each node maintains the following information in its routing table [Fee99]:

- next hop towards each destination
- a cost metric for each destination
- a destination sequence number that is created by the destination itself (to detect stale routes form new ones)
- a new sequence number unique to the broadcast (to avoid loops)

Every node periodically sends its routing table to its neighbour. Meanwhile the node increments and appends its sequence number. This sequence number will be attached to route entries created for this route. Each receiving node compares the broadcast sequence number for each destination with the one in its routing table. If the sequence

number is higher, the receiver updates its routing table entry, naming the sender as the next-hop and incrementing the distance by one hop. If the sequence numbers are equal, the route with the smaller metric is used to shorten the path. When a link failure is detected by a node the distance to each destination via this node is set to infinity and the sequence number are all incremented.
To reduce network traffic there are two ways of broadcasting the routing table. The first is known as a "full dump" and contains the full routing table. The other one is the "incremental" update that is used to relay only the information's which has changed since the last "full dump". The mobile nodes maintain a separate table in which the "incremental" updated information is stored.

5.2. Simulation Runs

To compare all these algorithms with each other, a testing environment had to be created in which the starting position and conditions are the same for all algorithms. This could be done through creating a simulation environment in ns2. But testing these algorithms in one simulation case (or test case) only can provide very specific results, so, to get more "across the board" results, many different test cases were created as shown in tables 5.1 and 5.2.

The amount of nodes in the TCP-simulations was set to 50 (UDP: 30) with each of this nodes acting as a sending node. The amount of simulated seconds was set to 1000 for every simulation run. Each of the runs was simulated five times with different seeds, which result in a slightly changed order of events but keeping the general simulation case equal. These five results for each run were merged together to build an average case, which was then used for comparisons. Actually we simulated many cases more, but the ones listed above are the ones which are relevant for our evaluations. Since one simulation for one seed takes some time to complete, the simulations were split up onto several different machines to speed up the overall simulation process. This was achieved by distributing several blocks of simulations to some machines by creating a small shell script that looks like this:

```
#!/bin/bash
WORKINGDIR="/home/pg439/share/simulation/"
ssh -f fluor  "cd ${WORKINGDIR};nice +18 ./Sim1" &
ssh -f mangan  "cd ${WORKINGDIR};nice +18 ./Sim2" &
ssh -f chlor  "cd ${WORKINGDIR};nice +18 ./Sim3" &
```

Transport Protocol: TCP
BeeHive: x=0; DSR: x=300; AODV: x=600; DSDV: x=900

	mobility			packets		topology		
Run-Nr.	minSpeed	maxSpeed	pauseTime	1/sec	size	x-size	y-size	#nodes
1+x	0	5	60	10	512	2400	480	50
2+x	0	10	60	10	512	2400	480	50
3+x	0	15	60	10	512	2400	480	50
4+x	0	20	60	10	512	2400	480	50
6+x	0	20	60	30	512	2400	480	50
8+x	0	20	60	60	512	2400	480	50
10+x	0	20	60	100	512	2400	480	50
16+x	0	20	60	100	512	2400	480	50
20+x	0	20	30	100	512	2400	480	50
24+x	0	20	1	100	512	2400	480	50
26+x	0	20	60	100	512	1073	1073	50
28+x	0	20	60	100	512	3400	340	50
102+x	0	10	60	10	2048	2400	480	50
202+x	0	10	60	10	4096	2400	480	50

Table 5.1.: TCP-Runs for BeeHive, DSR, AODV and DSDV

Transport Protocol: UDP
BeeHive: x=0; DSR: x=300; AODV: x=600

	mobility			packets		topology		
Run-Nr.	minSpeed	maxSpeed	pauseTime	1/sec	size	x-size	y-size	#nodes
51+x	0	5	60	10	512	2400	480	30
52+x	0	10	60	10	512	2400	480	30
53+x	0	15	60	10	512	2400	480	30
54+x	0	20	60	10	512	2400	480	30

Table 5.2.: UDP-Runs for BeeHive, DSR and AODV

```
ssh -f barium  "cd ${WORKINGDIR};nice +18 ./Sim4" &
ssh -f gold  "cd ${WORKINGDIR};nice +18 ./Sim5" &
ssh -f calcium  "cd ${WORKINGDIR};nice +18 ./Sim6" &
ssh -f blei  "cd ${WORKINGDIR};nice +18 ./Sim7" &
ssh -f eisen  "cd ${WORKINGDIR};nice +18 ./Sim8" &
```

In the following you can see an example for a shell script like Sim1:

```
#!/bin/bash

simtime=1000
nodes=50
tnodes=50

#RUN 1
for i in 86430 68431 53142 61313 14874;
do
./ns results.tcl -rp Beehive -seed $i -rwpmin 0 -rwpmax 5 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0001-$nodes-$simtime-$i -ifqlen 100
./ns results.tcl -rp DSR -seed $i -rwpmin 0 -rwpmax 5 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0301-$nodes-$simtime-$i -ifqlen 100
./ns results.tcl -rp AODV -seed $i -rwpmin 0 -rwpmax 5 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0601-$nodes-$simtime-$i -ifqlen 100
./ns results.tcl -rp DSDV -seed $i -rwpmin 0 -rwpmax 5 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0901-$nodes-$simtime-$i -ifqlen 100
done

#RUN 2
for i in 86430 68431 53142 61313 14874;
do
./ns results.tcl -rp Beehive -seed $i -rwpmin 0 -rwpmax 10 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0002-$nodes-$simtime-$i -ifqlen 100
./ns results.tcl -rp DSR -seed $i -rwpmin 0 -rwpmax 10 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0302-$nodes-$simtime-$i -ifqlen 100
./ns results.tcl -rp AODV -seed $i -rwpmin 0 -rwpmax 10 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0602-$nodes-$simtime-$i -ifqlen 100
./ns results.tcl -rp DSDV -seed $i -rwpmin 0 -rwpmax 10 -rwppause 60 -pktsize 512 -pktrate 0.041 \
  -nn $nodes -tn $tnodes -$simtime simtime -filename r0902-$nodes-$simtime-$i -ifqlen 100
done
```

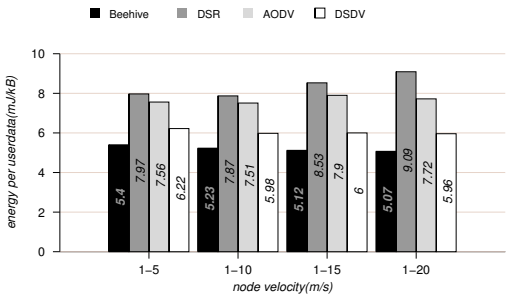


Figure 5.1.: Energy results depending on node velocity (from runs 1+x, 2+x, 3+x, 4+x)

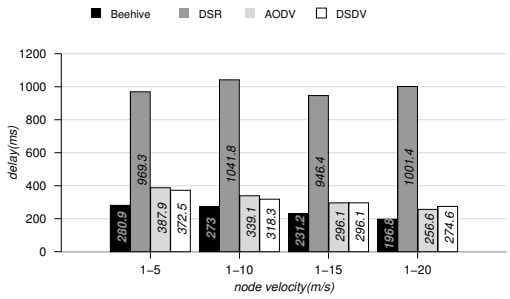


Figure 5.2.: Delay results depending on node velocity (from runs 1+x, 2+x, 3+x, 4+x)

5.3. Testing Mobility Be(e)haviour

5.3.1. Node Velocity

At first, the influence of different node movement behaviours was tested using four cases with maximum speeds v_{max} of 5, 10, 15 and 20 m/s and a minimum speed of $v_{min} = 0$ m/s.

Figure 5.1 describes the influence of the movement speed towards the energy that has been used to deliver one kB of user data to the desired destination in average. This includes as well the proportional energy of the involved control packets as the pure energy consumption to send and receive this data.

As one can see BeeHive does consume less energy (per delivered user data) than any other algorithm tested. This may be a result of the simplicity of the BeeHive algorithm in comparison to the others. AODV and DSR do consume significantly more energy than BeeHive. This is a result of the AODV and DSR specific behaviour, like route error messages and packet salvaging (DSR) which both does not exist in any comparable way in BeeHive. BeeHive abandons these mechanisms and so accepts to lose some more packets but as one can see in figure 5.4, BeeHive is also able to deliver a higher amount of user packets to the destinations. AODV also has a higher energy consumption which partially is a result of the route error mechanisms similar to DSR. Since AODV has no extensive broadcast route detection mechanism like DSR (and BeeHive) the energy consumption is slightly less than DSR. Both algorithms (DSR, AODV) have route caching mechanisms, which are theoretically promisingly mechanisms, but practically they seem to deteriorate the results. In the developing time of BeeHive there were also

made some experiments of using route caches but they all made the results rather worse than better. Keeping old routes too long results in more route faults which can explain the rising energy consumption. DSR has a disadvantage in comparison to AODV because AODV maintains route table with the use of "hello"-messages which are non-existent in DSR. Furthermore, DSR uses packet salvaging which - in combination with very old cached routes - is bad for energy behaviour. If a node realises that a route, which should be used for sending a data packet, is down, it searches its cache and maybe chooses a route that that itself is also very old and already down which may result in another route error and packet salvaging (maybe over an old, non-existing route again, and again, and so on). Like BeeHive, DSDV is a quite simple algorithm that is less complicated and hence less power-consuming.

Figure 5.2 describes the influence of the movement speed towards the average delay it takes to deliver a user data packet from its source to its destination. This includes as well the time it takes to discover a new route (only necessary in some protocols (DSR, AODV (if no route is cached), BeeHive (if no bee is available) as the pure travelling (transmitting/receiving) time of a packet.
Simulations show that all algorithms have decreasing delays with increasing mobility (node velocity) whereas BeeHive is significantly better than AODV and DSDV. DSR has a disproportionate long, but nearly constant delay. These cognitions have to be taken as they are, since there is no real explanation for this, yet.

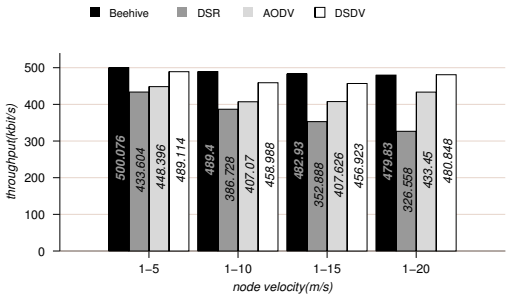


Figure 5.3.: Throughput results depending on the node velocity (from runs 1+x, 2+x, 3+x, 4+x)

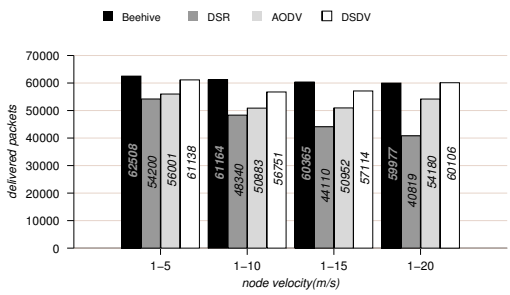


Figure 5.4.: Total packets successfully delivered to destination (from runs 1+x, 2+x, 3+x, 4+x)

Figures 5.3 and 5.4 describe the throughput achieved and the total amount of successfully delivered packets. Both diagrams are connected closely together so they look similar. As one can see BeeHive, AODV and DSDV do not care much about the velocity of the nodes in the network. The throughput and amount of delivered packets

nearly remain the same. Only the throughput of DSR decreases with increasing node velocity. This may rather be a result of DSR-specific mechanisms like caching and/or packet salvaging. BeeHive deals with this problem in a good manner: simplicity and not using these mechanisms. DSR also has problems to keep its delivery ratio on a certain level with increasing speed so one can conclude that caching routes too long is a problem in "fast" networks. Since BeeHive does not have a similar mechanism BeeHive doesn't suffer from this effect. Also the two other algorithms have ways bypassing this problem by using "hello"-messages (AODV) or broadcasting routing tables (DSDV). These mechanisms refresh the known routes in certain time intervals. DSR has to wait for route errors (after retrying to retransmit the packet several times) and salvage the packet costly. This is also an explanation for the long delay.

5.3.2. Pause Time

Now, the pause time is varied (figures 5.5, 5.6 and 5.7). Each node moves randomly to a specific destination, waits there for a time which is represented by *pause time*. Then it moves to another point, waits again and so on.
Generally the results are quite comprehensible. All algorithms have decreasing values of delay. If a node rests for a longer time all algorithms have got more time to adapt to this new situation. Routes exist a longer period of time, which decreases the time that packets have to wait while new routes have to be discovered, so, the delay of all algorithms decrease with increasing pause time. This also provides the possibility to send more packets over known, working routes, that results in a higher throughput. BeeHive's energy consumption benefits of this situation as well, but all other algorithms have increasing energy consumption with decreasing network mobility. Until now we don't have an explanation for this behaviour. We analyzed the simulation data and found the reason for the increasing energy consumption. In networks with longer pause times (60 seconds) BeeHive (as comparison) needs less control packets to deliver more user data to the destinations (which is perfectly clear). On the other hand, for example, DSR uses six times more control packets to deliver only 8% more data packets to its destination and DSDV needs twice as much control packets to deliver "only" 69% more data packets.

pause time	BeeHive		DSR		AODV		DSDV	
	control	delivered	control	delivered	control	delivered	control	delivered
1 second	65575	148544	60361	141451	962799	139925	116516	157080
60 seconds	61929	220850	396038	152651	701780	199108	235517	226285

Table 5.3.: table: packet delivery depending on pause time

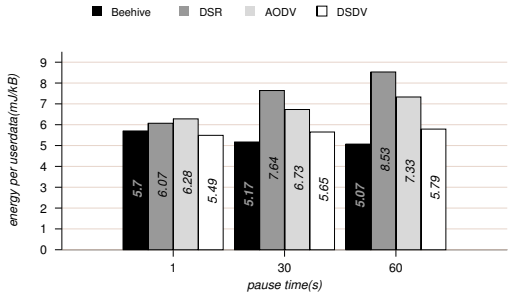


Figure 5.5.: Energy results depending on pause time (from runs 24+x, 20+x, 16+x)

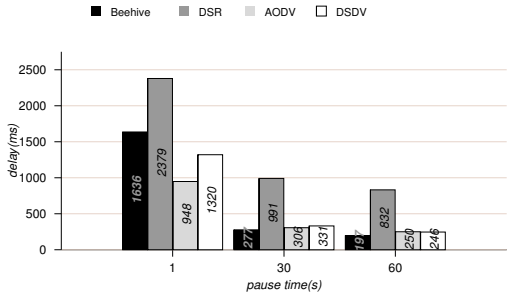


Figure 5.6.: Delay results depending on pause time (from runs 24+x, 20+x, 16+x)

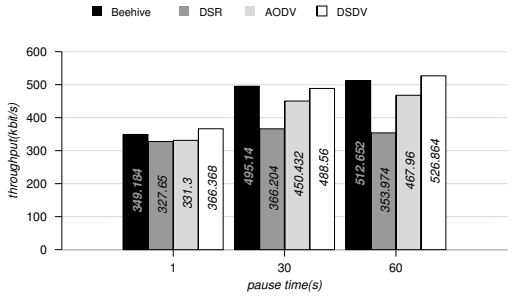


Figure 5.7.: Throughput results depending on pause time (from runs 24+x, 20+x, 16+x)

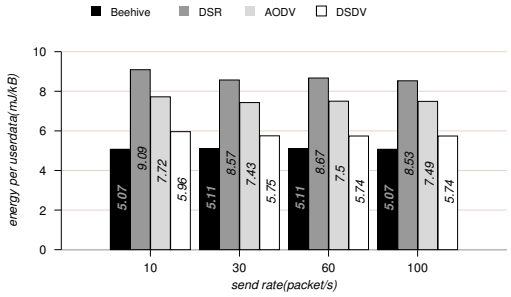


Figure 5.8.: Energy results depending on send rate (from runs 4+x, 6+x, 8+x, 10+x)

5.4. Testing Send Rate Be(e)haviour

Sending more packets per time unit will result in a higher amount of sent packets and so a higher amount of packets delivered successfully, which results in a higher throughput. The delay of all four algorithms is slightly decreasing (but nearly constant, anyway) only the high delay of DSR is decreasing from 1000 to 800 ms. This is explainable by the fact that the beginning route discovery time (that is wasted) can be portioned to more overall packets. The energy consumption decreases with an increasing amount of packets sent because with the same amount of control packets more user data can be sent. This effect pertains all four algorithms.

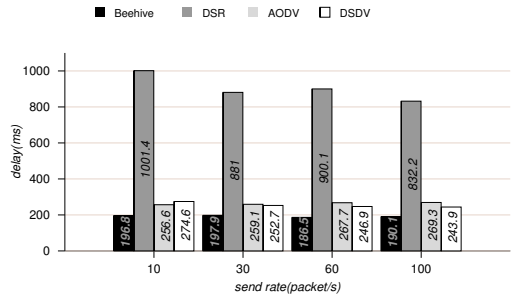


Figure 5.9.: Delay results depending on send rate (from runs 4+x, 6+x, 8+x, 10+x)

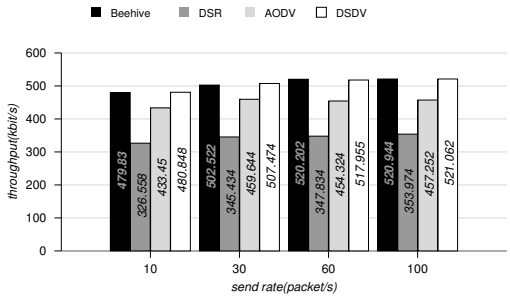


Figure 5.10.: Throughput results depending on send rate (from runs 4+x, 6+x, 8+x, 10+x)

5.5. Testing Packet Size Be(e)haviour

5.6. Testing Packet Size Be(e)haviour

In figures 5.11, 5.12 and 5.13 the influence of varying packet sizes is shown. When increasing the packet size the energy needed to deliver one byte of user data is decreasing (figure 5.11). There are two explanations for this. First, the send energy is calculated by a constant factor per packet plus a factor multiplied with the packet size. With increasing packet size the constant factor is so portioned to more bytes. The second reason is that the same amount of energy used for control packets can be portioned to a bigger amount of delivered user bytes. For example, if you use 20 control packets to deliver 10 user packets it is sure, that when the user packet size is bigger in size you can deliver more data with the same amount of control packet energy used. As shown all

algorithms benefit from this effect in the same way, whereas BeeHive does consume the least energy at all, closely followed by DSDV. The more complex algorithms here show also their disadvantages.

In figure 5.12 one can see that the delay increases with increasing packet size, which can be explained by the longer transmission time for each packet. One could think actually it shouldn't make this big difference but the results seem to be unequivocal. Due to Wireless LAN characteristics in an accumulation of nodes a transmission of a packet block other nodes for short time interval. So, the bigger the packet size, the longer the blocking interval inside this accumulation will be. And since every single node does send data packets in the simulations performed, this effect dominates with an increase in packet size.

For sure one can see, that BeeHive still has the shortest delay, closely followed by DSDV and AODV and DSR still has a disproportionate long delay.

The throughput (figure 5.13) surely increases when increasing packet size, because the same amount of packets should be sent and sending the same amount of bigger packets will increase the throughput. "Should be sent" is here used because the simulations did not really send the same amount of packets (although they should) which may be a result of the increased delay: the TCP agent just sends a new packet on receive of the acknowledge of the previous packet. So with a longer delay the amount of packets sent by TCP will decrease.

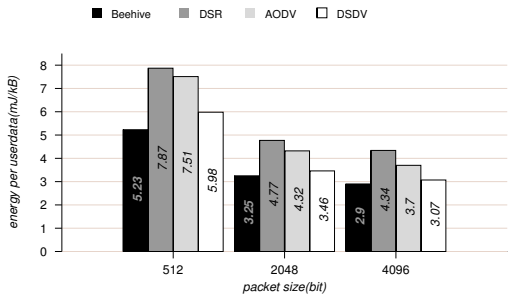


Figure 5.11.: Energy results depending on packet size (from runs 2+x, 102+x, 202+x)

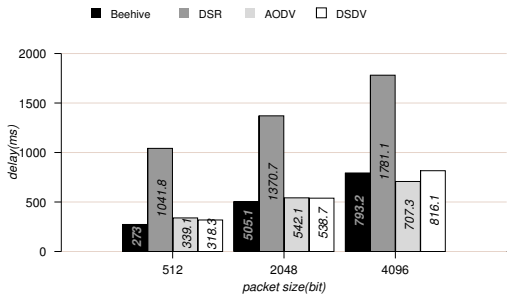


Figure 5.12.: Delay results depending on packet size (from runs 2+x, 102+x, 202+x)

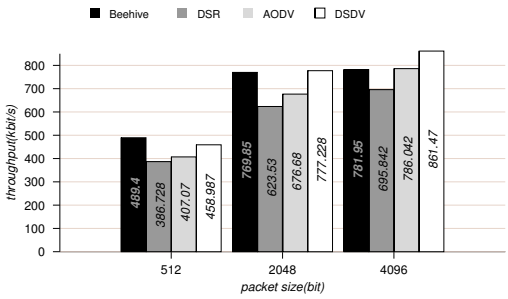


Figure 5.13.: Throughput results depending on packet size (from runs 2+x, 102+x, 202+x)

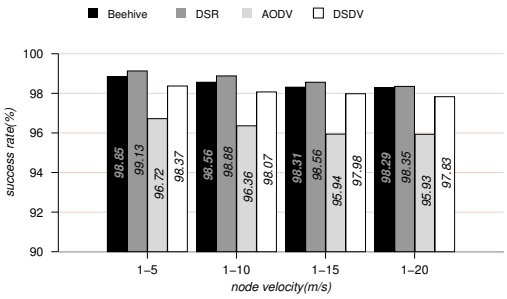


Figure 5.14.: Success results depending on node velocity (from runs 1+x, 2+x, 3+x, 4+x)

5.7. Success Rates

The success rate as shown in figures 5.14, 5.15 and 5.16 describes the percentage of data packets which reach the destination successfully. In this contest the DSR algorithm has slightly the best success rates. BeeHive places second (with a nearly negligible leeway of 0.5%-point max), followed by DSDV with another leeway of 0.4%. AODV seems to have big problems in mobile networks; the success rate gets worse with increasing node mobility. The send rate does not influence the success rate of any algorithm in a mentionable manner.

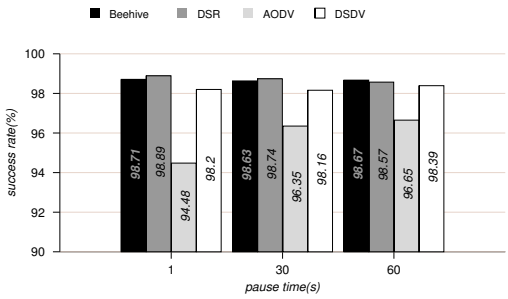


Figure 5.15.: Success results depending on pause time (from runs 24+x, 20+x, 16+x)

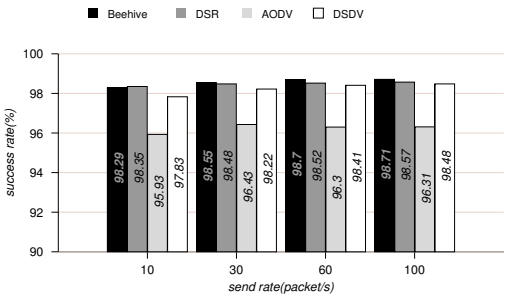


Figure 5.16.: Success results depending on send rate (from runs 4+x, 6+x, 8+x, 10+x)

5.8. Testing Area Size Be(e)haviour

This section is not really important for rating an algorithm but it is still nice to know how algorithms behave in different topologies. For this, three different topologies with each the same amount of footprint (in m^2) where created:

- 2400 x 480 meters
- 1073 x 1073 meters
- 3400 x 340 meters

As one can see in figure 5.17, the topology does not have a big effect on the energy used for delivering user data.

Figure 5.18 shows that the delay increases in squared topologies which may be a result of send and receive collisions of the wireless node (since only one node can send at one time, and all others in range only can listen (receive) or have to wait). If the topology is now a rectangle with clearly unequal side lengths the delay and throughput will be better because of more widely-spread nodes not blocking each others transmission by forming some kind of line.

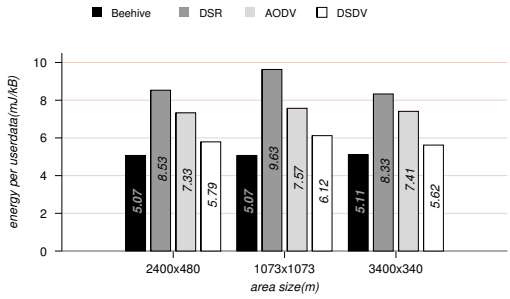


Figure 5.17.: Energy results depending on different topologies (from runs 16+x, 26+x, 28+x)

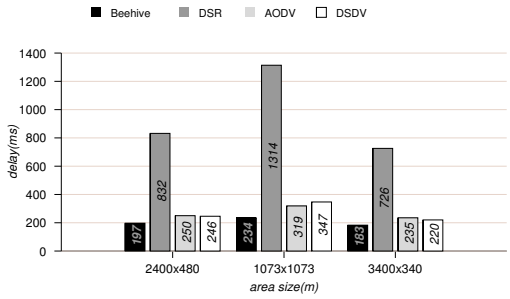


Figure 5.18.: Delay results depending on different topologies (from runs 16+x, 26+x, 28+x)

5.9. UDP

All the simulations above were performed with TCP as the responsible transport protocol. This section will evaluate the results performed with UDP.

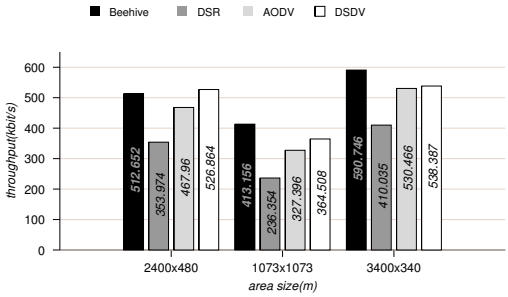


Figure 5.19.: Throughput results depending on different topologies (from runs 16+x, 26+x, 28+x)

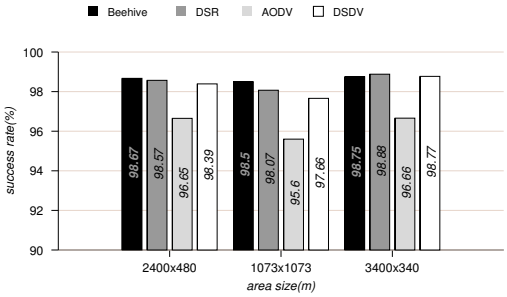


Figure 5.20.: Success results depending on different topologies (from runs 16+x, 26+x, 28+x)

At first the number of nodes in the simulation was reduced to 30 nodes, to reduce the complexity of the simulation. First simulations with 50 nodes often aborted after less than 200 seconds (simulated time). The NS2 implementation of DSDV seems to be not compatible with UDP, because no simulation run was finished. Hence we have only compared BeeHive with DSR and AODV. For UDP, we also activated the scout caching, since without it the results where quite bad. If further UDP developing is desired, the reason has to be figured out.

The main difference between the UDP and the TCP agent is that UDP does not care about the successful delivery of its packets. It keeps sending packets all the time with the justified send rate. TCP instead waits for the acknowledge that the previous packet did reach its destination successfully before sending another packet. Because of these circumstances it was predictable that the success rate will crash, what actually occurred (see figure 5.21). This also lead to higher energy values (figure 5.22), because a lot of "wasted" energy of lost packets had to be portioned to the successful delivered packets. Overall, the figures (5.22 - 5.24) show that BeeHive still is the most auspicious algorithm. BeeHive is able to deliver about 24-25% of packets, whereas DSR only is able to deliver 13-15% and AODV 23-25% (figure 5.21). This, for sure, reflects in the throughput (more delivered packets = more throughput) (figure 5.24) and energy (more delivered packets = more packets to portion the "wasted energy" to) (figure 5.22).

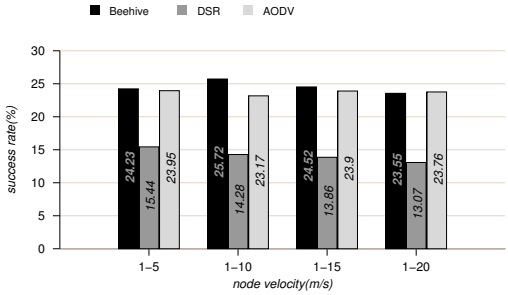


Figure 5.21.: UDP: Success results depending on velocity (from runs 51+x, 52+x, 53+x, 54+x)

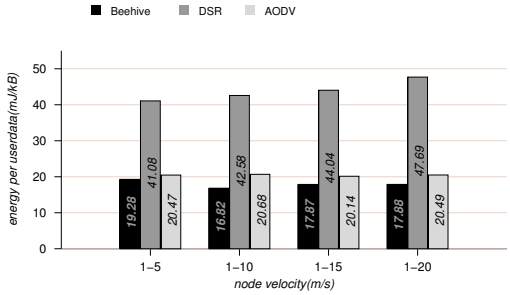


Figure 5.22.: UDP: Energy results on node velocity (from runs 51+x, 52+x, 53+x, 54+x)

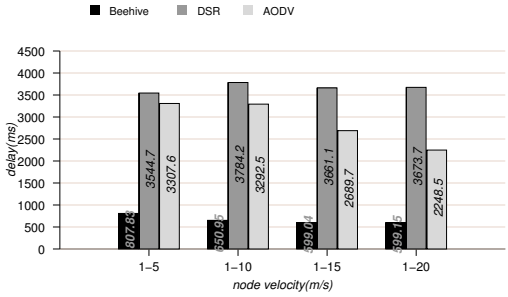


Figure 5.23.: UDP: Delay results depending on node velocity (from runs 51+x, 52+x, 53+x, 54+x)

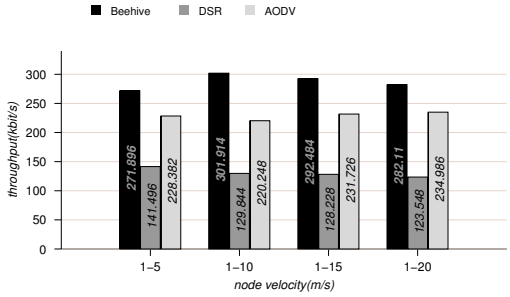


Figure 5.24.: UDP: Throughput results depending on velocity (from runs 51+x, 52+x, 53+x, 54+x)

5.10. Examination of Delay

In the previous sections one can see that the delay results are conspicuously high. As further examination we have tried to find out whether outliers are responsible for these values. Therefore we created a table (5.4) that shows how long it took until a certain percentage of packets reached their destinations.

	Delay TCP				Delay UDP		
	BeeHive	DSR	AODV	DSDV	BeeHive	DSR	AODV
80% delay	105.64	167.73	156.85	117.89	61.92	1257.67	1006.9
90% delay	153.84	278.58	220.52	176.01	180.2	1897.6	1761.29
95% delay	191.36	396.29	269.31	223.76	348.3	2398.54	2310.29
100% delay	280.97	969.39	387.96	372.55	807.83	3544.73	3307.6

Table 5.4.: comparison of TCP and UDP delay (from runs 1+x)

5.11. Appendix/Conclusion

In comparison with DSR, AODV and DSDV BeeHive shows quite good results in all "disciplines". The results indicate that the target to develop an energy-aware routing algorithm has been achieved. In all cases, the energy consumed by BeeHive to deliver a certain amount of user data is less than the one's consumed by the other algorithms. Furthermore the success rate is not impaired. Through renouncement of complicated mechanisms the delay and throughput could also be in improved. During the developmental period of BeeHive we also made some attempts to improve these parameters. Most of these improvement has lead to converse results, so they were cancelled in early stages of development or even called off. This let's conclude that for wireless ad-hoc networks simple algorithms are the best solution to achieve good energy, delay and throughput ratios. This is also underpinned by the results of DSDV, which is also quite simple and so places second in the competition of the four algorithms (although the results could not be verified with UDP).

Last, but not least, a little annotation concerning the send rate in ns2. In NS2, one can find the parameter "-pktrate" to set the rate the packets should be send with. Theoretically the value to set is the pause time between the transmissions of two packets. If it is set to 0.1, for example, the packet rate should be 10 packets per second. While evaluating it was discovered that this assumption is wrong, although it should be correct. We found out, that the packet size also affects the send rate. So, if the "-pktsize" parameter is modified, the send rate also changes although the "-pktrate" is not touched.

Since the effects of the modifications were predictable, we had to find out the correct values by the "trial-and-error" method. For this, we build a small 2 node network, one node sending and counted the packets sent/received (whereas packets sent = packets received) and modified packet rate and size until we got the values we needed. The following table shows the results (columns: packet size; rows: desired packet rate, the cells represent the values that have to be set as "-pktrate"):

	512 bit	2048 bit	4096 bit
10	0.041	0.01025	0.005125
30	0.0137	N/A	N/A
60	0.00684	N/A	N/A
100	0.00411	N/A	N/A

N/A: not ascertained

5.11.1. Detailed Result for each Simulation

Tables 5.5 up to 5.72 (5.5 - 5.60:TCP; 5.61 - 5.72:UDP) provide the detailed average results received from each result. For this, another parser was written (in Java) that parsed the output provided by the first (ns2-output-)parser, merged the ones with same run numbers together (that is, merging together the different seeds from a run) and automatically generated a TeX file for each run containing a table providing the detailed results.

TCP

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.4 μJ/B
Throughput: 500.076 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1073.896	496.0	15782.4	821.39
	rcvDataEnergy	494.266	346.8	5860.32	282.639
Control:	sendCtrlEnergy	372.91	310.8	556.8	79.978
	rcvCtrlEnergy	85.983	72.0	360.24	53.763
	→ *sendCtrlP2PEnergy*	502.975	488.4	556.8	20.965
	→ *rcvCtrlP2PEnergy*	348.251	345.12	360.24	4.656
	→ *sendCtrlBCEnergy*	325.861	310.8	379.2	16.277
	→ *rcvCtrlBCEnergy*	74.981	72.0	90.0	4.067
Miscellaneous:	batteryLevel (%):	6.008	0.0	68.417	10.143
	delay 80%:	528.22	2.0	633.59	554.41
	delay 90%:	769.18	2.0	1020.02	871.63
	delay 95%:	956.82	2.0	1503.47	1170.96
	delay 98%:	1130.56	2.0	2305.31	1521.49
	delay 99%:	1217.98	2.0	3155.79	1747.03
	delay 100%:	280.97	2.0	44564.22	579.36

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	63237	217908	62508	215183	98.85%	98.75%	728	2725	N/A	N/A
	Data	69027	217378	68461	215183	99.18%	98.99%	564	2195	N/A	529
Route	Net Data	75124	235592	74559	233397	99.25%	99.07%	564	2195	0	0
	Net Control	567	14225	3097	83095	546.21%	584.15%	N/A	N/A	N/A	N/A
	→ *peer2peer*	160	3769	139	3289	86.88%	87.26%	20	480	N/A	N/A
	→ *broadcast*	407	10455	2957	79806	726.54%	763.33%	N/A	N/A	N/A	N/A
MAC	Net Sum	75693	249817	77657	316492	102.59%	126.69%	N/A	-66675	2329	18255

Table 5.5.: Average Result Table: Run 00001

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.23 μJ/B
Throughput: 489.4 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1037.724	496.0	15782.4	672.093
	rcvDataEnergy	477.868	346.8	5860.32	211.233
Control:	sendCtrlEnergy	371.067	310.8	556.8	79.146
	rcvCtrlEnergy	84.935	72.0	360.24	50.297
	→ *sendCtrlP2PEnergy*	506.885	488.4	556.8	21.047
	→ *rcvCtrlP2PEnergy*	349.066	345.12	360.24	4.695
	→ *sendCtrlBCEnergy*	326.896	310.8	379.2	16.115
	→ *rcvCtrlBCEnergy*	75.391	72.0	90.0	4.105
Miscellaneous:	batteryLevel (%):	4.048	0.0	47.639	7.634
	delay 80%:	538.3	2.0	394.45	474.26
	delay 90%:	740.21	2.0	760.13	739.12
	delay 95%:	907.12	2.0	1223.76	1016.44
	delay 98%:	1065.17	2.0	1962.45	1347.3
	delay 99%:	1146.73	2.0	2677.13	1567.55
	delay 100%:	273.02	2.0	97853.73	678.41

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	62057	213554	61164	210666	98.56%	98.65%	892	2888	N/A	N/A
	Data	67576	212813	66964	210666	99.09%	98.99%	611	2147	N/A	740
Route	Net Data	71044	223265	70432	221117	99.14%	99.04%	611	2147	0	0
	Net Control	552	13560	3085	80895	558.88%	596.57%	N/A	N/A	N/A	N/A
	→ *peer2peer*	148	3323	123	2812	83.11%	84.62%	23	511	N/A	N/A
	→ *broadcast*	404	10236	2961	78082	732.92%	762.82%	N/A	N/A	N/A	N/A
MAC	Net Sum	71597	236825	73518	302012	102.68%	127.53%	N/A	-65187	2448	19664

Table 5.6.: Average Result Table: Run 00002

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.12 μJ/B
Throughput: 482.93 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1017.9	496.0	15782.4	586.266
	rcvDataEnergy	469.006	346.8	5860.32	167.423
Control:	sendCtrlEnergy	371.753	310.8	556.8	79.707
	rcvCtrlEnergy	85.253	72.0	360.24	50.529
	→ *sendCtrlP2PEnergy*	508.896	488.4	556.8	21.979
	→ *rcvCtrlP2PEnergy*	349.621	345.12	360.24	4.931
	→ *sendCtrlBCEnergy*	327.639	310.8	379.2	16.743
	→ *rcvCtrlBCEnergy*	75.644	72.0	90.0	4.323
Miscellaneous:	batteryLevel (%):	6.151	0.0	40.218	8.099
	delay 80%:	518.06	2.0	327.67	422.5
	delay 90%:	674.58	2.0	553.13	604.55
	delay 95%:	803.03	2.0	823.08	806.61
	delay 98%:	921.0	2.0	1205.56	1040.68
	delay 99%:	981.86	2.0	1785.51	1200.55
	delay 100%:	231.29	2.0	51607.02	571.32

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	61405	211078	60365	207831	98.31%	98.46%	1039	3247	N/A	N/A
	Data	66766	210210	66072	207831	98.96%	98.87%	693	2379	N/A	868
Route	Net Data	68830	216586	68136	214207	98.99%	98.9%	693	2379	0	0
	Net Control	481	11670	2595	67240	539.5%	576.18%	N/A	N/A	N/A	N/A
	→ *peer2peer*	129	2844	107	2359	82.95%	82.95%	21	485	N/A	N/A
	→ *broadcast*	351	8825	2488	64881	708.83%	735.2%	N/A	N/A	N/A	N/A
MAC	Net Sum	69312	228256	70732	281448	102.05%	123.3%	N/A	-53191	2151	15643

Table 5.7.: Average Result Table: Run 00003

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.07 μJ/B
Throughput: 479.83 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1006.314	496.0	15782.4	553.743
	rcvDataEnergy	463.661	346.8	5860.32	149.155
Control:	sendCtrlEnergy	369.503	310.8	556.8	78.732
	rcvCtrlEnergy	84.825	72.0	360.24	48.373
	→ *sendCtrlP2PEnergy*	513.625	488.4	556.8	21.503
	→ *rcvCtrlP2PEnergy*	350.71	345.12	360.24	4.809
	→ *sendCtrlBCEnergy*	328.537	310.8	379.2	16.515
	→ *rcvCtrlBCEnergy*	76.076	72.0	90.0	4.346
Miscellaneous:	batteryLevel (%):	11.154	0.0	50.043	9.095
	delay 80%:	484.09	2.0	308.77	375.79
	delay 90%:	611.33	2.0	495.79	508.67
	delay 95%:	708.74	2.0	679.44	648.18
	delay 98%:	797.7	2.0	939.08	814.77
	delay 99%:	842.21	2.0	1353.11	924.79
	delay 100%:	196.87	2.0	37984.33	519.23

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	61021	209749	59977	206713	98.29%	98.55%	1043	3036	N/A	N/A
	Data	66295	208799	65644	206713	99.02%	99.0%	651	2086	N/A	950
Route	Net Data	67511	212601	66859	210515	99.03%	99.02%	651	2086	0	0
	Net Control	467	11131	2493	63119	533.83%	567.06%	N/A	N/A	N/A	N/A
	→ *peer2peer*	118	2461	96	2007	81.36%	81.55%	21	453	N/A	N/A
	→ *broadcast*	349	8670	2397	61111	686.82%	704.86%	N/A	N/A	N/A	N/A
MAC	Net Sum	67979	223732	69353	273634	102.02%	122.3%	N/A	-49901	2054	15463

Table 5.8.: Average Result Table: Run 00004

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.11 μJ/B
Throughput: 502.522 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1016.668	496.0	12495.2	563.402
	rcvDataEnergy	468.764	346.8	4659.36	153.2
Control:	sendCtrlEnergy	369.541	310.8	556.8	77.538
	rcvCtrlEnergy	85.092	72.0	360.24	47.681
	→ *sendCtrlP2PEnergy*	513.83	488.4	556.8	19.863
	→ *rcvCtrlP2PEnergy*	350.753	345.12	360.24	4.453
	→ *sendCtrlBCEnergy*	329.721	310.8	379.2	16.172
	→ *rcvCtrlBCEnergy*	76.571	72.0	90.0	4.268
Miscellaneous:	batteryLevel (%):	7.679	0.0	41.663	8.653
	delay 80%:	507.35	2.0	297.76	375.37
	delay 90%:	630.49	2.0	453.34	499.69
	delay 95%:	723.57	2.0	639.21	630.15
	delay 98%:	809.26	2.0	991.28	789.54
	delay 99%:	854.4	2.0	1654.01	906.45
	delay 100%:	197.93	2.0	65406.81	493.23

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	63735	219357	62814	216397	98.55%	98.65%	919	2959	N/A	N/A
	Data	69458	218710	68756	216397	98.99%	98.94%	702	2312	N/A	646
Route	Net Data	71493	225072	70790	222759	99.02%	98.97%	702	2312	0	0
	Net Control	472	11113	2470	61103	523.31%	549.83%	N/A	N/A	N/A	N/A
	→ *peer2peer*	115	2397	90	1882	78.26%	78.51%	24	514	N/A	N/A
	→ *broadcast*	356	8716	2379	59220	668.26%	679.44%	N/A	N/A	N/A	N/A
MAC	Net Sum	71966	236185	73261	283862	101.8%	120.19%	N/A	-47676	2162	16276

Table 5.9.: Average Result Table: Run 00006

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.11 μJ/B
Throughput: 520.202 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1016.279	496.0	14131.2	573.451
	rcvDataEnergy	468.667	346.8	5258.16	159.917
Control:	sendCtrlEnergy	371.564	310.8	556.8	78.954
	rcvCtrlEnergy	85.51	72.0	360.24	48.041
	→ *sendCtrlP2PEnergy*	516.931	488.4	556.8	19.625
	→ *rcvCtrlP2PEnergy*	351.502	345.12	360.24	4.394
	→ *sendCtrlBCEnergy*	330.614	310.8	379.2	16.26
	→ *rcvCtrlBCEnergy*	76.883	72.0	90.0	4.315
Miscellaneous:	batteryLevel (%):	7.145	0.0	41.123	8.035
	delay 80%:	509.41	1.74	297.76	359.04
	delay 90%:	622.28	1.74	453.34	467.64
	delay 95%:	700.78	1.74	636.61	567.84
	delay 98%:	776.46	1.74	931.92	706.55
	delay 99%:	815.58	1.74	1267.6	804.67
	delay 100%:	186.56	1.74	45855.8	448.37

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	65878	226882	65024	224035	98.7%	98.75%	853	2847	N/A	N/A
	Data	71828	226267	71175	224035	99.09%	99.01%	652	2232	N/A	615
Route	Net Data	73984	232953	73331	230720	99.12%	99.04%	652	2232	0	0
	Net Control	517	11984	2694	65668	521.08%	547.96%	N/A	N/A	N/A	N/A
	→ *peer2peer*	130	2634	102	2055	78.46%	78.02%	27	579	N/A	N/A
	→ *broadcast*	386	9349	2592	63613	671.5%	680.43%	N/A	N/A	N/A	N/A
MAC	Net Sum	74503	244937	76026	296389	102.04%	121.01%	N/A	-51452	2171	17039

Table 5.10.: Average Result Table: Run 00008

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.07 μJ/B
Throughput: 520.944 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1008.743	496.0	12495.2	546.846
	rcvDataEnergy	465.291	346.8	4659.36	143.977
Control:	sendCtrlEnergy	369.817	310.8	556.8	78.006
	rcvCtrlEnergy	85.021	72.0	360.24	46.676
	→ *sendCtrlP2PEnergy*	518.282	488.4	556.8	20.595
	→ *rcvCtrlP2PEnergy*	351.803	345.12	360.24	4.612
	→ *sendCtrlBCEnergy*	330.801	310.8	379.2	16.338
	→ *rcvCtrlBCEnergy*	76.919	72.0	90.0	4.34
Miscellaneous:	batteryLevel (%):	7.731	0.0	42.066	8.409
	delay 80%:	519.37	2.0	297.76	373.21
	delay 90%:	636.47	2.0	453.34	485.86
	delay 95%:	718.02	2.0	636.61	589.76
	delay 98%:	793.11	2.0	931.92	721.6
	delay 99%:	831.5	2.0	1218.74	813.44
	delay 100%:	190.16	2.0	38957.63	460.2

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	65965	227195	65117	224449	98.71%	98.79%	847	2746	N/A	N/A
	Data	71919	226579	71273	224449	99.1%	99.06%	646	2130	N/A	616
Route	Net Data	73512	231569	72865	229439	99.12%	99.08%	646	2130	0	0
	Net Control	499	11497	2604	63119	521.84%	549.0%	N/A	N/A	N/A	N/A
	→ *peer2peer*	120	2393	93	1854	77.5%	77.48%	26	539	N/A	N/A
	→ *broadcast*	378	9104	2510	61265	664.02%	672.95%	N/A	N/A	N/A	N/A
MAC	Net Sum	74012	243067	75469	292558	101.97%	120.36%	N/A	-49491	2117	16368

Table 5.11.: Average Result Table: Run 00010

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.07 μJ/B
Throughput: 512.652 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1008.981	496.0	15782.4	545.516
	rcvDataEnergy	465.421	346.8	5860.32	143.893
Control:	sendCtrlEnergy	370.913	310.8	556.8	78.739
	rcvCtrlEnergy	85.238	72.0	360.24	47.526
	→ *sendCtrlP2PEnergy*	517.234	488.4	556.8	19.909
	→ *rcvCtrlP2PEnergy*	351.573	345.12	360.24	4.467
	→ *sendCtrlBCEnergy*	330.471	310.8	379.2	16.23
	→ *rcvCtrlBCEnergy*	76.811	72.0	90.0	4.295
Miscellaneous:	batteryLevel (%):	8.242	0.0	44.54	8.588
	delay 80%:	520.07	2.0	289.73	375.46
	delay 90%:	636.96	2.0	405.3	487.64
	delay 95%:	727.12	2.0	614.88	614.22
	delay 98%:	811.29	2.0	955.79	771.63
	delay 99%:	854.86	2.0	1351.34	882.21
	delay 100%:	197.54	2.0	42629.58	471.07

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	64948	223641	64081	220850	98.67%	98.75%	867	2790	N/A	N/A
	Data	70804	223020	70139	220850	99.06%	99.03%	664	2169	N/A	621
Route	Net Data	72440	228152	71774	225983	99.08%	99.05%	664	2169	0	0
	Net Control	488	11306	2537	61929	519.88%	547.75%	N/A	N/A	N/A	N/A
	→ *peer2peer*	121	2434	94	1889	77.69%	77.61%	26	545	N/A	N/A
	→ *broadcast*	367	8872	2443	60040	665.67%	676.74%	N/A	N/A	N/A	N/A
MAC	Net Sum	72929	239459	74313	287912	101.9%	120.23%	N/A	-48453	2133	16252

Table 5.12.: Average Result Table: Run 00016

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.17 μJ/B
Throughput: 495.14 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1027.199	496.0	10874.4	578.676
	rcvDataEnergy	473.742	346.8	4063.92	160.659
Control:	sendCtrlEnergy	373.6	310.8	556.8	80.344
	rcvCtrlEnergy	85.18	72.0	360.24	48.237
	→ *sendCtrlP2PEnergy*	514.886	488.4	556.8	20.369
	→ *rcvCtrlP2PEnergy*	351.04	345.12	360.24	4.596
	→ *sendCtrlBCEnergy*	329.981	310.8	379.2	16.087
	→ *rcvCtrlBCEnergy*	76.479	72.0	90.0	4.249
Miscellaneous:	batteryLevel (%):	2.574	0.0	27.417	4.419
	delay 80%:	625.68	2.0	438.04	479.2
	delay 90%:	806.17	2.0	717.27	692.53
	delay 95%:	947.76	2.0	988.35	907.15
	delay 98%:	1078.7	2.0	1611.71	1164.54
	delay 99%:	1147.99	2.0	2303.81	1349.24
	delay 100%:	277.09	2.0	198281.25	788.74

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	62749	216061	61892	213000	98.63%	98.58%	857	3061	N/A	N/A
	Data	68473	215561	67749	213000	98.94%	98.81%	723	2561	N/A	499
Route	Net Data	71260	224246	70535	221685	98.98%	98.86%	723	2561	0	0
	Net Control	505	11781	2749	68293	544.36%	579.69%	N/A	N/A	N/A	N/A
	→ *peer2peer*	135	2776	104	2149	77.04%	77.41%	29	626	N/A	N/A
	→ *broadcast*	369	9005	2644	66143	716.53%	734.51%	N/A	N/A	N/A	N/A
MAC	Net Sum	71765	236028	73286	289978	102.12%	122.86%	N/A	-53950	2418	18737

Table 5.13.: Average Result Table: Run 00020

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 989.6
Energy / UserData: 5.7 μJ/B
Throughput: 349.184 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1132.324	496.0	15782.4	962.193
	rcvDataEnergy	520.442	346.8	5860.32	346.003
Control:	sendCtrlEnergy	376.904	310.8	556.8	83.159
	rcvCtrlEnergy	85.534	72.0	360.24	49.461
	→ *sendCtrlP2PEnergy*	518.509	488.4	556.8	20.567
	→ *rcvCtrlP2PEnergy*	351.832	345.12	360.24	4.602
	→ *sendCtrlBCEnergy*	330.12	310.8	379.2	15.838
	→ *rcvCtrlBCEnergy*	76.403	72.0	90.0	4.205
Miscellaneous:	batteryLevel (%):	0.522	0.0	5.931	0.91
	delay 80%:	3796.97	2.0	3038.14	3822.22
	delay 90%:	5332.83	2.0	4536.77	5695.09
	delay 95%:	6382.74	2.0	6003.86	7129.93
	delay 98%:	7217.17	2.0	8083.1	8460.99
	delay 99%:	7574.33	2.0	9705.22	9133.6
	delay 100%:	1636.11	2.0	71348.0	2282.18

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	43729	150662	43167	148544	98.71%	98.59%	561	2118	N/A	N/A
	Data	47888	150513	47307	148544	98.79%	98.69%	580	1969	N/A	148
Route	Net Data	54775	171305	54194	169336	98.94%	98.85%	580	1969	0	0
	Net Control	469	10791	2637	65575	562.26%	607.68%	N/A	N/A	N/A	N/A
	→ *peer2peer*	134	2673	110	2172	82.09%	81.26%	24	501	N/A	N/A
	→ *broadcast*	333	8117	2526	63403	758.56%	781.11%	N/A	N/A	N/A	N/A
MAC	Net Sum	55245	182097	56831	234911	102.87%	129.0%	N/A	-52814	2281	20588

Table 5.14.: Average Result Table: Run 00024

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.07 μJ/B
Throughput: 413.156 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1006.109	496.0	10874.4	536.306
	rcvDataEnergy	464.042	346.8	4063.92	138.463
Control:	sendCtrlEnergy	379.14	310.8	556.8	81.487
	rcvCtrlEnergy	87.407	72.0	360.24	51.998
	→ *sendCtrlP2PEnergy*	515.844	488.4	556.8	18.743
	→ *rcvCtrlP2PEnergy*	351.261	345.12	360.24	4.206
	→ *sendCtrlBCEnergy*	332.794	310.8	379.2	17.215
	→ *rcvCtrlBCEnergy*	77.216	72.0	90.0	4.561
Miscellaneous:	batteryLevel (%):	2.297	0.0	25.349	3.467
	delay 80%:	537.43	2.0	363.33	415.43
	delay 90%:	699.79	2.0	602.4	612.33
	delay 95%:	829.35	2.0	886.98	814.81
	delay 98%:	946.34	2.0	1393.96	1042.83
	delay 99%:	1006.32	2.0	2005.75	1197.64
	delay 100%:	234.63	2.0	51285.15	577.29

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	52428	180443	51643	177838	98.5%	98.56%	784	2605	N/A	N/A
	Data	57120	179892	56519	177838	98.95%	98.86%	600	2054	N/A	551
Route	Net Data	58249	183477	57648	181423	98.97%	98.88%	600	2054	0	0
	Net Control	459	10362	2312	55411	503.7%	534.75%	N/A	N/A	N/A	N/A
	→ *peer2peer*	129	2636	101	2052	78.29%	77.85%	27	584	N/A	N/A
	→ *broadcast*	328	7726	2210	53358	673.78%	690.63%	N/A	N/A	N/A	N/A
MAC	Net Sum	58709	193840	59962	236834	102.13%	122.18%	N/A	-42994	2019	15529

Table 5.15.: Average Result Table: Run 00026

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.11 μJ/B
Throughput: 590.746 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1018.758	496.0	15782.4	565.121
	rcvDataEnergy	469.748	346.8	5860.32	153.762
Control:	sendCtrlEnergy	362.548	310.8	556.8	73.136
	rcvCtrlEnergy	83.993	72.0	360.24	45.151
	→ *sendCtrlP2PEnergy*	512.554	488.4	556.8	19.06
	→ *rcvCtrlP2PEnergy*	350.491	345.12	360.24	4.307
	→ *sendCtrlBCEnergy*	328.584	310.8	379.2	15.584
	→ *rcvCtrlBCEnergy*	76.401	72.0	90.0	4.103
Miscellaneous:	batteryLevel (%):	10.831	0.0	52.978	11.711
	delay 80%:	519.04	2.0	283.24	361.47
	delay 90%:	628.76	2.0	385.95	462.64
	delay 95%:	705.05	2.0	587.34	558.24
	delay 98%:	778.53	2.0	924.4	690.88
	delay 99%:	814.9	2.0	1296.71	777.14
	delay 100%:	183.77	2.0	68732.15	424.85

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	74779	257553	73842	254457	98.75%	98.8%	937	3096	N/A	N/A
	Data	81548	256869	80831	254457	99.12%	99.06%	716	2412	N/A	684
Route	Net Data	84210	265108	83492	262695	99.15%	99.09%	716	2412	0	0
	Net Control	490	11758	2483	62002	506.73%	527.32%	N/A	N/A	N/A	N/A
	→ *peer2peer*	102	2173	81	1709	79.41%	78.65%	21	464	N/A	N/A
	→ *broadcast*	386	9585	2402	60292	622.28%	629.02%	N/A	N/A	N/A	N/A
MAC	Net Sum	84700	276866	85976	324698	101.51%	117.28%	N/A	-47831	2135	15608

Table 5.16.: Average Result Table: Run 00028

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 3.25 μJ/B
Throughput: 769.854 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	2563.995	496.0	40396.8	2190.716
	rcvDataEnergy	813.082	346.8	11064.24	504.919
Control:	sendCtrlEnergy	369.297	310.8	556.8	77.321
	rcvCtrlEnergy	84.728	72.0	360.24	46.608
	→ *sendCtrlP2PEnergy*	514.837	488.4	556.8	19.313
	→ *rcvCtrlP2PEnergy*	350.927	345.12	360.24	4.368
	→ *sendCtrlBCEnergy*	330.001	310.8	379.2	15.567
	→ *rcvCtrlBCEnergy*	76.618	72.0	90.0	4.056
Miscellaneous:	batteryLevel (%):	4.522	0.0	59.589	7.734
	delay 80%:	1148.96	2.0	885.37	931.47
	delay 90%:	1498.12	2.0	1612.09	1338.51
	delay 95%:	1765.04	2.0	2443.95	1738.76
	delay 98%:	2016.95	2.0	4039.59	2235.26
	delay 99%:	2147.13	2.0	5737.22	2576.66
	delay 100%:	505.18	2.0	113977.87	1291.34

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	98443	93995	96231	91999	97.75%	97.88%	2211	1996	N/A	N/A
	Data	100324	93525	98761	91999	98.44%	98.37%	1562	1526	N/A	470
Route	Net Data	104459	97438	102896	95912	98.5%	98.43%	1562	1526	0	0
	Net Control	553	12939	2945	72595	532.55%	561.06%	N/A	N/A	N/A	N/A
	→ *peer2peer*	133	2750	103	2135	77.44%	77.64%	29	614	N/A	N/A
	→ *broadcast*	418	10189	2840	70460	679.43%	691.53%	N/A	N/A	N/A	N/A
MAC	Net Sum	105012	110377	105841	168507	100.79%	152.66%	N/A	-58130	3407	19382

Table 5.17.: Average Result Table: Run 00102

Algorithm: Beehive, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 2.9 μJ/B
Throughput: 781.95 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	4581.055	496.0	58541.6	4248.579
	rcvDataEnergy	1255.537	346.8	14600.88	951.559
Control:	sendCtrlEnergy	371.444	310.8	556.8	78.242
	rcvCtrlEnergy	84.779	72.0	360.24	45.229
	→ *sendCtrlP2PEnergy*	516.849	488.4	556.8	18.367
	→ *rcvCtrlP2PEnergy*	351.435	345.12	360.24	4.154
	→ *sendCtrlBCEnergy*	331.085	310.8	379.2	15.359
	→ *rcvCtrlBCEnergy*	77.147	72.0	90.0	4.05
Miscellaneous:	batteryLevel (%):	5.432	0.0	66.56	9.005
	delay 80%:	1972.47	2.0	1307.34	1558.8
	delay 90%:	2517.79	2.0	2236.4	2157.88
	delay 95%:	2921.24	2.0	3309.64	2732.67
	delay 98%:	3294.08	2.0	4895.14	3434.15
	delay 99%:	3485.32	2.0	7149.76	3914.68
	delay 100%:	793.23	2.0	199195.64	1649.16

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	101128	49081	97743	47491	96.65%	96.76%	3384	1589	N/A	N/A
	Data	101588	48708	99048	47491	97.5%	97.5%	2540	1217	N/A	372
Route	Net Data	105074	50465	102534	49248	97.58%	97.59%	2540	1217	0	0
	Net Control	644	14789	3588	85834	557.14%	580.39%	N/A	N/A	N/A	N/A
	→ *peer2peer*	160	3215	116	2344	72.5%	72.91%	43	871	N/A	N/A
	→ *broadcast*	483	11573	3471	83490	718.63%	721.42%	N/A	N/A	N/A	N/A
MAC	Net Sum	105719	65254	106123	135083	100.38%	207.01%	N/A	-69828	4853	24418

Table 5.18.: Average Result Table: Run 00202

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.97 μJ/B
Throughput: 433.604 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1563.77	496.0	77860.8	1949.059
	rcvDataEnergy	716.584	346.8	27972.48	759.037
Control:	sendCtrlEnergy	524.709	480.8	1438.4	42.296
	rcvCtrlEnergy	157.636	72.0	582.0	111.144
	→ *sendCtrlP2PEnergy*	524.709	480.8	2192.8	42.296
	→ *rcvCtrlP2PEnergy*	358.773	345.12	582.0	15.899
	→ *sendCtrlBCEnergy*	0.0	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	100.81	72.0	530.0	33.528
Miscellaneous:	batteryLevel (%):	1.356	0.0	28.807	3.736
	delay 80%:	838.67	1.94	978.71	925.0
	delay 90%:	1392.9	1.94	2248.08	1858.82
	delay 95%:	1981.47	1.94	4598.89	3139.03
	delay 98%:	2727.96	1.94	10053.77	5331.22
	delay 99%:	3236.16	1.94	18010.95	7364.03
	delay 100%:	969.39	1.94	492376.73	4719.31

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	54673	188607	54200	187138	99.13%	99.22%	472	1469	N/A	N/A
	Data	64105	208495	58141	187138	90.7%	89.76%	5963	21357	N/A	-19888
Route	Net Data	95530	311112	83500	270441	87.41%	86.93%	12029	40671	0	0
	Net Control	3570	46063	10096	122313	282.8%	265.53%	N/A	N/A	N/A	N/A
	→ *peer2peer*	1225	22762	1816	27010	148.24%	118.66%	-590	-4247	N/A	N/A
	→ *broadcast*	0	0	8279	95303	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	99101	357176	93597	392755	94.45%	109.96%	N/A	-35579	5914	33609

Table 5.19.: Average Result Table: Run 00301

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.87 μJ/B
Throughput: 386.728 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1492.36	496.0	93248.0	2209.278
	rcvDataEnergy	679.823	346.8	33094.56	851.894
Control:	sendCtrlEnergy	538.334	480.8	2023.6	63.981
	rcvCtrlEnergy	167.865	72.0	701.28	121.428
	→ *sendCtrlP2PEnergy*	538.334	480.8	3097.2	63.981
	→ *rcvCtrlP2PEnergy*	366.939	345.12	701.28	27.102
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	97.575	72.0	444.0	28.35
Miscellaneous:	batteryLevel (%):	1.301	0.0	30.795	3.555
	delay 80%:	601.78	1.94	548.43	571.95
	delay 90%:	974.03	1.94	1510.92	1241.66
	delay 95%:	1431.98	1.94	3784.72	2350.32
	delay 98%:	2114.52	1.94	10290.27	4646.46
	delay 99%:	2670.09	1.94	21943.22	7288.88
	delay 100%:	1041.86	1.94	740913.04	7046.26

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	48889	168468	48340	166871	98.88%	99.05%	548	1596	N/A	N/A
	Data	58167	187701	51834	166871	89.11%	88.9%	6332	20830	N/A	-19233
Route	Net Data	84400	269759	71109	228998	84.25%	84.89%	13290	40761	0	0
	Net Control	9338	99356	18498	224335	198.09%	225.79%	N/A	N/A	N/A	N/A
	→ *peer2peer*	2705	44427	5005	58163	185.03%	130.92%	-2300	-13736	N/A	N/A
	→ *broadcast*	0	0	13491	166172	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	93740	369115	89608	453333	95.59%	122.82%	N/A	-84218	8563	54929

Table 5.20.: Average Result Table: Run 00302

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 947.8
Energy / UserData: 8.53 μJ/B
Throughput: 352.888 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1566.994	496.0	88642.0	2565.577
	rcvDataEnergy	710.561	346.8	31776.96	994.287
Control:	sendCtrlEnergy	545.358	480.8	1955.2	70.316
	rcvCtrlEnergy	175.996	72.0	763.44	125.217
	→ *sendCtrlP2PEnergy*	545.358	480.8	3378.4	70.316
	→ *rcvCtrlP2PEnergy*	369.488	345.12	763.44	27.577
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	99.228	72.0	444.0	29.22
Miscellaneous:	batteryLevel (%):	1.04	0.0	18.192	2.908
	delay 80%:	528.03	1.77	458.66	455.41
	delay 90%:	805.65	1.77	1209.67	938.83
	delay 95%:	1160.36	1.77	2811.56	1813.88
	delay 98%:	1701.37	1.77	7257.06	3663.67
	delay 99%:	2158.34	1.77	16434.52	5913.79
	delay 100%:	946.46	1.77	469996.26	7227.28

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	44754	154024	44110	152180	98.56%	98.8%	643	1844	N/A	N/A
	Data	53216	170527	47296	152180	88.88%	89.24%	5920	18347	N/A	-16502
Route	Net Data	80791	254798	67252	215081	83.24%	84.41%	13539	39717	0	0
	Net Control	14979	152336	26883	310119	179.47%	203.58%	N/A	N/A	N/A	N/A
	→ *peer2peer*	4100	63629	8091	87967	197.34%	138.25%	-3990	-24337	N/A	N/A
	→ *broadcast*	0	0	18791	222151	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	95771	407135	94135	525200	98.29%	129.0%	N/A	-118065	10662	69472

Table 5.21.: Average Result Table: Run 00303

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 9.09 μJ/B
Throughput: 326.558 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1618.426	496.0	99123.2	2836.109
	rcvDataEnergy	731.041	346.8	35341.92	1092.145
Control:	sendCtrlEnergy	549.094	480.8	2426.4	72.696
	rcvCtrlEnergy	180.067	72.0	827.28	126.088
	→ *sendCtrlP2PEnergy*	549.094	480.8	3978.8	72.696
	→ *rcvCtrlP2PEnergy*	371.473	345.12	827.28	29.721
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	102.011	72.0	404.0	31.437
Miscellaneous:	batteryLevel (%):	1.162	0.0	16.287	2.768
	delay 80%:	502.65	1.94	424.48	418.47
	delay 90%:	744.33	1.94	1183.73	825.8
	delay 95%:	1051.97	1.94	2791.74	1582.19
	delay 98%:	1536.55	1.94	8004.99	3278.0
	delay 99%:	1981.36	1.94	19625.34	5608.07
	delay 100%:	1001.45	1.94	382769.89	8107.46

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	41502	142717	40819	140817	98.35%	98.67%	682	1899	N/A	N/A
	Data	49899	159188	43768	140817	87.71%	88.46%	6130	18370	N/A	-16470
Route	Net Data	77595	242336	63225	201362	81.48%	83.09%	14369	40974	0	0
	Net Control	21054	204751	36512	396885	173.42%	193.84%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5355	80707	11088	114908	207.06%	142.38%	-5732	-34201	N/A	N/A
	→ *broadcast*	0	0	25424	281976	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	98650	447087	99738	598247	101.1%	133.81%	N/A	-151159	12955	84788

Table 5.22.: Average Result Table: Run 00304

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 8.57 μJ/B
Throughput: 345.434 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1519.059	496.0	108638.4	2665.155
	rcvDataEnergy	688.628	346.8	37445.28	1028.619
Control:	sendCtrlEnergy	551.924	480.8	2076.8	77.889
	rcvCtrlEnergy	182.74	72.0	830.64	128.099
	→ *sendCtrlP2PEnergy*	551.924	480.8	2076.8	77.889
	→ *rcvCtrlP2PEnergy*	372.552	345.12	830.64	30.216
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	101.002	72.0	444.0	30.129
Miscellaneous:	batteryLevel (%):	1.114	0.0	17.261	2.942
	delay 80%:	485.26	1.94	364.0	387.42
	delay 90%:	705.36	1.94	1024.0	755.83
	delay 95%:	986.36	1.94	2666.2	1442.89
	delay 98%:	1442.01	1.94	8968.96	3069.56
	delay 99%:	1853.33	1.94	23607.15	5203.83
	delay 100%:	881.0	1.94	527869.33	7054.44

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	43846	150870	43179	148949	98.48%	98.73%	666	1920	N/A	N/A
	Data	51761	166021	46261	148949	89.37%	89.72%	5499	17071	N/A	-15151
Route	Net Data	76522	241507	63500	203531	82.98%	84.28%	13021	37975	0	0
	Net Control	22139	212085	36271	397488	163.83%	187.42%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5646	83302	11830	119564	209.53%	143.53%	-6183	-36261	N/A	N/A
	→ *broadcast*	0	0	24440	277924	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	98662	453592	99772	601020	101.13%	132.5%	N/A	-147427	12669	85996

Table 5.23.: Average Result Table: Run 00306

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 8.67 μJ/B
Throughput: 347.834 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1541.472	496.0	98885.6	2697.368
	rcvDataEnergy	698.641	346.8	35052.24	1047.975
Control:	sendCtrlEnergy	550.345	480.8	2373.2	75.58
	rcvCtrlEnergy	180.887	72.0	825.6	126.346
	→ *sendCtrlP2PEnergy*	550.345	480.8	4305.6	75.58
	→ *rcvCtrlP2PEnergy*	371.5	345.12	825.6	29.09
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	101.813	72.0	404.0	30.964
Miscellaneous:	batteryLevel (%):	1.01	0.0	14.585	2.459
	delay 80%:	507.14	1.94	371.35	397.04
	delay 90%:	717.81	1.94	922.38	730.95
	delay 95%:	981.47	1.94	2267.45	1368.33
	delay 98%:	1420.26	1.94	7632.19	2947.58
	delay 99%:	1833.84	1.94	17949.2	5159.82
	delay 100%:	900.16	1.94	396489.8	7479.6

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	44132	151903	43478	150013	98.52%	98.76%	653	1889	N/A	N/A
	Data	52212	167755	46581	150013	89.22%	89.42%	5629	17742	N/A	-15852
Route	Net Data	77899	246287	64604	207113	82.93%	84.09%	13295	39173	0	0
	Net Control	21496	209510	36701	400637	170.73%	191.23%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5563	83012	11327	117271	203.61%	141.27%	-5763	-34259	N/A	N/A
	→ *broadcast*	0	0	25373	283366	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	99396	455797	101306	607751	101.92%	133.34%	N/A	-151953	12918	87641

Table 5.24.: Average Result Table: Run 00308

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 8.53 μJ/B
Throughput: 353.974 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1517.219	496.0	100665.6	2669.267
	rcvDataEnergy	688.128	346.8	35052.24	1031.802
Control:	sendCtrlEnergy	549.519	480.8	2373.2	72.932
	rcvCtrlEnergy	179.671	72.0	797.04	126.569
	→ *sendCtrlP2PEnergy*	549.519	480.8	3135.2	72.932
	→ *rcvCtrlP2PEnergy*	371.445	345.12	797.04	29.63
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	100.778	72.0	404.0	30.038
Miscellaneous:	batteryLevel (%):	1.001	0.0	14.848	2.574
	delay 80%:	496.55	1.94	345.4	381.94
	delay 90%:	698.85	1.94	826.11	702.51
	delay 95%:	948.94	1.94	1976.09	1300.76
	delay 98%:	1351.47	1.94	5988.11	2716.94
	delay 99%:	1702.28	1.94	13545.93	4479.71
	delay 100%:	832.26	1.94	431635.79	7359.98

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	44886	154544	44246	152651	98.57%	98.78%	640	1892	N/A	N/A
	Data	52639	169484	47392	152651	90.03%	90.07%	5246	16832	N/A	-14940
Route	Net Data	77621	246107	64805	208120	83.49%	84.56%	12815	37987	0	0
	Net Control	21315	208686	35792	396038	167.92%	189.78%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5525	82927	11176	115488	202.28%	139.26%	-5650	-32561	N/A	N/A
	→ *broadcast*	0	0	24616	280549	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	98937	454794	100598	604158	101.68%	132.84%	N/A	-149363	12664	87282

Table 5.25.: Average Result Table: Run 00310

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 8.53 μJ/B
Throughput: 353.974 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1517.219	496.0	100665.6	2669.267
	rcvDataEnergy	688.128	346.8	35052.24	1031.802
Control:	sendCtrlEnergy	549.519	480.8	2373.2	72.932
	rcvCtrlEnergy	179.671	72.0	797.04	126.569
	→ *sendCtrlP2PEnergy*	549.519	480.8	3135.2	72.932
	→ *rcvCtrlP2PEnergy*	371.445	345.12	797.04	29.63
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	100.778	72.0	404.0	30.038
Miscellaneous:	batteryLevel (%):	1.001	0.0	14.848	2.574
	delay 80%:	496.55	1.94	345.4	381.94
	delay 90%:	698.85	1.94	826.11	702.51
	delay 95%:	948.94	1.94	1976.09	1300.76
	delay 98%:	1351.47	1.94	5988.11	2716.94
	delay 99%:	1702.28	1.94	13545.93	4479.71
	delay 100%:	832.26	1.94	431635.79	7359.98

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	44886	154544	44246	152651	98.57%	98.78%	640	1892	N/A	N/A
	Data	52639	169484	47392	152651	90.03%	90.07%	5246	16832	N/A	-14940
Route	Net Data	77621	246107	64805	208120	83.49%	84.56%	12815	37987	0	0
	Net Control	21315	208686	35792	396038	167.92%	189.78%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5525	82927	11176	115488	202.28%	139.26%	-5650	-32561	N/A	N/A
	→ *broadcast*	0	0	24616	280549	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	98937	454794	100598	604158	101.68%	132.84%	N/A	-149363	12664	87282

Table 5.26.: Average Result Table: Run 00316

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 991.8
Energy / UserData: 7.64 μJ/B
Throughput: 366.204 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1337.693	496.0	100445.2	2179.83
	rcvDataEnergy	609.582	346.8	35356.56	842.835
Control:	sendCtrlEnergy	540.239	480.8	2054.0	62.226
	rcvCtrlEnergy	170.638	72.0	691.2	124.274
	→ *sendCtrlP2PEnergy*	540.239	480.8	3849.6	62.226
	→ *rcvCtrlP2PEnergy*	366.612	345.12	691.2	25.156
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	94.886	72.0	402.0	24.675
Miscellaneous:	batteryLevel (%):	0.725	0.0	12.589	1.798
	delay 80%:	477.5	1.92	325.35	357.59
	delay 90%:	673.79	1.92	885.89	680.81
	delay 95%:	943.17	1.92	2408.26	1375.59
	delay 98%:	1445.44	1.92	9177.55	3318.75
	delay 99%:	1931.9	1.92	25591.45	5988.75
	delay 100%:	991.72	1.92	503066.68	8380.53

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	45957	158342	45379	156626	98.74%	98.92%	577	1716	N/A	N/A
	Data	53970	173432	48590	156626	90.03%	90.31%	5379	16806	N/A	-15090
Route	Net Data	70928	226104	59786	192944	84.29%	85.33%	11141	33160	0	0
	Net Control	16523	183690	29369	373926	177.75%	203.56%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5313	85979	8845	103818	166.48%	120.75%	-3531	-17839	N/A	N/A
	→ *broadcast*	0	0	20524	270107	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	87452	409795	89156	566870	101.95%	138.33%	N/A	-157075	11731	89907

Table 5.27.: Average Result Table: Run 00320

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 6.07 μJ/B
Throughput: 327.65 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1191.235	496.0	59253.2	1054.179
	rcvDataEnergy	547.139	346.8	21792.72	386.691
Control:	sendCtrlEnergy	519.972	480.8	1567.6	38.231
	rcvCtrlEnergy	153.772	72.0	659.28	113.27
	→ *sendCtrlP2PEnergy*	519.972	480.8	2314.4	38.231
	→ *rcvCtrlP2PEnergy*	355.578	345.12	659.28	13.323
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	93.763	72.0	402.0	29.245
Miscellaneous:	batteryLevel (%):	0.204	0.0	4.865	0.528
	delay 80%:	2876.32	1.94	3016.55	3258.15
	delay 90%:	4677.24	1.94	6038.22	6097.36
	delay 95%:	6353.28	1.94	10984.52	9375.56
	delay 98%:	8303.64	1.94	22221.16	14563.74
	delay 99%:	9471.98	1.94	32571.99	18605.47
	delay 100%:	2379.16	1.94	195908.43	6408.78

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	41413	142784	40955	141451	98.89%	99.07%	456	1333	N/A	N/A
	Data	50238	161431	43904	141451	87.39%	87.62%	6333	19980	N/A	-18647
Route	Net Data	59018	190543	51145	165748	86.66%	86.99%	7872	24795	0	0
	Net Control	1596	22017	4454	60367	279.07%	274.18%	N/A	N/A	N/A	N/A
	→ *peer2peer*	666	12873	813	13462	122.07%	104.58%	-145	-589	N/A	N/A
	→ *broadcast*	0	0	3640	46904	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	60615	212560	55600	226115	91.73%	106.38%	N/A	-13554	2591	17790

Table 5.28.: Average Result Table: Run 00324

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 9.63 μJ/B
Throughput: 236.354 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1592.227	496.0	111882.0	3085.583
	rcvDataEnergy	718.516	346.8	38873.76	1191.163
Control:	sendCtrlEnergy	548.613	480.8	2160.4	70.675
	rcvCtrlEnergy	203.571	72.0	755.04	134.2
	→ *sendCtrlP2PEnergy*	548.613	480.8	3272.0	70.675
	→ *rcvCtrlP2PEnergy*	369.445	345.12	755.04	26.579
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	100.238	72.0	530.0	31.276
Miscellaneous:	batteryLevel (%):	0.785	0.0	12.648	1.906
	delay 80%:	500.74	1.94	431.76	404.94
	delay 90%:	791.36	1.94	1333.88	968.28
	delay 95%:	1221.93	1.94	3471.52	2147.25
	delay 98%:	2035.93	1.94	12725.59	5366.99
	delay 99%:	2919.33	1.94	33331.07	10546.04
	delay 100%:	1314.46	1.94	385909.3	8930.88

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	30124	103550	29543	101775	98.07%	98.29%	579	1774	N/A	N/A
	Data	37282	119950	31685	101775	84.99%	84.85%	5595	18174	N/A	-16399
Route	Net Data	56684	178226	43858	139839	77.37%	78.46%	12825	38386	0	0
	Net Control	21288	219847	29691	335577	139.47%	152.64%	N/A	N/A	N/A	N/A
	→ *peer2peer*	6245	94500	11810	128779	189.11%	136.27%	-5565	-34279	N/A	N/A
	→ *broadcast*	0	0	17880	206798	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	77973	398073	73550	475416	94.33%	119.43%	N/A	-77343	12720	89478

Table 5.29.: Average Result Table: Run 00326

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 8.33 μJ/B
Throughput: 410.035 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1510.881	496.0	96622.4	2632.222
	rcvDataEnergy	686.382	346.8	33840.48	1022.52
Control:	sendCtrlEnergy	557.209	480.8	2373.2	82.018
	rcvCtrlEnergy	177.37	72.0	844.08	123.363
	→ *sendCtrlP2PEnergy*	557.209	480.8	2373.2	82.018
	→ *rcvCtrlP2PEnergy*	375.356	345.12	844.08	32.386
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	106.473	72.0	486.0	35.266
Miscellaneous:	batteryLevel (%):	2.956	0.0	40.229	6.443
	delay 80%:	385.66	1.84	336.96	289.64
	delay 90%:	541.71	1.84	775.23	540.1
	delay 95%:	742.32	1.84	1943.18	1034.39
	delay 98%:	1075.3	1.84	5758.92	2231.42
	delay 99%:	1360.47	1.84	12412.06	3648.89
	delay 100%:	726.68	1.84	575495.41	5501.15

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	51837	178681	51254	176879	98.88%	98.99%	582	1801	N/A	N/A
	Data	60873	196120	54914	176879	90.21%	90.19%	5958	19241	N/A	-17439
Route	Net Data	89501	283833	75178	241157	84.0%	84.96%	14322	42676	0	0
	Net Control	25295	223987	46606	463397	184.25%	206.89%	N/A	N/A	N/A	N/A
	→ *peer2peer*	5604	79493	12883	121902	229.89%	153.35%	-7278	-42409	N/A	N/A
	→ *broadcast*	0	0	33722	341495	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	114797	507820	121785	704554	106.09%	138.74%	N/A	-196734	14737	93692

Table 5.30.: Average Result Table: Run 00328

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 4.77 μJ/B
Throughput: 623.53 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	3544.562	496.0	248263.2	6307.773
	rcvDataEnergy	1114.646	346.8	67761.36	1677.716
Control:	sendCtrlEnergy	538.362	480.8	1575.2	61.071
	rcvCtrlEnergy	159.242	72.0	654.24	117.398
	→ *sendCtrlP2PEnergy*	538.362	480.8	3074.4	61.071
	→ *rcvCtrlP2PEnergy*	365.505	345.12	654.24	23.915
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	95.785	72.0	404.0	27.245
Miscellaneous:	batteryLevel (%):	1.134	0.0	28.936	3.156
	delay 80%:	1028.32	1.94	956.29	841.35
	delay 90%:	1492.72	1.94	2347.52	1599.78
	delay 95%:	2072.84	1.94	5487.4	2999.43
	delay 98%:	2955.48	1.94	16545.61	5996.32
	delay 99%:	3657.42	1.94	36561.81	9286.25
	delay 100%:	1370.72	1.94	498941.47	8790.6

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	79491	76037	77941	74752	98.05%	98.31%	1550	1285	N/A	N/A
	Data	92139	86058	79503	74752	86.29%	86.86%	12634	11306	N/A	-10020
Route	Net Data	126905	118267	102888	97280	81.07%	82.25%	24016	20986	0	0
	Net Control	9339	106515	18619	236040	199.37%	221.6%	N/A	N/A	N/A	N/A
	→ *peer2peer*	3024	49792	4562	55308	150.86%	111.08%	-1537	-5515	N/A	N/A
	→ *broadcast*	0	0	14056	180731	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	136246	224782	121508	333320	89.18%	148.29%	N/A	-108538	12129	63047

Table 5.31.: Average Result Table: Run 00402

Algorithm: DSR, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 4.34 μJ/B
Throughput: 695.842 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	6485.674	496.0	373802.8	12854.403
	rcvDataEnergy	1761.842	346.8	90997.92	3087.834
Control:	sendCtrlEnergy	537.459	480.8	1962.8	59.536
	rcvCtrlEnergy	157.739	72.0	659.28	116.448
	→ *sendCtrlP2PEnergy*	537.459	480.8	2778.0	59.536
	→ *rcvCtrlP2PEnergy*	364.926	345.12	659.28	23.223
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	95.419	72.0	404.0	26.374
Miscellaneous:	batteryLevel (%):	1.129	0.0	31.093	3.199
	delay 80%:	1508.73	1.94	1246.45	1183.45
	delay 90%:	2135.61	1.94	3119.39	2188.94
	delay 95%:	2907.72	1.94	7151.68	4019.62
	delay 98%:	4028.25	1.94	16074.45	7672.03
	delay 99%:	4881.16	1.94	30377.58	11552.62
	delay 100%:	1781.18	1.94	517215.45	10736.96

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	89432	43558	86979	42461	97.26%	97.48%	2451	1096	N/A	N/A
	Data	107577	51198	87876	42461	81.69%	82.93%	19700	8736	N/A	-7640
Route	Net Data	148939	70777	114119	55392	76.62%	78.26%	34819	15385	0	0
	Net Control	9121	106560	17898	229437	196.23%	215.31%	N/A	N/A	N/A	N/A
	→ *peer2peer*	3049	50549	4298	52930	140.96%	104.71%	-1248	-2380	N/A	N/A
	→ *broadcast*	0	0	13599	176507	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	158061	177338	132018	284829	83.52%	160.61%	N/A	-107491	15461	63628

Table 5.32.: Average Result Table: Run 00502

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 999.4
Energy / UserData: 7.56 μJ/B
Throughput: 448.396 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1428.285	496.0	1756549.2	3340.975
	rcvDataEnergy	665.315	355.2	1215149.76	2242.972
Control:	sendCtrlEnergy	350.501	310.8	584.4	39.328
	rcvCtrlEnergy	82.587	72.0	348.48	27.027
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.529	310.8	584.4	5.229
	→ *rcvCtrlBCEnergy*	79.837	72.0	144.0	1.222
Miscellaneous:	batteryLevel (%):	1.811	0.0	34.288	4.727
	delay 80%:	784.23	1.71	691.05	785.73
	delay 90%:	1102.62	1.71	1133.23	1179.66
	delay 95%:	1346.54	1.71	1696.63	1554.07
	delay 98%:	1575.41	1.71	2650.65	2010.85
	delay 99%:	1690.52	1.71	3547.68	2305.38
	delay 100%:	387.96	1.71	171176.77	799.23

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	57902	197733	56001	190297	96.72%	96.24%	1900	7436	N/A	N/A
	Data	61706	197556	0	0	0.0%	0.0%	61706	197556	N/A	177
Route	Net Data	87042	277585	25323	79972	29.09%	28.81%	61718	197613	0	0
	Net Control	3115	67485	27560	592454	884.75%	877.9%	N/A	N/A	N/A	N/A
	→ *peer2peer*	169	3952	259	6062	153.25%	153.39%	-90	-2109	N/A	N/A
	→ *broadcast*	2828	60796	27299	586392	965.31%	964.52%	N/A	N/A	N/A	N/A
MAC	Net Sum	90158	345070	52883	672426	58.66%	194.87%	N/A	-327356	17667	174449

Table 5.33.: Average Result Table: Run 00601

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.51 μJ/B
Throughput: 407.07 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1399.368	496.0	1724318.8	3339.387
	rcvDataEnergy	651.165	355.2	1192847.04	2251.769
Control:	sendCtrlEnergy	350.607	310.8	630.0	39.323
	rcvCtrlEnergy	82.259	72.0	348.48	25.306
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.588	310.8	630.0	4.949
	→ *rcvCtrlBCEnergy*	79.838	72.0	156.0	1.188
Miscellaneous:	batteryLevel (%):	2.421	0.0	42.671	6.126
	delay 80%:	707.29	1.69	527.99	607.53
	delay 90%:	951.99	1.69	900.3	911.57
	delay 95%:	1148.5	1.69	1326.25	1225.07
	delay 98%:	1335.59	1.69	2149.69	1611.44
	delay 99%:	1434.91	1.69	3087.29	1885.31
	delay 100%:	339.14	1.69	227085.43	831.23

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	52807	180057	50883	172929	96.36%	96.04%	1923	7127	N/A	N/A
	Data	56266	179892	0	0	0.0%	0.0%	56266	179892	N/A	164
Route	Net Data	78116	248802	21832	68836	27.95%	27.67%	56282	179965	0	0
	Net Control	3491	75548	33325	716235	954.6%	948.05%	N/A	N/A	N/A	N/A
	→ *peer2peer*	192	4491	274	6403	142.71%	142.57%	-81	-1911	N/A	N/A
	→ *broadcast*	3184	68392	33050	709832	1038.0%	1037.89%	N/A	N/A	N/A	N/A
MAC	Net Sum	81607	324351	55158	785071	67.59%	242.04%	N/A	-460720	21654	215786

Table 5.34.: Average Result Table: Run 00602

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.9 μJ/B
Throughput: 407.626 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1448.455	496.0	1805902.0	2926.114
	rcvDataEnergy	672.125	355.2	1249300.8	1843.423
Control:	sendCtrlEnergy	353.567	310.8	630.0	44.135
	rcvCtrlEnergy	82.688	72.0	348.48	27.488
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.635	310.8	630.0	4.864
	→ *rcvCtrlBCEnergy*	79.847	72.0	156.0	1.173
Miscellaneous:	batteryLevel (%):	1.644	0.0	25.547	4.148
	delay 80%:	637.6	1.94	477.46	509.68
	delay 90%:	827.18	1.94	763.14	728.46
	delay 95%:	979.01	1.94	1132.05	964.24
	delay 98%:	1130.75	1.94	1764.72	1287.24
	delay 99%:	1214.71	1.94	2664.52	1531.48
	delay 100%:	296.17	1.94	583093.79	1027.9

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	53109	180708	50952	173279	95.94%	95.89%	2155	7429	N/A	N/A
	Data	56561	180521	0	0	0.0%	0.0%	56561	180521	N/A	187
Route	Net Data	81415	258360	24786	77638	30.44%	30.05%	56628	180721	0	0
	Net Control	4433	96012	38911	836058	877.76%	870.78%	N/A	N/A	N/A	N/A
	→ *peer2peer*	319	7438	376	8783	117.87%	118.08%	-57	-1344	N/A	N/A
	→ *broadcast*	3971	85259	38534	827275	970.39%	970.31%	N/A	N/A	N/A	N/A
MAC	Net Sum	85849	354372	63698	913696	74.2%	257.84%	N/A	-559324	23197	229489

Table 5.35.: Average Result Table: Run 00603

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.72 μJ/B
Throughput: 433.45 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1426.094	496.0	2051658.8	3626.88
	rcvDataEnergy	661.484	355.2	1419359.04	2401.51
Control:	sendCtrlEnergy	353.329	310.8	599.6	44.133
	rcvCtrlEnergy	83.155	72.0	348.48	29.761
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.53	310.8	599.6	5.256
	→ *rcvCtrlBCEnergy*	79.816	72.0	148.0	1.293
Miscellaneous:	batteryLevel (%):	1.409	0.0	29.175	3.397
	delay 80%:	578.48	1.81	342.06	442.53
	delay 90%:	737.25	1.81	549.61	619.69
	delay 95%:	862.08	1.81	851.05	807.25
	delay 98%:	984.96	1.81	1481.32	1061.63
	delay 99%:	1054.62	1.81	2242.53	1265.59
	delay 100%:	256.6	1.81	183418.88	755.67

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	56479	192145	54180	184429	95.93%	95.98%	2298	7716	N/A	N/A
	Data	60151	191951	0	0	0.0%	0.0%	60151	191951	N/A	194
Route	Net Data	85170	270039	24945	77888	29.29%	28.84%	60225	192150	0	0
	Net Control	4164	90312	33983	731279	816.11%	809.73%	N/A	N/A	N/A	N/A
	→ *peer2peer*	292	6807	390	9088	133.56%	133.51%	-97	-2281	N/A	N/A
	→ *broadcast*	3713	79825	33592	722190	904.71%	904.72%	N/A	N/A	N/A	N/A
MAC	Net Sum	89335	360351	58928	809167	65.96%	224.55%	N/A	-448816	19495	191076

Table 5.36.: Average Result Table: Run 00604

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.43 μJ/B
Throughput: 459.644 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1381.783	496.0	1557123.6	3020.886
	rcvDataEnergy	640.989	355.2	1077151.68	1967.028
Control:	sendCtrlEnergy	353.354	310.8	614.8	44.043
	rcvCtrlEnergy	82.964	72.0	348.48	28.805
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.576	310.8	614.8	4.928
	→ *rcvCtrlBCEnergy*	79.83	72.0	152.0	1.227
Miscellaneous:	batteryLevel (%):	1.263	0.0	19.335	3.139
	delay 80%:	607.23	1.73	346.38	442.1
	delay 90%:	762.95	1.73	565.77	613.36
	delay 95%:	888.47	1.73	874.88	804.83
	delay 98%:	1009.83	1.73	1492.99	1053.66
	delay 99%:	1076.62	1.73	2143.65	1242.57
	delay 100%:	259.15	1.73	255939.85	787.73

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	59579	203145	57454	195948	96.43%	96.46%	2123	7196	N/A	N/A
	Data	63470	202956	0	0	0.0%	0.0%	63470	202956	N/A	188
Route	Net Data	87293	277362	23766	74250	27.23%	26.77%	63527	203112	0	0
	Net Control	3956	85732	32214	692759	814.31%	808.05%	N/A	N/A	N/A	N/A
	→ *peer2peer*	278	6491	346	8065	124.46%	124.25%	-67	-1574	N/A	N/A
	→ *broadcast*	3539	76035	31868	684694	900.48%	900.5%	N/A	N/A	N/A	N/A
MAC	Net Sum	91250	363095	55981	767010	61.35%	211.24%	N/A	-403915	18625	183844

Table 5.37.: Average Result Table: Run 00606

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.5 μJ/B
Throughput: 454.324 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1395.783	496.0	1725356.4	3125.625
	rcvDataEnergy	647.804	355.2	1193550.72	2044.448
Control:	sendCtrlEnergy	353.44	310.8	584.4	44.198
	rcvCtrlEnergy	83.031	72.0	348.48	29.153
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.562	310.8	584.4	5.009
	→ *rcvCtrlBCEnergy*	79.826	72.0	144.0	1.236
Miscellaneous:	batteryLevel (%):	1.252	0.0	21.878	3.212
	delay 80%:	639.3	1.94	373.73	466.34
	delay 90%:	799.69	1.94	607.1	638.79
	delay 95%:	927.2	1.94	923.29	828.93
	delay 98%:	1052.59	1.94	1546.81	1087.36
	delay 99%:	1122.56	1.94	2210.15	1287.95
	delay 100%:	267.79	1.94	221875.34	724.48

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	58972	200958	56790	193671	96.3%	96.37%	2181	7287	N/A	N/A
	Data	62818	200753	0	0	0.0%	0.0%	62818	200753	N/A	205
Route	Net Data	86759	275881	23905	75014	27.55%	27.19%	62854	200866	0	0
	Net Control	3884	84212	31553	678694	812.38%	805.94%	N/A	N/A	N/A	N/A
	→ *peer2peer*	274	6397	346	8074	126.28%	126.22%	-71	-1677	N/A	N/A
	→ *broadcast*	3471	74573	31206	670619	899.05%	899.28%	N/A	N/A	N/A	N/A
MAC	Net Sum	90645	360094	55459	753708	61.18%	209.31%	N/A	-393614	18207	179351

Table 5.38.: Average Result Table: Run 00608

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.49 μJ/B
Throughput: 457.252 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1392.303	496.0	1725356.4	3075.505
	rcvDataEnergy	646.481	355.2	1193550.72	2009.559
Control:	sendCtrlEnergy	353.581	310.8	584.4	44.344
	rcvCtrlEnergy	82.967	72.0	348.48	28.859
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.58	310.8	584.4	4.954
	→ *rcvCtrlBCEnergy*	79.828	72.0	144.0	1.228
Miscellaneous:	batteryLevel (%):	1.268	0.0	19.558	3.224
	delay 80%:	629.08	1.94	373.74	455.88
	delay 90%:	785.2	1.94	607.1	623.08
	delay 95%:	908.73	1.94	923.29	806.09
	delay 98%:	1030.96	1.94	1546.81	1058.96
	delay 99%:	1100.69	1.94	2210.15	1263.88
	delay 100%:	269.37	1.94	339457.12	953.7

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	59348	202246	57156	194868	96.31%	96.35%	2192	7378	N/A	N/A
	Data	63216	202040	0	0	0.0%	0.0%	63216	202040	N/A	206
Route	Net Data	87083	277023	23825	74851	27.36%	27.02%	63258	202172	0	0
	Net Control	3925	85072	32001	688250	815.31%	809.02%	N/A	N/A	N/A	N/A
	→ *peer2peer*	281	6563	345	8055	122.78%	122.73%	-63	-1492	N/A	N/A
	→ *broadcast*	3507	75345	31655	680194	902.62%	902.77%	N/A	N/A	N/A	N/A
MAC	Net Sum	91009	362095	55827	763101	61.34%	210.75%	N/A	-401005	18293	180515

Table 5.39.: Average Result Table: Run 00610

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 998.8
Energy / UserData: 7.33 μJ/B
Throughput: 467.96 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1363.548	496.0	1774709.2	3097.39
	rcvDataEnergy	633.299	355.2	1227701.76	2024.133
Control:	sendCtrlEnergy	353.572	310.8	584.4	44.398
	rcvCtrlEnergy	82.929	72.0	348.48	28.724
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.565	310.8	584.4	4.94
	→ *rcvCtrlBCEnergy*	79.825	72.0	144.0	1.232
Miscellaneous:	batteryLevel (%):	1.267	0.0	22.8	3.175
	delay 80%:	592.54	1.94	336.88	430.73
	delay 90%:	738.3	1.94	526.77	584.58
	delay 95%:	850.54	1.94	809.72	746.5
	delay 98%:	962.29	1.94	1345.36	974.98
	delay 99%:	1025.76	1.94	2093.88	1159.96
	delay 100%:	250.05	1.94	424763.25	927.57

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	60441	206312	58419	199108	96.65%	96.51%	2021	7203	N/A	N/A
	Data	64394	206114	0	0	0.0%	0.0%	64394	206114	N/A	198
Route	Net Data	87493	278146	23047	71890	26.34%	25.85%	64445	206256	0	0
	Net Control	3986	86417	32627	701780	818.54%	812.09%	N/A	N/A	N/A	N/A
	→ *peer2peer*	284	6626	347	8104	122.18%	122.31%	-62	-1478	N/A	N/A
	→ *broadcast*	3565	76598	32279	693675	905.44%	905.6%	N/A	N/A	N/A	N/A
MAC	Net Sum	91481	364563	55675	773670	60.86%	212.22%	N/A	-409107	18789	185540

Table 5.40.: Average Result Table: Run 00616

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 6.73 μJ/B
Throughput: 450.432 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1242.739	496.0	1175394.8	1929.154
	rcvDataEnergy	577.896	355.2	813003.84	1176.377
Control:	sendCtrlEnergy	353.479	310.8	538.8	44.007
	rcvCtrlEnergy	81.994	72.0	348.48	23.859
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.688	310.8	538.8	4.394
	→ *rcvCtrlBCEnergy*	79.859	72.0	132.0	1.087
Miscellaneous:	batteryLevel (%):	1.253	0.0	22.433	2.963
	delay 80%:	675.67	1.71	403.77	512.84
	delay 90%:	860.01	1.71	663.1	720.66
	delay 95%:	1012.0	1.71	1028.16	959.26
	delay 98%:	1167.75	1.71	1787.15	1300.34
	delay 99%:	1258.65	1.71	2836.21	1581.27
	delay 100%:	306.54	1.71	334492.38	889.65

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	58436	199182	56303	192105	96.35%	96.45%	2133	7076	N/A	N/A
	Data	62269	199026	0	0	0.0%	0.0%	62269	199026	N/A	155
Route	Net Data	77405	246363	15114	47280	19.53%	19.19%	62290	199083	0	0
	Net Control	3847	83218	40123	861539	1042.97%	1035.28%	N/A	N/A	N/A	N/A
	→ *peer2peer*	273	6368	293	6841	107.33%	107.43%	-20	-473	N/A	N/A
	→ *broadcast*	3477	74604	39829	854698	1145.5%	1145.65%	N/A	N/A	N/A	N/A
MAC	Net Sum	81253	329582	55238	908819	67.98%	275.75%	N/A	-579237	27298	274041

Table 5.41.: Average Result Table: Run 00620

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 926.4
Energy / UserData: 6.28 μJ/B
Throughput: 331.3 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1163.435	496.0	486470.0	1243.702
	rcvDataEnergy	543.586	355.2	336283.2	736.8
Control:	sendCtrlEnergy	347.95	310.8	630.0	33.912
	rcvCtrlEnergy	80.52	72.0	348.48	13.82
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.673	310.8	630.0	4.344
	→ *rcvCtrlBCEnergy*	79.812	72.0	156.0	1.225
Miscellaneous:	batteryLevel (%):	0.317	0.0	5.608	0.763
	delay 80%:	2156.66	1.93	1654.86	1997.88
	delay 90%:	3003.25	1.93	2680.94	3084.53
	delay 95%:	3627.24	1.93	3714.37	4017.2
	delay 98%:	4144.96	1.93	5057.66	4923.41
	delay 99%:	4371.62	1.93	6138.62	5390.25
	delay 100%:	948.7	1.93	256214.38	1432.65

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	43830	148229	41412	139925	94.48%	94.4%	2418	8304	N/A	N/A
	Data	46690	148122	0	0	0.0%	0.0%	46690	148122	N/A	106
Route	Net Data	54152	171704	7467	23613	13.79%	13.75%	46683	148091	0	0
	Net Control	2341	50474	44767	962799	1912.3%	1907.51%	N/A	N/A	N/A	N/A
	→ *peer2peer*	95	2221	108	2541	113.68%	114.41%	-13	-319	N/A	N/A
	→ *broadcast*	2213	47512	44657	960258	2017.94%	2021.09%	N/A	N/A	N/A	N/A
MAC	Net Sum	56493	222178	52235	986413	92.46%	443.97%	N/A	-764234	51331	524498

Table 5.42.: Average Result Table: Run 00624

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.57 μJ/B
Throughput: 327.396 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1364.117	496.0	1875902.4	2492.578
	rcvDataEnergy	632.654	355.2	1297739.52	1507.475
Control:	sendCtrlEnergy	352.043	310.8	645.2	41.68
	rcvCtrlEnergy	82.716	72.0	348.48	27.536
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.683	310.8	645.2	4.789
	→ *rcvCtrlBCEnergy*	79.866	72.0	160.0	1.112
Miscellaneous:	batteryLevel (%):	0.994	0.0	13.093	2.251
	delay 80%:	662.32	1.79	507.07	510.03
	delay 90%:	869.94	1.79	865.48	770.84
	delay 95%:	1045.46	1.79	1353.57	1065.52
	delay 98%:	1223.88	1.79	2276.16	1464.87
	delay 99%:	1323.93	1.79	3299.65	1767.84
	delay 100%:	319.71	1.79	204640.13	914.15

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	42808	145480	40923	138960	95.6%	95.52%	1884	6519	N/A	N/A
	Data	45601	145376	0	0	0.0%	0.0%	45601	145376	N/A	103
Route	Net Data	62042	196416	16385	50892	26.41%	25.91%	45656	145524	0	0
	Net Control	4036	87322	33318	715379	825.52%	819.24%	N/A	N/A	N/A	N/A
	→ *peer2peer*	252	5884	325	7572	128.97%	128.69%	-72	-1687	N/A	N/A
	→ *broadcast*	3647	78264	32993	707807	904.66%	904.38%	N/A	N/A	N/A	N/A
MAC	Net Sum	66079	283738	49704	766271	75.22%	270.06%	N/A	-482533	20976	208941

Table 5.43.: Average Result Table: Run 00626

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 7.41 μJ/B
Throughput: 530.466 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1398.374	496.0	2084392.8	3152.073
	rcvDataEnergy	649.3	355.2	1442010.24	2064.601
Control:	sendCtrlEnergy	352.197	310.8	523.6	42.372
	rcvCtrlEnergy	83.362	72.0	348.48	30.684
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.498	310.8	523.6	5.174
	→ *rcvCtrlBCEnergy*	79.809	72.0	128.0	1.309
Miscellaneous:	batteryLevel (%):	2.834	0.0	48.708	6.536
	delay 80%:	614.13	1.94	342.72	431.66
	delay 90%:	752.82	1.94	522.61	570.11
	delay 95%:	857.63	1.94	782.36	715.23
	delay 98%:	955.23	1.94	1265.57	898.38
	delay 99%:	1006.83	1.94	1859.42	1031.69
	delay 100%:	235.9	1.94	454279.4	843.34

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	68599	234127	66307	226421	96.66%	96.71%	2291	7705	N/A	N/A
	Data	73078	233851	0	0	0.0%	0.0%	73078	233851	N/A	276
Route	Net Data	101800	324484	28637	90381	28.13%	27.85%	73162	234102	0	0
	Net Control	3662	79408	27538	592794	751.99%	746.52%	N/A	N/A	N/A	N/A
	→ *peer2peer*	234	5468	336	7849	143.59%	143.54%	-101	-2380	N/A	N/A
	→ *broadcast*	3286	70672	27201	584945	827.78%	827.69%	N/A	N/A	N/A	N/A
MAC	Net Sum	105462	403892	56176	683176	53.27%	169.15%	N/A	-279284	15254	147538

Table 5.44.: Average Result Table: Run 00628

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 4.32 μJ/B
Throughput: 676.68 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	3186.802	496.0	1053050.4	4443.732
	rcvDataEnergy	1019.958	355.2	728329.92	2317.011
Control:	sendCtrlEnergy	353.452	310.8	569.2	43.779
	rcvCtrlEnergy	82.039	72.0	348.48	23.893
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.782	310.8	569.2	4.037
	→ *rcvCtrlBCEnergy*	79.895	72.0	140.0	0.956
Miscellaneous:	batteryLevel (%):	1.481	0.0	42.647	4.352
	delay 80%:	1180.49	1.69	772.73	935.35
	delay 90%:	1539.01	1.69	1286.79	1360.6
	delay 95%:	1817.14	1.69	2106.46	1784.24
	delay 98%:	2083.53	1.69	3794.83	2330.2
	delay 99%:	2233.56	1.69	5404.79	2762.14
	delay 100%:	542.1	1.69	187953.58	1450.06

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	89008	83867	84584	79662	95.03%	94.99%	4423	4205	N/A	N/A
	Data	90452	83699	0	0	0.0%	0.0%	90452	83699	N/A	168
Route	Net Data	115115	105982	24645	22248	21.41%	20.99%	90470	83733	0	0
	Net Control	3345	72272	31432	673854	939.67%	932.39%	N/A	N/A	N/A	N/A
	→ *peer2peer*	235	5499	229	5345	97.45%	97.2%	6	154	N/A	N/A
	→ *broadcast*	3033	65005	31202	668508	1028.75%	1028.39%	N/A	N/A	N/A	N/A
MAC	Net Sum	118461	178255	56077	696103	47.34%	390.51%	N/A	-517848	22780	217377

Table 5.45.: Average Result Table: Run 00702

Algorithm: AODV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 3.7 μJ/B
Throughput: 786.042 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	5452.214	496.0	761939.2	6435.812
	rcvDataEnergy	1508.082	355.2	526901.76	2463.237
Control:	sendCtrlEnergy	354.091	310.8	584.4	44.337
	rcvCtrlEnergy	81.626	72.0	348.48	21.278
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	340.935	310.8	584.4	3.232
	→ *rcvCtrlBCEnergy*	79.93	72.0	144.0	0.777
Miscellaneous:	batteryLevel (%):	1.91	0.0	42.118	4.48
	delay 80%:	1734.38	1.94	986.66	1276.36
	delay 90%:	2169.38	1.94	1453.89	1733.7
	delay 95%:	2469.07	1.94	2168.15	2120.97
	delay 98%:	2745.4	1.94	3874.14	2626.89
	delay 99%:	2908.95	1.94	6787.92	3105.52
	delay 100%:	707.31	1.94	463186.66	2119.72

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	104108	49931	98255	47013	94.38%	94.16%	5852	2917	N/A	N/A
	Data	104839	49784	0	0	0.0%	0.0%	104839	49784	N/A	146
Route	Net Data	127188	60276	22387	10494	17.6%	17.41%	104800	49782	0	0
	Net Control	3724	80312	35437	758616	951.58%	944.59%	N/A	N/A	N/A	N/A
	→ *peer2peer*	274	6402	206	4807	75.18%	75.09%	67	1594	N/A	N/A
	→ *broadcast*	3393	72619	35231	753808	1038.34%	1038.03%	N/A	N/A	N/A	N/A
MAC	Net Sum	130913	140589	57825	769111	44.17%	547.06%	N/A	-628521	26405	249875

Table 5.46.: Average Result Table: Run 00802

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 6.22 μJ/B
Throughput: 489.114 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1204.471	496.0	10737.6	823.398
	rcvDataEnergy	561.01	355.2	5683.2	271.004
Control:	sendCtrlEnergy	923.836	310.8	1428.0	384.224
	rcvCtrlEnergy	248.581	72.0	366.0	96.988
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	923.836	310.8	1428.0	384.224
	→ *rcvCtrlBCEnergy*	248.581	72.0	366.0	96.988
Miscellaneous:	batteryLevel (%):	2.588	0.0	53.689	6.341
	delay 80%:	589.47	1.94	561.01	620.7
	delay 90%:	880.04	1.94	1053.76	1026.72
	delay 95%:	1118.79	1.94	1646.3	1433.11
	delay 98%:	1351.24	1.94	2649.57	1938.46
	delay 99%:	1476.73	1.94	4025.3	2300.32
	delay 100%:	372.55	1.94	495622.33	1791.15

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	62154	213816	61138	209806	98.37%	98.12%	1015	4009	N/A	N/A
	Data	66322	213790	0	0	0.0%	0.0%	66322	213790	N/A	25
Route	Net Data	80221	257272	13898	43482	17.32%	16.9%	66322	213790	0	0
	Net Control	4987	14185	63598	165454	1275.28%	1166.4%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	4987	14185	63598	165454	1275.28%	1166.4%	N/A	N/A	N/A	N/A
MAC	Net Sum	85209	271457	77497	208937	90.95%	76.97%	N/A	62520	18005	41188

Table 5.47.: Average Result Table: Run 00901

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 989.0
Energy / UserData: 5.98 μJ/B
Throughput: 458.9875 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1141.68	496.0	10509.6	720.436
	rcvDataEnergy	532.714	355.2	3991.68	224.269
Control:	sendCtrlEnergy	899.496	310.8	1428.0	377.243
	rcvCtrlEnergy	238.25	72.0	366.0	96.715
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	899.496	310.8	1428.0	377.243
	→ *rcvCtrlBCEnergy*	238.25	72.0	366.0	96.715
Miscellaneous:	batteryLevel (%):	2.527	0.0	54.315	6.616
	delay 80%:	460.76	1.94	435.27	419.07
	delay 90%:	629.68	1.94	763.56	629.07
	delay 95%:	766.87	1.94	1215.34	850.43
	delay 98%:	904.79	1.94	2077.66	1150.78
	delay 99%:	981.08	1.94	3033.09	1374.98
	delay 100%:	318.32	1.94	587183.17	2313.73

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	57870	198822	56751	194731	98.07%	97.94%	1119	4091	N/A	N/A
	Data	61747	198802	0	0	0.0%	0.0%	61747	198802	N/A	20
Route	Net Data	71085	228482	9337	29681	13.13%	12.99%	61747	198801	0	0
	Net Control	5972	17663	76486	211220	1280.74%	1195.83%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	5972	17663	76486	211220	1280.74%	1195.83%	N/A	N/A	N/A	N/A
MAC	Net Sum	77058	246146	85824	240901	111.38%	97.87%	N/A	5244	19209	45814

Table 5.48.: Average Result Table: Run 00902

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 6.0 μJ/B
Throughput: 456.9225 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1132.66	496.0	12016.4	713.893
	rcvDataEnergy	528.526	355.2	4561.92	222.001
Control:	sendCtrlEnergy	918.802	310.8	1428.0	374.683
	rcvCtrlEnergy	239.812	72.0	366.0	96.255
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	918.802	310.8	1428.0	374.683
	→ *rcvCtrlBCEnergy*	239.812	72.0	366.0	96.255
Miscellaneous:	batteryLevel (%):	2.363	0.0	38.869	5.236
	delay 80%:	447.49	1.94	364.81	379.71
	delay 90%:	589.9	1.94	640.14	546.25
	delay 95%:	707.28	1.94	1019.5	733.21
	delay 98%:	820.72	1.94	1552.61	969.73
	delay 99%:	881.15	1.94	2258.98	1137.92
	delay 100%:	296.1	1.94	720034.49	3089.63

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	58293	200183	57114	196040	97.98%	97.93%	1178	4143	N/A	N/A
	Data	62193	200159	0	0	0.0%	0.0%	62193	200159	N/A	24
Route	Net Data	71041	228505	8847	28345	12.45%	12.4%	62193	200159	0	0
	Net Control	7482	21655	86767	240234	1159.68%	1109.37%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	7482	21655	86767	240234	1159.68%	1109.37%	N/A	N/A	N/A	N/A
MAC	Net Sum	78524	250160	95615	268580	121.77%	107.36%	N/A	-18419	19492	46029

Table 5.49.: Average Result Table: Run 00903

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.96 μJ/B
Throughput: 480.8475 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1121.372	496.0	13523.2	712.063
	rcvDataEnergy	522.667	355.2	5132.16	220.838
Control:	sendCtrlEnergy	854.875	310.8	1428.0	360.606
	rcvCtrlEnergy	218.421	72.0	366.0	93.465
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	854.875	310.8	1428.0	360.606
	→ *rcvCtrlBCEnergy*	218.421	72.0	366.0	93.465
Miscellaneous:	batteryLevel (%):	3.354	0.0	37.658	6.108
	delay 80%:	391.07	1.94	307.51	316.47
	delay 90%:	507.1	1.94	498.19	447.76
	delay 95%:	597.24	1.94	756.87	583.3
	delay 98%:	685.11	1.94	1262.26	762.94
	delay 99%:	732.84	1.94	1842.58	896.42
	delay 100%:	274.62	1.94	719776.26	4280.16

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	61438	210806	60106	206505	97.83%	97.96%	1332	4301	N/A	N/A
	Data	65546	210784	0	0	0.0%	0.0%	65546	210784	N/A	22
Route	Net Data	74175	237900	8628	27116	11.63%	11.4%	65546	210784	0	0
	Net Control	7827	25067	80805	253000	1032.39%	1009.3%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	7827	25067	80805	253000	1032.39%	1009.3%	N/A	N/A	N/A	N/A
MAC	Net Sum	82003	262967	89434	280116	109.06%	106.52%	N/A	-17148	17012	43779

Table 5.50.: Average Result Table: Run 00904

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.75 μJ/B
Throughput: 507.474 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1092.609	496.0	13523.2	660.719
	rcvDataEnergy	510.737	355.2	5132.16	198.07
Control:	sendCtrlEnergy	820.66	310.8	1428.0	351.919
	rcvCtrlEnergy	209.405	72.0	366.0	90.926
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	820.66	310.8	1428.0	351.919
	→ *rcvCtrlBCEnergy*	209.405	72.0	366.0	90.926
Miscellaneous:	batteryLevel (%):	3.134	0.0	33.627	5.474
	delay 80%:	511.15	1.94	304.55	396.31
	delay 90%:	650.32	1.94	489.14	547.96
	delay 95%:	760.55	1.94	775.62	713.8
	delay 98%:	866.21	1.94	1305.36	926.55
	delay 99%:	923.03	1.94	1895.72	1082.47
	delay 100%:	252.72	1.94	730514.37	3178.04

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	64584	221974	63434	218076	98.22%	98.24%	1149	3898	N/A	N/A
	Data	68908	221947	0	0	0.0%	0.0%	68908	221947	N/A	27
Route	Net Data	76031	245034	7122	23087	9.37%	9.42%	68908	221947	0	0
	Net Control	6879	23414	69939	232289	1016.7%	992.09%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	6879	23414	69939	232289	1016.7%	992.09%	N/A	N/A	N/A	N/A
MAC	Net Sum	82911	268448	77062	255376	92.95%	95.13%	N/A	13072	15181	41918

Table 5.51.: Average Result Table: Run 00906

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.74 μJ/B
Throughput: 517.955 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1091.466	496.0	10509.6	661.861
	rcvDataEnergy	509.925	355.2	3991.68	197.094
Control:	sendCtrlEnergy	820.608	310.8	1428.0	354.815
	rcvCtrlEnergy	209.894	72.0	366.0	92.146
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	820.608	310.8	1428.0	354.815
	→ *rcvCtrlBCEnergy*	209.894	72.0	366.0	92.146
Miscellaneous:	batteryLevel (%):	2.61	0.0	30.986	5.34
	delay 80%:	418.92	1.94	316.43	309.07
	delay 90%:	518.89	1.94	443.02	408.55
	delay 95%:	593.38	1.94	629.5	510.89
	delay 98%:	664.23	1.94	992.42	645.4
	delay 99%:	702.65	1.94	1603.97	748.32
	delay 100%:	246.97	1.94	677192.22	3693.36

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	65787	226319	64743	222590	98.41%	98.35%	1043	3728	N/A	N/A
	Data	70198	226296	0	0	0.0%	0.0%	70198	226296	N/A	23
Route	Net Data	77452	248853	7252	22557	9.36%	9.06%	70198	226295	0	0
	Net Control	6917	23538	71282	236032	1030.53%	1002.77%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	6917	23538	71282	236032	1030.53%	1002.77%	N/A	N/A	N/A	N/A
MAC	Net Sum	84369	272391	78535	258589	93.09%	94.93%	N/A	13802	14406	39974

Table 5.52.: Average Result Table: Run 00908

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.74 μJ/B
Throughput: 521.0625 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1087.55	496.0	13523.2	659.028
	rcvDataEnergy	508.188	355.2	4561.92	196.212
Control:	sendCtrlEnergy	834.016	310.8	1428.0	346.081
	rcvCtrlEnergy	214.264	72.0	366.0	89.079
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	834.016	310.8	1428.0	346.081
	→ *rcvCtrlBCEnergy*	214.264	72.0	366.0	89.079
Miscellaneous:	batteryLevel (%):	2.793	0.0	22.992	5.246
	delay 80%:	421.51	1.94	297.0	311.31
	delay 90%:	521.41	1.94	413.83	409.61
	delay 95%:	597.74	1.94	642.79	516.77
	delay 98%:	672.67	1.94	1039.38	664.99
	delay 99%:	713.47	1.94	1544.96	777.04
	delay 100%:	243.99	1.94	727020.2	3357.92

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	66138	227591	65132	223941	98.48%	98.4%	1006	3650	N/A	N/A
	Data	70574	227568	0	0	0.0%	0.0%	70574	227568	N/A	23
Route	Net Data	77522	249441	6947	21874	8.96%	8.77%	70573	227566	0	0
	Net Control	7368	24244	77050	244498	1045.74%	1008.49%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	7368	24244	77050	244498	1045.74%	1008.49%	N/A	N/A	N/A	N/A
MAC	Net Sum	84890	273685	83999	266372	98.95%	97.33%	N/A	7313	15728	41558

Table 5.53.: Average Result Table: Run 00910

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.79 μJ/B
Throughput: 526.864 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1102.743	496.0	10509.6	683.651
	rcvDataEnergy	515.223	355.2	3991.68	208.476
Control:	sendCtrlEnergy	822.403	310.8	1428.0	350.828
	rcvCtrlEnergy	209.824	72.0	366.0	90.389
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	822.403	310.8	1428.0	350.828
	→ *rcvCtrlBCEnergy*	209.824	72.0	366.0	90.389
Miscellaneous:	batteryLevel (%):	2.507	0.0	22.591	4.781
	delay 80%:	535.38	1.94	303.88	395.77
	delay 90%:	661.43	1.94	441.27	518.77
	delay 95%:	758.17	1.94	683.07	654.67
	delay 98%:	852.72	1.94	1038.5	840.77
	delay 99%:	903.65	1.94	1506.61	978.94
	delay 100%:	246.76	1.94	755810.0	3323.07

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	66934	230240	65857	226285	98.39%	98.28%	1076	3955	N/A	N/A
	Data	71421	230216	0	0	0.0%	0.0%	71421	230216	N/A	24
Route	Net Data	79511	255825	8090	25610	10.17%	10.01%	71421	230215	0	0
	Net Control	7008	23646	71580	235517	1021.4%	996.01%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	7008	23646	71580	235517	1021.4%	996.01%	N/A	N/A	N/A	N/A
MAC	Net Sum	86520	279472	79671	261127	92.08%	93.44%	N/A	18344	15072	41356

Table 5.54.: Average Result Table: Run 00916

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.65 μJ/B
Throughput: 488.56 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1072.516	496.0	12016.4	634.951
	rcvDataEnergy	501.124	355.2	4561.92	184.731
Control:	sendCtrlEnergy	937.862	310.8	1428.0	380.761
	rcvCtrlEnergy	247.487	72.0	366.0	97.201
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	937.862	310.8	1428.0	380.761
	→ *rcvCtrlBCEnergy*	247.487	72.0	366.0	97.201
Miscellaneous:	batteryLevel (%):	1.606	0.0	25.471	3.854
	delay 80%:	373.7	1.94	375.7	288.89
	delay 90%:	474.37	1.94	618.0	398.66
	delay 95%:	557.7	1.94	989.62	528.72
	delay 98%:	646.06	1.94	1917.88	727.98
	delay 99%:	699.97	1.94	3307.28	904.53
	delay 100%:	331.43	1.94	503830.97	2747.43

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	62215	213793	61069	209768	98.16%	98.12%	1145	4025	N/A	N/A
	Data	66379	213764	0	0	0.0%	0.0%	66379	213764	N/A	28
Route	Net Data	72099	231116	5719	17352	7.93%	7.51%	66379	213764	0	0
	Net Control	6970	19725	93497	250087	1341.42%	1267.87%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	6970	19725	93497	250087	1341.42%	1267.87%	N/A	N/A	N/A	N/A
MAC	Net Sum	79069	250842	99218	267440	125.48%	106.62%	N/A	-16597	24951	57340

Table 5.55.: Average Result Table: Run 00920

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 999.4
Energy / UserData: 5.49 μJ/B
Throughput: 366.368 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1076.856	496.0	5989.2	627.437
	rcvDataEnergy	503.259	355.2	2280.96	181.647
Control:	sendCtrlEnergy	930.885	310.8	1428.0	440.779
	rcvCtrlEnergy	268.442	72.0	366.0	106.094
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	930.885	310.8	1428.0	440.779
	→ *rcvCtrlBCEnergy*	268.442	72.0	366.0	106.094
Miscellaneous:	batteryLevel (%):	0.385	0.0	6.348	0.823
	delay 80%:	2606.72	1.94	2374.51	2705.03
	delay 90%:	3881.1	1.94	3957.04	4479.59
	delay 95%:	4828.95	1.94	5560.43	5955.19
	delay 98%:	5620.4	1.94	7837.42	7383.62
	delay 99%:	5974.96	1.94	9665.98	8146.15
	delay 100%:	1320.44	1.94	401271.86	2612.57

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	46592	160210	45755	157080	98.2%	98.05%	836	3129	N/A	N/A
	Data	49708	160178	0	0	0.0%	0.0%	49708	160178	N/A	32
Route	Net Data	54090	174434	4382	14256	8.1%	8.17%	49708	160178	0	0
	Net Control	1917	5477	48364	116516	2522.9%	2127.37%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	1917	5477	48364	116516	2522.9%	2127.37%	N/A	N/A	N/A	N/A
MAC	Net Sum	56009	179911	52746	130773	94.17%	72.69%	N/A	49138	23424	47792

Table 5.56.: Average Result Table: Run 00924

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 6.12 μJ/B
Throughput: 364.508 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1125.263	496.0	12016.4	714.809
	rcvDataEnergy	524.593	355.2	4561.92	222.234
Control:	sendCtrlEnergy	1032.821	310.8	1428.0	360.036
	rcvCtrlEnergy	267.0	72.0	366.0	92.051
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	1032.821	310.8	1428.0	360.036
	→ *rcvCtrlBCEnergy*	267.0	72.0	366.0	92.051
Miscellaneous:	batteryLevel (%):	1.214	0.0	12.152	2.542
	delay 80%:	607.83	1.94	437.97	486.46
	delay 90%:	802.86	1.94	767.12	728.76
	delay 95%:	965.23	1.94	1219.17	996.3
	delay 98%:	1126.17	1.94	2020.32	1345.74
	delay 99%:	1216.93	1.94	3158.76	1618.25
	delay 100%:	347.89	1.94	609966.69	4001.09

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	46654	160063	45563	156083	97.66%	97.51%	1090	3980	N/A	N/A
	Data	49770	160039	0	0	0.0%	0.0%	49770	160039	N/A	24
Route	Net Data	56541	181106	6770	21067	11.97%	11.63%	49770	160039	0	0
	Net Control	8875	21990	89225	215450	1005.35%	979.76%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	8875	21990	89225	215450	1005.35%	979.76%	N/A	N/A	N/A	N/A
MAC	Net Sum	65417	203097	95996	236518	146.74%	116.46%	N/A	-33420	21188	46290

Table 5.57.: Average Result Table: Run 00926

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 5.62 μJ/B
Throughput: 538.38667 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1072.719	496.0	12016.4	637.863
	rcvDataEnergy	502.406	355.2	3991.68	188.451
Control:	sendCtrlEnergy	691.73	310.8	1428.0	305.465
	rcvCtrlEnergy	173.963	72.0	366.0	79.137
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	691.73	310.8	1428.0	305.465
	→ *rcvCtrlBCEnergy*	173.963	72.0	366.0	79.137
Miscellaneous:	batteryLevel (%):	9.256	0.0	57.807	11.732
	delay 80%:	297.29	1.94	287.24	207.8
	delay 90%:	365.15	1.94	405.14	275.73
	delay 95%:	411.92	1.94	604.37	336.75
	delay 98%:	456.19	1.94	888.6	417.88
	delay 99%:	479.56	1.94	1241.17	476.9
	delay 100%:	220.03	1.94	796864.72	3908.26

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	68135	234788	67298	231665	98.77%	98.67%	837	3123	N/A	N/A
	Data	72719	234778	0	0	0.0%	0.0%	72719	234778	N/A	10
Route	Net Data	78969	255164	6249	20386	7.91%	7.99%	72718	234777	0	0
	Net Control	6026	26576	59565	259125	988.47%	975.03%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	6026	26576	59565	259125	988.47%	975.03%	N/A	N/A	N/A	N/A
MAC	Net Sum	84996	281740	65815	279512	77.43%	99.21%	N/A	2228	10329	36097

Table 5.58.: Average Result Table: Run 00928

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 3.46 μJ/B
Throughput: 777.228 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	2663.379	496.0	30938.4	2309.442
	rcvDataEnergy	852.696	355.2	8507.52	534.324
Control:	sendCtrlEnergy	776.398	310.8	1428.0	360.186
	rcvCtrlEnergy	202.424	72.0	366.0	94.046
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	776.398	310.8	1428.0	360.186
	→ *rcvCtrlBCEnergy*	202.424	72.0	366.0	94.046
Miscellaneous:	batteryLevel (%):	1.876	0.0	44.998	5.111
	delay 80%:	1086.23	1.94	778.81	894.36
	delay 90%:	1415.8	1.94	1388.29	1270.21
	delay 95%:	1668.1	1.94	2187.8	1649.0
	delay 98%:	1920.63	1.94	3988.98	2177.0
	delay 99%:	2061.96	1.94	5688.13	2587.01
	delay 100%:	538.7	1.94	586617.63	3659.09

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	99707	95086	97152	92412	97.44%	97.19%	2554	2674	N/A	N/A
	Data	101532	95064	0	0	0.0%	0.0%	101532	95064	N/A	22
Route	Net Data	109709	102977	8176	7913	7.45%	7.68%	101532	95064	0	0
	Net Control	3962	14647	47914	167521	1209.34%	1143.72%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	3962	14647	47914	167521	1209.34%	1143.72%	N/A	N/A	N/A	N/A
MAC	Net Sum	113671	117624	56091	175435	49.35%	149.15%	N/A	-57810	14696	37922

Table 5.59.: Average Result Table: Run 01002

Algorithm: DSDV, simulation time: 1000s , 50 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 3.07 μJ/B
Throughput: 861.47 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	4774.266	496.0	49860.4	4518.872
	rcvDataEnergy	1316.568	355.2	12453.12	1015.702
Control:	sendCtrlEnergy	758.839	310.8	1428.0	357.05
	rcvCtrlEnergy	198.854	72.0	366.0	93.63
	→ *sendCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlP2PEnergy*	NaN	NaN	NaN	NaN
	→ *sendCtrlBCEnergy*	758.839	310.8	1428.0	357.05
	→ *rcvCtrlBCEnergy*	198.854	72.0	366.0	93.63
Miscellaneous:	batteryLevel (%):	2.695	0.0	51.66	6.262
	delay 80%:	1018.8	1.94	1022.43	811.34
	delay 90%:	1311.5	1.94	1722.04	1138.94
	delay 95%:	1530.42	1.94	2733.1	1458.78
	delay 98%:	1754.04	1.94	5115.96	1929.94
	delay 99%:	1884.85	1.94	6886.95	2322.37
	delay 100%:	816.11	1.94	542584.48	4801.63

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	112098	54215	107683	51983	96.06%	95.88%	4414	2231	N/A	N/A
	Data	113109	54200	0	0	0.0%	0.0%	113109	54200	N/A	15
Route	Net Data	121361	58374	8254	4175	6.8%	7.15%	113107	54199	0	0
	Net Control	3707	14101	44423	158517	1198.35%	1124.15%	N/A	N/A	N/A	N/A
	→ *peer2peer*	0	0	0	0	NaN%	NaN%	0	0	N/A	N/A
	→ *broadcast*	3707	14101	44423	158517	1198.35%	1124.15%	N/A	N/A	N/A	N/A
MAC	Net Sum	125069	72476	52677	162692	42.12%	224.48%	N/A	-90216	16181	38033

Table 5.60.: Average Result Table: Run 01102

UDP

Algorithm: Beehive, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 19.28 μJ/B
Throughput: 271.896 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1463.851	1392.8	15022.4	504.876
	rcvDataEnergy	614.668	545.04	5692.32	309.705
Control:	sendCtrlEnergy	326.23	310.8	556.8	30.366
	rcvCtrlEnergy	75.428	72.0	360.24	15.787
	→ *sendCtrlP2PEnergy*	518.621	488.4	556.8	26.467
	→ *rcvCtrlP2PEnergy*	351.385	345.12	360.24	5.775
	→ *sendCtrlBCEnergy*	322.586	310.8	379.2	15.428
	→ *rcvCtrlBCEnergy*	74.531	72.0	90.0	3.921
Miscellaneous:	batteryLevel (%):	11.831	0.0	50.88	10.577
	delay 80%:	309.62	5.47	1429.41	429.97
	delay 90%:	900.99	5.47	3780.71	1972.61
	delay 95%:	1741.49	5.47	6482.47	4241.49
	delay 98%:	2666.5	5.47	10328.33	6774.75
	delay 99%:	3131.96	5.47	13868.37	8188.28
	delay 100%:	807.83	5.47	284550.06	2614.28

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	140237	280474	33986	67973	24.23%	24.24%	106250	212500	N/A	N/A
	Data	44916	84813	35876	67973	79.87%	80.14%	9039	16839	N/A	195661
Route	Net Data	52175	98202	43135	81362	82.67%	82.85%	9039	16839	0	0
	Net Control	23276	614985	97901	2677828	420.61%	435.43%	N/A	N/A	N/A	N/A
	→ *peer2peer*	527	10725	390	8154	74.0%	76.03%	137	2571	N/A	N/A
	→ *broadcast*	22748	604259	97511	2669674	428.66%	441.81%	N/A	N/A	N/A	N/A
MAC	Net Sum	75452	713187	141037	2759191	186.92%	386.88%	N/A	-2046004	60562	612531

Table 5.61.: Average Result Table: Run 00051

Algorithm: Beehive, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 16.82 μJ/B
Throughput: 301.914 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1447.052	1392.8	15022.4	393.243
	rcvDataEnergy	589.25	545.04	5692.32	218.74
Control:	sendCtrlEnergy	327.577	310.8	556.8	31.891
	rcvCtrlEnergy	75.724	72.0	360.24	15.487
	→ *sendCtrlP2PEnergy*	515.555	488.4	556.8	27.775
	→ *rcvCtrlP2PEnergy*	350.377	345.12	360.24	6.127
	→ *sendCtrlBCEnergy*	323.813	310.8	379.2	17.617
	→ *rcvCtrlBCEnergy*	74.923	72.0	90.0	4.59
Miscellaneous:	batteryLevel (%):	8.497	0.0	39.594	8.628
	delay 80%:	182.28	5.47	378.63	135.4
	delay 90%:	627.58	5.47	2918.76	1587.16
	delay 95%:	1413.85	5.47	4803.72	3759.03
	delay 98%:	2156.11	5.47	8394.88	5624.0
	delay 99%:	2533.28	5.47	11974.17	6746.32
	delay 100%:	650.95	5.47	284439.68	2184.04

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	146744	293489	37739	75478	25.72%	25.72%	109005	218010	N/A	N/A
	Data	50195	94794	39822	75478	79.33%	79.62%	10372	19316	N/A	198694
Route	Net Data	56025	105583	45652	86267	81.49%	81.71%	10372	19316	0	0
	Net Control	20358	532360	91137	2456588	447.67%	461.45%	N/A	N/A	N/A	N/A
	→ *peer2peer*	525	10533	347	7259	66.1%	68.92%	177	3273	N/A	N/A
	→ *broadcast*	19833	521827	90788	2449329	457.76%	469.38%	N/A	N/A	N/A	N/A
MAC	Net Sum	76385	637943	136790	2542856	179.08%	398.6%	N/A	-1904912	59647	576325

Table 5.62.: Average Result Table: Run 00052

Algorithm: Beehive, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 17.87 μJ/B
Throughput: 292.484 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1447.463	1392.8	11940.4	379.676
	rcvDataEnergy	587.912	545.04	3991.68	207.078
Control:	sendCtrlEnergy	330.274	310.8	556.8	34.89
	rcvCtrlEnergy	76.652	72.0	360.24	16.257
	→ *sendCtrlP2PEnergy*	521.451	488.4	556.8	27.886
	→ *rcvCtrlP2PEnergy*	351.753	345.12	360.24	6.338
	→ *sendCtrlBCEnergy*	325.821	310.8	379.2	19.133
	→ *rcvCtrlBCEnergy*	75.781	72.0	90.0	5.195
Miscellaneous:	batteryLevel (%):	11.466	0.0	40.495	7.593
	delay 80%:	171.25	5.47	250.47	124.63
	delay 90%:	673.07	5.47	2499.61	1769.02
	delay 95%:	1451.73	5.47	4151.23	3774.7
	delay 98%:	2096.42	5.47	5445.33	5203.51
	delay 99%:	2376.4	5.47	8074.98	5882.7
	delay 100%:	599.04	5.47	143260.64	1845.89

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	149119	298239	36559	73120	24.52%	24.52%	112559	225118	N/A	N/A
	Data	52277	98620	38578	73120	73.8%	74.14%	13699	25499	N/A	199619
Route	Net Data	58252	109684	44552	84184	76.48%	76.75%	13699	25499	0	0
	Net Control	22452	571231	99735	2575255	444.21%	450.83%	N/A	N/A	N/A	N/A
	→ *peer2peer*	685	12982	419	8185	61.17%	63.05%	266	4797	N/A	N/A
	→ *broadcast*	21766	558248	99316	2567070	456.29%	459.84%	N/A	N/A	N/A	N/A
MAC	Net Sum	80704	680915	144289	2659440	178.79%	390.57%	N/A	-1978524	69238	644593

Table 5.63.: Average Result Table: Run 00053

Algorithm: Beehive, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 17.88 μJ/B
Throughput: 282.11 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1425.36	1392.8	10600.8	275.254
	rcvDataEnergy	569.492	545.04	3458.4	143.277
Control:	sendCtrlEnergy	329.55	310.8	556.8	38.435
	rcvCtrlEnergy	76.277	72.0	360.24	17.936
	→ *sendCtrlP2PEnergy*	522.917	488.4	556.8	25.831
	→ *rcvCtrlP2PEnergy*	352.382	345.12	360.24	6.01
	→ *sendCtrlBCEnergy*	323.506	310.8	379.2	17.751
	→ *rcvCtrlBCEnergy*	75.177	72.0	90.0	4.791
Miscellaneous:	batteryLevel (%):	21.69	4.377	47.935	6.615
	delay 80%:	192.62	5.47	976.17	235.2
	delay 90%:	822.38	5.47	3165.13	2120.6
	delay 95%:	1620.39	5.47	4362.05	4016.54
	delay 98%:	2243.42	5.47	5482.35	5297.92
	delay 99%:	2491.52	5.47	8794.01	5820.56
	delay 100%:	599.15	5.47	796446.53	2148.52

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	149760	299520	35264	70528	23.55%	23.55%	114495	228991	N/A	N/A
	Data	47910	90470	37202	70528	77.65%	77.96%	10708	19942	N/A	209049
Route	Net Data	51490	97109	40781	77166	79.2%	79.46%	10708	19942	0	0
	Net Control	19879	518557	83737	2227453	421.23%	429.55%	N/A	N/A	N/A	N/A
	→ *peer2peer*	845	15753	463	8824	54.79%	56.01%	381	6928	N/A	N/A
	→ *broadcast*	19033	502804	83273	2218629	437.52%	441.25%	N/A	N/A	N/A	N/A
MAC	Net Sum	71370	615666	124519	2304620	174.47%	374.33%	N/A	-1688953	56430	537861

Table 5.64.: Average Result Table: Run 00054

Algorithm: DSR, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 41.08 μJ/B
Throughput: 141.496 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	2537.872	1392.8	89829.2	2434.953
	rcvDataEnergy	1094.179	545.04	30695.52	1099.33
Control:	sendCtrlEnergy	516.694	480.8	1332.0	31.319
	rcvCtrlEnergy	170.001	72.0	573.6	122.599
	→ *sendCtrlP2PEnergy*	516.694	480.8	2572.8	31.319
	→ *rcvCtrlP2PEnergy*	354.112	345.12	573.6	9.811
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	90.298	72.0	366.0	22.587
Miscellaneous:	batteryLevel (%):	2.685	0.0	28.295	5.647
	delay 80%:	6288.37	5.42	6957.61	7426.25
	delay 90%:	9487.98	5.42	11468.62	11604.48
	delay 95%:	11992.72	5.42	16963.72	15607.26
	delay 98%:	14292.95	5.42	25412.37	20189.68
	delay 99%:	15440.78	5.42	36243.33	23117.44
	delay 100%:	3544.73	5.42	165156.0	6806.25

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	114551	229103	17686	35374	15.44%	15.44%	96864	193729	N/A	N/A
	Data	122697	234450	18454	35374	15.04%	15.09%	104242	199076	N/A	-5347
Route	Net Data	218789	413850	90362	171217	41.3%	41.37%	128426	242632	0	0
	Net Control	3247	52895	6342	99587	195.32%	188.27%	N/A	N/A	N/A	N/A
	→ *peer2peer*	1542	31048	1676	29888	108.69%	96.26%	-133	1160	N/A	N/A
	→ *broadcast*	0	0	4666	69698	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	222037	466745	96705	270804	43.55%	58.02%	N/A	195941	77788	159508

Table 5.65.: Average Result Table: Run 00351

Algorithm: DSR, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 42.58 μJ/B
Throughput: 129.844 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	2513.442	1392.8	56927.6	2355.491
	rcvDataEnergy	1089.223	545.04	20576.64	1064.402
Control:	sendCtrlEnergy	521.429	480.8	1636.0	40.556
	rcvCtrlEnergy	182.069	72.0	630.72	127.388
	→ *sendCtrlP2PEnergy*	521.429	480.8	3044.0	40.556
	→ *rcvCtrlP2PEnergy*	357.065	345.12	630.72	15.444
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	91.919	72.0	404.0	24.111
Miscellaneous:	batteryLevel (%):	1.703	0.0	20.301	3.781
	delay 80%:	5444.37	5.4	6387.95	6692.78
	delay 90%:	8541.44	5.4	11049.87	10981.61
	delay 95%:	11163.71	5.4	18222.59	15591.87
	delay 98%:	14027.73	5.4	32683.54	22529.72
	delay 99%:	15694.95	5.4	51611.44	27994.44
	delay 100%:	3784.2	5.4	264742.85	9034.32

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	113639	227279	16230	32460	14.28%	14.28%	97409	194818	N/A	N/A
	Data	127848	244401	16932	32460	13.24%	13.28%	110915	211940	N/A	-17121
Route	Net Data	221002	418820	86100	163282	38.96%	38.99%	134901	255538	0	0
	Net Control	4533	66369	7503	110750	165.52%	166.87%	N/A	N/A	N/A	N/A
	→ *peer2peer*	1942	37231	2377	37666	122.4%	101.17%	-434	-435	N/A	N/A
	→ *broadcast*	0	0	5125	73083	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	225536	485189	93604	274032	41.5%	56.48%	N/A	211157	83489	173315

Table 5.66.: Average Result Table: Run 00352

Algorithm: DSR, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 44.04 μJ/B
Throughput: 128.228 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	2497.898	1392.8	75357.6	2379.08
	rcvDataEnergy	1115.698	545.04	24650.64	1091.72
Control:	sendCtrlEnergy	527.214	480.8	1598.0	52.794
	rcvCtrlEnergy	185.522	72.0	849.12	129.483
	→ *sendCtrlP2PEnergy*	527.214	480.8	2869.2	52.794
	→ *rcvCtrlP2PEnergy*	360.806	345.12	849.12	21.344
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	92.673	72.0	444.0	24.355
Miscellaneous:	batteryLevel (%):	1.351	0.0	20.881	3.183
	delay 80%:	4610.96	5.4	4898.38	5680.2
	delay 90%:	7389.83	5.4	9001.09	9690.02
	delay 95%:	9897.44	5.4	16364.76	14393.96
	delay 98%:	12768.38	5.4	31960.33	21853.58
	delay 99%:	14558.4	5.4	54234.4	28186.14
	delay 100%:	3661.17	5.4	404150.68	9740.97

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	115630	231261	16028	32056	13.86%	13.86%	99602	199204	N/A	N/A
	Data	127720	244188	16729	32056	13.1%	13.13%	110991	212131	N/A	-12926
Route	Net Data	221211	419180	85175	161520	38.5%	38.53%	136035	257660	0	0
	Net Control	7015	90607	10002	139262	142.58%	153.7%	N/A	N/A	N/A	N/A
	→ *peer2peer*	2519	45728	3432	47928	136.24%	104.81%	-913	-2200	N/A	N/A
	→ *broadcast*	0	0	6568	91333	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	228227	509787	95178	300782	41.7%	59.0%	N/A	209005	84009	179343

Table 5.67.: Average Result Table: Run 00353

Algorithm: DSR, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 47.69 μJ/B
Throughput: 123.548 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	2522.736	1392.8	56388.0	2450.927
	rcvDataEnergy	1150.655	545.04	20457.36	1139.281
Control:	sendCtrlEnergy	529.905	480.8	2206.0	60.461
	rcvCtrlEnergy	189.028	72.0	724.8	131.395
	→ *sendCtrlP2PEnergy*	529.905	480.8	3834.4	60.461
	→ *rcvCtrlP2PEnergy*	362.934	345.12	724.8	24.606
	→ *sendCtrlBCEnergy*	NaN	NaN	NaN	NaN
	→ *rcvCtrlBCEnergy*	93.296	72.0	366.0	25.084
Miscellaneous:	batteryLevel (%):	1.288	0.0	18.408	3.095
	delay 80%:	4613.05	5.41	4731.58	5752.27
	delay 90%:	7477.27	5.41	9192.98	9954.92
	delay 95%:	10003.64	5.41	16923.35	14603.55
	delay 98%:	13027.92	5.41	36187.24	22707.85
	delay 99%:	14870.93	5.41	52250.4	29181.96
	delay 100%:	3673.75	5.41	519848.8	9550.22

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	118143	236288	15443	30886	13.07%	13.07%	102700	205401	N/A	N/A
	Data	129715	247956	16126	30886	12.43%	12.46%	113588	217069	N/A	-11668
Route	Net Data	227431	430725	86752	164417	38.14%	38.17%	140678	266307	0	0
	Net Control	8687	106020	11492	154896	132.29%	146.1%	N/A	N/A	N/A	N/A
	→ *peer2peer*	2920	51694	4213	54964	144.28%	106.33%	-1292	-3270	N/A	N/A
	→ *broadcast*	0	0	7279	99931	NaN%	NaN%	N/A	N/A	N/A	N/A
MAC	Net Sum	236119	536745	98244	319313	41.61%	59.49%	N/A	217432	84881	183076

Table 5.68.: Average Result Table: Run 00354

Algorithm: AODV, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 20.47 μJ/B
Throughput: 228.382 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1995.527	1392.8	2463996.8	4058.574
	rcvDataEnergy	952.85	553.44	1704067.2	4232.073
Control:	sendCtrlEnergy	352.514	310.8	538.8	46.187
	rcvCtrlEnergy	84.901	72.0	348.48	38.228
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	338.947	310.8	538.8	9.529
	→ *rcvCtrlBCEnergy*	79.364	72.0	120.0	2.425
Miscellaneous:	batteryLevel (%):	4.373	0.0	40.663	8.547
	delay 80%:	5034.51	5.4	7307.38	7565.77
	delay 90%:	8806.46	5.4	12438.52	13022.36
	delay 95%:	11551.43	5.4	17065.39	17272.31
	delay 98%:	13860.82	5.4	24142.97	21476.23
	delay 99%:	14941.06	5.4	30278.46	23910.76
	delay 100%:	3307.6	5.4	86954.32	5786.16

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	119216	238432	28547	57094	23.95%	23.95%	90669	181338	N/A	N/A
	Data	83239	160222	0	0	0.0%	0.0%	83239	160222	N/A	78210
Route	Net Data	134775	259418	53914	103776	40.0%	40.0%	80860	155642	0	0
	Net Control	3941	86921	22850	501342	579.8%	576.78%	N/A	N/A	N/A	N/A
	→ *peer2peer*	290	6779	440	10258	151.72%	151.32%	-149	-3479	N/A	N/A
	→ *broadcast*	3451	75515	22409	491084	649.35%	650.31%	N/A	N/A	N/A	N/A
MAC	Net Sum	138717	346339	76765	605118	55.34%	174.72%	N/A	-258778	27510	136613

Table 5.69.: Average Result Table: Run 00651

Algorithm: AODV, simulation time: 1000s , 30 nodes
Average Network Lifetime: 1000.0
Energy / UserData: 20.68 μJ/B
Throughput: 220.248 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1969.896	1392.8	2771192.8	4122.65
	rcvDataEnergy	909.796	553.44	1916640.0	4320.194
Control:	sendCtrlEnergy	354.247	310.8	584.4	48.738
	rcvCtrlEnergy	85.133	72.0	348.48	39.011
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	338.9	310.8	584.4	9.694
	→ *rcvCtrlBCEnergy*	79.359	72.0	124.0	2.437
Miscellaneous:	batteryLevel (%):	3.317	0.0	37.429	6.639
	delay 80%:	5770.94	5.4	7169.72	7949.28
	delay 90%:	9311.68	5.4	11105.03	12679.58
	delay 95%:	11864.91	5.4	15819.06	16491.62
	delay 98%:	14008.98	5.4	22944.5	20289.88
	delay 99%:	14982.58	5.4	28733.91	22388.41
	delay 100%:	3292.57	5.4	102242.51	5408.68

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	118824	237650	27530	55061	23.17%	23.17%	91294	182588	N/A	N/A
	Data	86591	166673	0	0	0.0%	0.0%	86591	166673	N/A	70976
Route	Net Data	135671	261142	51558	99240	38.0%	38.0%	84112	161901	0	0
	Net Control	4118	90897	23540	516637	571.64%	568.38%	N/A	N/A	N/A	N/A
	→ *peer2peer*	345	8059	474	11060	137.39%	137.24%	-128	-3001	N/A	N/A
	→ *broadcast*	3576	78282	23064	505577	644.97%	645.84%	N/A	N/A	N/A	N/A
MAC	Net Sum	139790	352039	75099	615878	53.72%	174.95%	N/A	-263838	29694	142267

Table 5.70.: Average Result Table: Run 00652

Algorithm: AODV, simulation time: 1000s , 30 nodes
Average Network Lifetime: 999.8
Energy / UserData: 20.14 μJ/B
Throughput: 231.726 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1973.527	1392.8	2384931.6	3269.652
	rcvDataEnergy	906.791	553.44	1649355.84	3222.663
Control:	sendCtrlEnergy	354.634	310.8	508.4	49.349
	rcvCtrlEnergy	85.691	72.0	348.48	40.877
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	338.842	310.8	508.4	9.754
	→ *rcvCtrlBCEnergy*	79.345	72.0	124.0	2.461
Miscellaneous:	batteryLevel (%):	2.326	0.0	29.997	4.748
	delay 80%:	4480.97	5.4	5727.25	6117.2
	delay 90%:	7379.29	5.4	9499.45	10165.26
	delay 95%:	9519.25	5.4	13382.44	13492.17
	delay 98%:	11323.32	5.4	19193.31	16778.64
	delay 99%:	12148.79	5.4	23647.38	18598.05
	delay 100%:	2689.75	5.4	126394.48	4618.28

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	121179	242358	28959	57918	23.9%	23.9%	92220	184440	N/A	N/A
	Data	86000	165535	0	0	0.0%	0.0%	86000	165535	N/A	76823
Route	Net Data	136599	262930	52941	101902	38.76%	38.76%	83658	161027	0	0
	Net Control	4123	91067	22042	484086	534.61%	531.57%	N/A	N/A	N/A	N/A
	→ *peer2peer*	356	8304	489	11400	137.36%	137.28%	-132	-3095	N/A	N/A
	→ *broadcast*	3571	78217	21552	472686	603.53%	604.33%	N/A	N/A	N/A	N/A
MAC	Net Sum	140723	353997	74983	585989	53.28%	165.54%	N/A	-231991	26562	128182

Table 5.71.: Average Result Table: Run 00653

Algorithm: AODV, simulation time: 1000s , 30 nodes
Average Network Lifetime: 999.8
Energy / UserData: 20.49 μJ/B
Throughput: 234.986 kbit/s

	Energy	Mean	Minimum	Maximum	Standard Deviation
Data:	sendDataEnergy	1995.27	1392.8	2479792.0	4163.53
	rcvDataEnergy	940.983	553.44	1713250.56	4331.684
Control:	sendCtrlEnergy	355.128	310.8	538.8	49.95
	rcvCtrlEnergy	86.231	72.0	348.48	42.496
	→ *sendCtrlP2PEnergy*	503.6	503.6	503.6	0.0
	→ *rcvCtrlP2PEnergy*	348.48	348.48	348.48	0.0
	→ *sendCtrlBCEnergy*	338.897	310.8	538.8	9.849
	→ *rcvCtrlBCEnergy*	79.358	72.0	128.0	2.464
Miscellaneous:	batteryLevel (%):	2.294	0.0	27.154	4.52
	delay 80%:	3826.97	5.4	4687.77	5231.72
	delay 90%:	6257.26	5.4	7461.57	8568.26
	delay 95%:	7998.38	5.4	10889.63	11187.18
	delay 98%:	9489.12	5.4	15658.47	13891.02
	delay 99%:	10169.69	5.4	19775.15	15388.78
	delay 100%:	2248.54	5.4	210472.92	3836.54

		Sent		Received		Delivery-Ratio		Lost		Dropped	
		kB	packets	kB	packets	kB	packets	kB	packets	kB	packets
App:	Wish	123588	247178	29361	58722	23.76%	23.76%	94227	188455	N/A	N/A
	Data	84332	162323	0	0	0.0%	0.0%	84332	162323	N/A	84854
Route	Net Data	137690	265028	55488	106806	40.3%	40.3%	82201	158222	0	0
	Net Control	4218	93129	21019	461422	498.32%	495.47%	N/A	N/A	N/A	N/A
	→ *peer2peer*	374	8727	505	11776	135.03%	134.94%	-130	-3048	N/A	N/A
	→ *broadcast*	3642	79737	20514	449646	563.26%	563.91%	N/A	N/A	N/A	N/A
MAC	Net Sum	141908	358158	76508	568228	53.91%	158.65%	N/A	-210070	23416	114033

Table 5.72.: Average Result Table: Run 00654

Part II.

BEEhive Inside Linux

6. Energyefficent TCP-IP stack

By Gero Kathagen (gero.kathagen@uni-dortmund.de)

6.1. Generally

There are many ways to make the TCP-IP stack energyefficent. One can start at every layer of the stack. But most promising is the TCP layer. The following approaches are promising: "TCP-Probing"[TL], "E^2TCP"[DHSS] and "Double Retransmissions"[KSC99].

Energy is consumed by sending and receiving of packets, so there are two ways to waste energy: data- and time overhead. These two are playing together: Reduction in dataoverhead will reduce the consumed time for a transmission too. For this reason, the whole protocol is more efficient. The main problem of reliable TCP is, that it is not clear if a packet is lost by transmission-errors in a wireless environment or by congestion. Because TCP is very optimized for wired networks, it is clear: if a packet is lost, there is a congestion on the route. For resolving this problem, its wise to reduce the traffic for this connection, as a result the congestion window will be set to a much lower size. This reduces the speed of the transmission. If we are in wireless environments, the packet-losses are not only caused by congestion, the most losses are caused by a low signal or by interferences in the environment. It is wrong to conclude from a packet loss that congestion exists on the line. This prevents the use of full bandwith of the connection, as a result, there is a time-overhead.

6.2. Proposals

6.2.1. TCP-Probing

Because of this reason, V. Tsaoussidis and A. Lahanas have developed the following proposal: they introduce a "Probing-Device" and call their new protocoll "TCP-Probing"[TL]. It works as follows:

- it monitors the network traffic, and if a packet-loss is detected, it tries to analyze the reason for it (if it is due to congestion, or an error in the transmission, ...)

- it stops the traffic as long as the failure exists, and afterwards the sending will be adapted to the conditions

If a packet is delayed and possibly lost, then it holds the transfer and initiates a probing-cycle. A few packets, which only consit of headers, are sent to watch the end-to-end behavior of the connection. Once the first packet returns the second is sent. If there are reproducable packet losses, it points to a problem with the network, and the congestion-window is shrinked normally. In the other case (both packets came back without important delay), it seems to be a random packetloss and it would be wrong to reduce the congestion-window. Connection traffic is resumed normally.

The simulations results of the authors show, that this protocoll has the same or better results as TCP-Reno in all cases. But the realization in Linux-Kernel is quite complicated.

6.2.2. E^2TCP

The implementation of Donkers, E^2TCP [DHSS] is even more complicated. It takes 4 points on which to try to optimize.

First point, E^2TCP will accept a partial (adjustable) reliability. For video or audio-streams it prevents expensive retransmissions, as a result it saves a lot of energy. For normal TCP-transport this option could be turned off, but in this case, it saves no energy.
The second point is a header-compression. The most unchanged fields like source or destination etc. are sent only in the first packet, in the following they will be represented by a unique number. This saves a little bit of transfer-volume as well as energy.
The third point is concerning the selective acknowledgments. Packets which are received out of order can be acknowledged. If one packet is lost but the following packets have reached the destination there is no need to retransmit all the packets.
The last point meets the congestion-window control, in a similar way as TCP-Probing.

Recapitulating we can say, that this protocol is not appropriate for our implementation. If we have a look to the first point, we can see, that UDP can do the same. The second point, the header-compression, is very hard to implement into an existing stack inside Linux kernel. Moreover it makes the packet format non-standard. The last point, the acknowledgements, is realized inside the linux-kernel, it is turned on per default. And the forth point is like TCP-Probing.

6.2.3. Double Retransmissions

We read a paper "Double Retransmissions" [KSC99]. The main idea is, that if a packet-delivery fails, a double retransmission of the same packet has much more chances to reach its destination. It should be energy efficient, because the transmission should be finished in a shorter amount of time, and the network card can fall in a sleep-state, when the transmission is done. This should save energy because a networkcard in its "active" state (while sending) needs just a little bit more energy than in the waiting state. In sleepstate, where it is not able to send, it needs a lot less energy.

6.3. Realisation

6.3.1. Double Retransmissions

The first attempt we realized was the "Double Retransmissions" concept. We have modified the Linux-Kernel in that way, that every retransmission is done twice. It took a long time for us to find the correct function and the correct call, but in the end, we have inserted the following few lines of code at the right position in tcp_output.c:

```
struct sk_buff *skb_rt2;
skb_rt2 = skb_clone(skb,GFP_ATOMIC);
        tcp_retransmit_skb(sk, skb_clone(skb,GFP_ATOMIC));
```

Listing 6.1: additions in tcp_output.c

For testing purposes we have taken two UML (UserModLinux) machines on a workstation (Athlon 1200) and modified the `uml_switch` so that it drops a defined percentage of packets. The results are not as good as expected: In some cases, it was better than the standard_tcp, but in other cases it was even worse.

We believe there are many improvements into the Wireless-Lan-Technology, and as a result, the concept does not hold anymore. In the work of Kravets, Schwan and Calvert they work with a 915 MHz Lucent WaveLAN PCMCIA wlan card with 150 KBps, this is about 2 Mbit. Since then some improvements are made in 802.11 to have new 802.11b standard. But we did not work in a wireless environement, we have worked only with a couple of two UML-machines on a uml_switch, so that this might be a reason.

Another quite more important reason might be, that some improvements in the Linux kernel, for example TCP NewReno [Flo99] was not used in their experiments. NewReno includes some improvements for Fast Retransmission, that have an impact on the results.

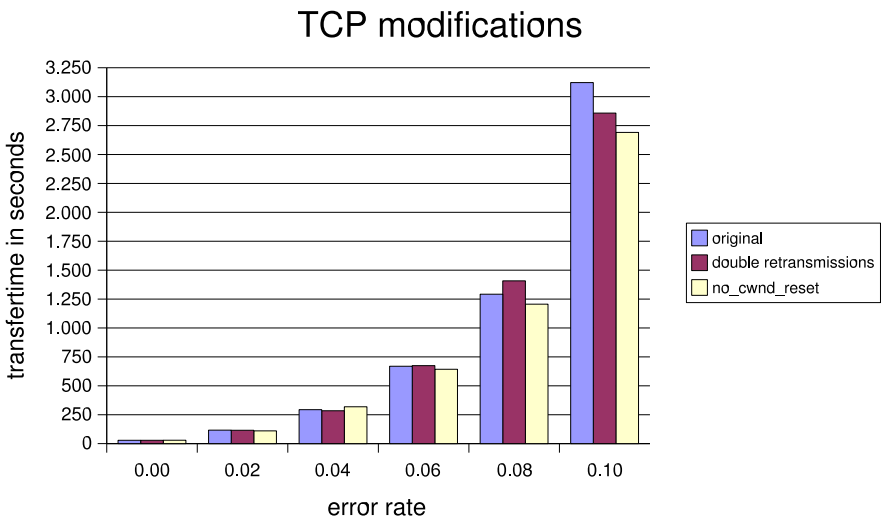


Figure 6.1.: Measurements of the variants

6.3.2. Realizability of TCP Probing

The problem with TCP-Probing is, that it assumes, we have two additional states in the statemachine of the TCP/IP stack inside Linux. The authors of this concept did not send the X-Kernel code to us. For this reason, we took the main idea, not to reduce the size of the congestion window, and set an option for disabling the function to set the size of the congestion-window to one and another option to disable the multiplication of the timeout-timer with two. For this option the patch creates two entry's in the /proc filesystem. They are initialized with the standard-value, but one can set it anytime. The first entry calls `/proc/sys/net/ipv4/tcp_no_cwnd_reset` and is normally set to 0, so that the kernel acts in a normal standard way. Setting it to 1 will anticipate the reset of the congestion window if a packet-loss is detected.

```
in: void tcp_enter_loss(struct sock *sk, int how)
        if (!sysctl_tcp_no_cwnd_reset)  tp->snd_ssthresh =
           tcp_recalc_ssthresh(tp);
        if (!sysctl_tcp_no_cwnd_reset)  tp->snd_cwnd       = 1;
        if (!sysctl_tcp_no_cwnd_reset)  tp->snd_cwnd_cnt   = 0;
```

Listing 6.2: changes in tcp_input.c

The second entry is called `/proc/sys/net/ipv4/tcp_no_timeout_timerdouble` and is also initialized with the standard-0. If it is set to 1, the timeout-timer for each packet

which is sent will not be doubled, if a packetloss is detected. These changes are done in tcp_timer.c:

```
in: static void tcp_retransmit_timer(struct sock *sk)
at: out_reset_timer:
    if (sysctl_tcp_no_timeout_timerdouble) tp->rto = min(tp->rto,
       TCP_RTO_MAX);
    else tp->rto = min(tp->rto << 1, TCP_RTO_MAX);
```

Listing 6.3: changes in tcp_timer.c

We have tested this configuration with UML and the random-drop from the uml_switch. The results with the modifications are much better than the original. So we patched two notebooks with the changes and tested it with the Wireless 802.11b network cards. At the beginning, we had no success with the tests, the influences of the environment was erratic, so that the results are not comparable. Moreover, in reality the packet losses are not as "regular" as in the UML, in reality the connection was often lost for more then 10 seconds. As a result the modification of the timeout timer works against us, the retransmission-timers fail with the retransmission of the packet and the connection times completely out. Therefore we tested only with and without the congestion_window resetting. We have done 32 runs for each parameter and got the following results: for a packet with length of 15 Megabytes transfer from one wireless node to the next it takes without our modifications 78 seconds mean value with a variance of 13. With the modification of not resetting the congestion-window, it takes the mean of 74 seconds with a variance of 16. Because of this small advantage we decided to remove our modifications inside the TCP-IP stack and to carry on developing with the linux standard stack.

The Linux Kernel Developers have added an implementation of TCP Westwood in the version change from 2.6.2 to 2.6.3, so there is a new way to save energy. TCP Westwood has a lot of optimizations for wireless connections, and is a sender-side-only modification.

7. Beehive implementation

By Lars Bensmann (lars.bensmann@uni-dortmund.de) and Mike Duhm (mike.duhm@udo.edu)

7.1. Overview

After investigating several methods of implementing a new routing algorithm for the Linux kernel we decided to use source routing in combination with the existing Linux Netfilter Architecture. The bee type is encoded into the IP header TOS field. For collecting and transporting bee information we introduced an RFC-compliant [rfc81] new IP-option. In addition to this we split the algorithm - as shown in Figure 7.1 - into a kernel space part responsible for manipulating the IP packets, and a user space part responsible for seraching new routes. These two units communicate with help of the proc filesystem and networking sockets. As a result, we get following advantages:

- Even computers not running the Beehive software can be part of the ad-hoc network as long as they accept RFC-compliant IP options, including source routing.
- Runnable with a really small kernel patch, needed for using the beehive- option.
- New code is encapsulated in a kernel module: Easy sharing and distribution.
- Relatively independent of the kernel version.
- Good for debugging, because the new code is in just a couple of new files and not scattered around the kernel source tree.
- Putting the route finding code into userspace lets us use standard C structures and library functions which are not available inside kernel space and this might improve stability of the kernel

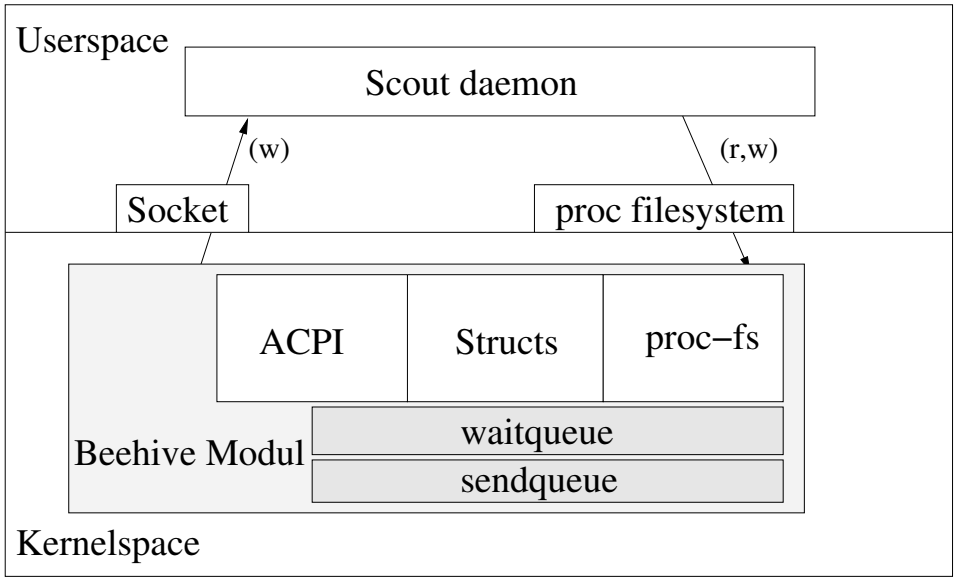


Figure 7.1.: Implementation overview

7.1.1. The Linux Netfilter Architecture

Netfilter ([net]) is the firewalling subsystem of the Linux 2.4/2.5 kernels. Although it's main purpose is to filter packets it can also do NAT (network address translation) as well as packet mangling.

Although there is no function present in the netfilter code that helps us directly with altering the routing table or decisions of the kernel there are several (five to be precise) hooks already present in the kernel that are called at different locations in the network

```
--->PRE------>[ROUTE]--->FWD---------->POST------>
    Mangle       |       Mangle   ^    Mangle
    NAT (Dst)    |       Filter   |    NAT (Src)
                 |                |
                 |             [ROUTE]
                 v                |
                 IN Mangle       OUT Mangle
                 |  Filter        ^  NAT (Dst)
                 |                |  Filter
                 v                |
```

Figure 7.2.: Netfilter hooks

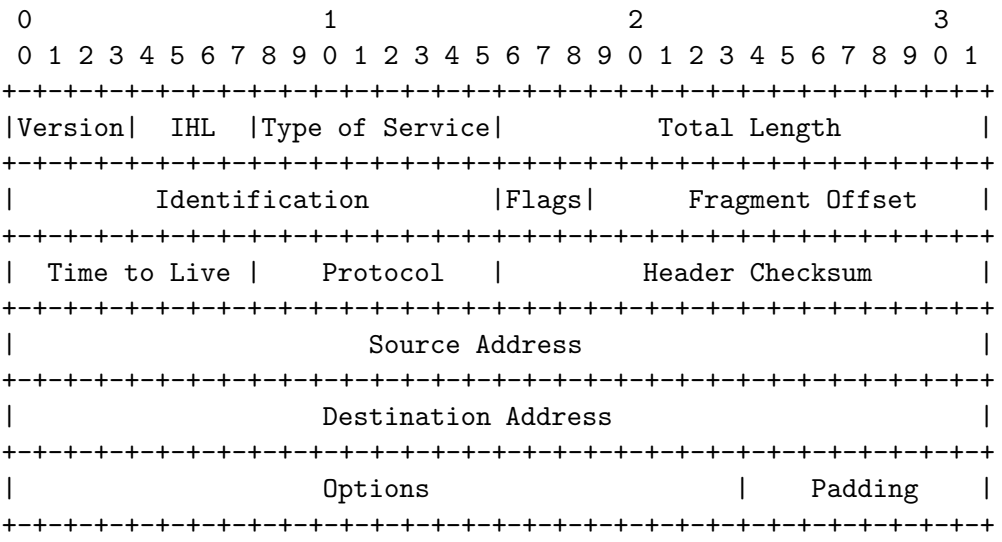


Figure 7.3.: Standard conform IP header

stack. Kernel modules may register functions that get called at each of these hooks. Figure 7.2 shows the different path of network packets through the kernel.

Remote packets enter the kernel on the left and the PREROUTING-hook is called. After this the kernel makes the routing decision. If the packet is to be delivered locally the INPUT-hook is executed and the packet leaves the kernel. Otherwise the FORWARD-hook is called. If the packet was generated locally it first passes the OUTPUT-hook and is then routed by the kernel. Now its path merges with forwarded packets. Before leaving the system both types of traffic once again are handed to a netfilter hook: The POSTROUTING-hook.

7.1.2. Using the IP header for transporting bee data

All additional beehive specific data is encapsulated inside the standard IP header of every data packet to achieve maximum compatibility with the existing networks. A standard conform IPv4 header consists of at least 20 bytes, its format is shown in fig. 7.3.

Bee types

First of all we have to define the type of bee we are using. Simulation in NS-2 works with three types of bees and so do we. The type is coded plainly into the TOS field whereas we use a straight mapping to existing TOS values. A delay bee carries the TOS

low delay bit, a throughput bee has set the TOS high throughput bit and last but not least an energy bee has the mincost bit set.

IP options

The BEEHIVE-Routing-Algorithm is based on source routing which means that a packet has all the information required for routing. In addition to this we have to collect data from every hop to evaluate the quality of a route according to our demands.

To realize this with IPv4 we use IP options which are defined in RFC 791 [rfc81]. We utilise the options field which may have a maximum length of 40 bytes and has to be a multiple of 4 bytes. There are two kinds of options:

- Type 1 consists of just one option-type octet: the NOOP (no operation) option and the EOL (end of option list) option.
- Type 2 consists of an option-type octet, a length octet and a variable count of data octets.

The option type octet has three fields:

```
Bit   7      6       5      4      3      2      1      0
   +------+------+------+------+------+------+------+------+
   | copy |option class |          option number           |
   +------+------+------+------+------+------+------+------+
```

If the copy bit is set, this option has to be copied into each fragment while fragmentation. The classes are 0 (control), 1 (reserved), 2 (debugging and measurement) and 3 (reserved).

Source routing

As the Linux Netfilter Architecture is not intended for routing algorithms we use source routing to guide the packets along their way. The source routes are inserted into (nearly) every packet by the Netfilter Beehive module (exceptions are broadcast packets). This way the routing table of the kernel consists of just one dummy entry. So all (non-local) packets are sent out through the standard WLAN network interface by the kernel without a need to apply a patch. This can work because we put all computers of the ad-hoc network in the same subnet 10.0.0.0/255.255.255.0. So the routing table of the kernel looks like this:

```
$ route
Kernel IP routing table
Destination  Gateway      Genmask        Flags Metric Ref Use Iface
10.0.0.0     *            255.0.0.0      U     0      0     0 eth0
192.168.1.0  *            255.255.255.0  U     0      0     0 eth1
default      192.168.1.1  0.0.0.0        UG    0      0     0 eth1
```

As can be seen from this example, it is still possible for a computer to be in several subnets. It does not have to use the Beehive algorithm for every subnet. If it still has a wired network interface (like in this example) it can use the conventional routing side by side with the Beehive algorithm. This enables an easy employment of the new algorithm.

There are two types Source Routing options, loose source routing (type 131) and strict source routing (type 137). We are using the second one because the complete route is calculated at the source host. The first data byte is a pointer which points to the first byte of the next hop address. Route data is composed of a series of IP addresses (four bytes per hop).

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//-+-+-+-+-+
|1 0 0 0 1 0 0 1|   length      |    pointer      | route data     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//-+-+-+-+-+
```

To send a source routed packet it has to be built in the following way: The source address has to contain the IP address of the sender, the destination address must not contain the address of the final recipient but of the next hop. The pointer has to point to byte four, the beginning of the route, which is the second hop on the way through the network. Route data begins with the third hop address and ends with the address of the final destination.

As options data is limited to 40 bytes, we will use three bytes for the source routing option beginning, 8 x 4 bytes = 32 bytes for hops, and are able to use five bytes for the beehive option. That means the longest possible route has a length of 10 hops.

The Beehive option

For data collection we introduce a new IP option. Our first task was to give our option a name. According to the database mentioned in [(Ed02] number 162 (copy, option class 1, option number 2) is free so we were able to (ab)use it for our interests. We decided to give it length five, which means having three data bytes. So we are able to collect

three bit of data from every hop, excluding source and destination hop. Our option is structured as follows:

```
byte:  1  |   2  |       3       |      4        |      5        |
   +-+//+-+-+//+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | 162  |   5  |0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|
   +-+//+-+-+//+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     type |length|     |     |     |     |     |     |     |     |
host:               8     7     6     5     4     3     2     1
```

Host 1 contains the data of the direct predecessor, host 2 the data of the pre-predecessor and so on. To insert it's own data every host has do a three bit left-shift on the 24 bit data field and to insert the value at the very right side (host 1).

7.1.3. Introduction to our implementation

As mentioned earlier we want to use the existing Netfilter Architecture. It offers us the possibility to hook us into the packet processing chain at different stages: `PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT` and `POSTROUTING` (see fig. 7.2).

Netfilter FORWARD-chain

Because all forwarded packets need to pass the `FORWARD`-chain, we decided to collect the our routing relevant data here. Afterwards we return with a `NF_ACCEPT`-code.

Netfilter OUTPUT-chain

The `OUTPUT`-chain is called for locally generated packets. This means we need to enter the source routing information into the header unless we are dealing with a scout packet. Broadcasts don't need any source routing information in the header either, because they are not routed anyway. Fortunately scouts are using broadcasts we don't need to handle them separately. We just pass broadcast packets without doing anything.

When a "regular" packet is generated, we need to check the stored foragers if a route to the desired destination is already present. If so, we just copy it into the IP header and delete the appropriate forager (unless it is the last one for this destination).

The next possibility is that no route information is present. In this case we need to check if we have already sent out a scout. If so we need to store the packet for a short amount of time and wait for the scout to return. Then we continue as described in

the last paragraph. Otherwise we send out a scout and initiate a search for the desired destination.

A single TCP connection does not generate a lot of packets before receiving an answer from the other side. This way the number of waiting packets is limited. But UDP does not wait for acknowledgements before sending the next packet. So packets to different or even the same target might need a lot of space. This might prove to be a problem when running netscans or similar traffic intense programs.

Netfilter INPUT-chain

The `INPUT`-chain is the last station of a packet before being delivered to the application. All packets destined for the local computer pass this hook. This is why we chose to use it to record all important information from the network packets.

Again we ignore broadcast. As we don't route broadcasts there is nothing for us to do. This way we ignore scouts as well (see 7.1.3).

Now we are just dealing with regular packets that are delivered through the Beehive routing algorithm. This means they contain valuable information that needs to be saved. Every packet is regarded as a forager which can find it's way back to the source computer. Of course we don't need to store the whole packet just the bits of information that are important for us. Once extracted we evaluate the efficiency with a rating function and store it in a table hashed with the source address.

7.2. Kernelspace

7.2.1. Necessary modifications inside the original Linux kernel code

Our goal was to touch the original kernel code as little as possible but a few modifications were necessary to fully include our new beehive option. Reader can find the complete kernel patch in the file `ipv4options.diff` inside our repository.

Modifications inside /include/linux/ip.h

Diff file 19 shows two passages out of the kernel header file `ip.h`. In the first part we see definitions of different IP header options. We just added the definition of our beehive option according to the description in 7.1.2.

The second part is inside the IP header data structure (`struct iphdr`). It contains kind of pointers (unsigned char) to the number of the octet inside the header in which one particular option starts. These pointers can contain values between 21, which points

to the first octet inside the options field - see 7.3, and 40, which points to the last octet inside the options field. We just added a pointer to the beginning of the beehive option here.

```
--- linux-2.6.6-orig/include/linux/ip.h
+++ linux-2.6.6-beehive/include/linux/ip.h
@@ -60,6 +60,7 @@
 #define IPOPT_SID       (8  |IPOPT_CONTROL|IPOPT_COPY)
 #define IPOPT_SSRR      (9  |IPOPT_CONTROL|IPOPT_COPY)
 #define IPOPT_RA        (20|IPOPT_CONTROL|IPOPT_COPY)
+#define IPOPT_BEEHIVE   (2  |IPOPT_RESERVED1|IPOPT_COPY)

 #define IPVERSION       4
 #define MAXTTL          255
@@ -100,6 +101,7 @@
                 ts_needtime:1,                  /* Need to record
                     timestamp                */
                 ts_needaddr:1;                  /* Need to record
                     addr of outgoing dev     */
   unsigned char router_alert;
+  unsigned char beehive;
   unsigned char __pad1;
   unsigned char __pad2;
   unsigned char __data[0];
```

Listing 7.1: Changes in ip.h

Modifications inside /net/ipv4/ip_options.c

This patch is inside the function `ip_options_compile(..)`, which is responsible for checking the correctness of IP options in incoming packets, detecting all known IP options and for filling the option pointers (see above) inside `struct iphdr` with correct values. As the beehive option is new we had to integrate finding it into this function as you can see in patch 18.

```
--- linux-2.6.6-orig/net/ipv4/ip_options.c
+++ linux-2.6.6-beehive/net/ipv4/ip_options.c
@@ -433,6 +433,14 @@
                         if (optptr[2] == 0 && optptr[3] == 0)
                                 opt->router_alert = optptr - iph;
                         break;
+                    case IPOPT_BEEHIVE:
```

```
+                       if (optlen < 3) {
+                               pp_ptr = optptr + 1;
+                               goto error;
+                       }
+                       opt->beehive = optptr - iph;
+                       break;
                case IPOPT_SEC:
                case IPOPT_SID:
                default:
```

Listing 7.2: Changes in ipoptions.c

Adding and modifying ipv4options match

To make life easier while building netfilter rules we took the ipt_ipv4options match (`ipt\_ipv4options.c`, `ipt\_ipv4options.h`) from the netfilter extensions and added it to the kernel patch. We slightly modified it to match against the beehive option. Of course the Makefile and Kconfig were adapted to be able to compile the new kernel.

7.2.2. Optional modifications inside the original Linux kernel code

In addition to this we had to make bigger changes inside the linux ACPI code to be able to collect energy data for evaluating the most energy aware route. We were forced to mesh with the code because all battery values have been declared static inside `/drivers/acpi/battery.c` and the only way to get the actual battery state was to read it from procfs. We did not change functionality but only changed accessibility of the needed data. You can find the patch in `acpi-beehive.diff` inside our repository. For people who do not like to work with modified ACPI code (and especially for UML which does not have real ACPI code) we made this patch optional. A machine without this patch or even without any ACPI support at all will be able to run inside a beehive network without any restrictions. The point is that such a machine will allways claim its battery to be full.

7.2.3. The kernel module

For this to work, we need to write the target (in this case `BEEHIVE`) as a kernel module named `ipt_BEEHIVE.c` and a shared library for the `iptables`-frontend.

To become as independent as possible from the actual kernel version we did not work inside the kernel source tree but in an extra directory. That gives us the opportunity to

work with the latest kernel version without having to modify our code (with exception of the kernel patches).
We divided the kernel module code into four units: The module specific code, like initialisation and module unloading as well as the packet handling code can be found in `ipt_BEEHIVE.c`. Data structures and constants are defined in `ipt_BEEHIVE.h`, methods for accessing and manipulating these structures are in `ipt_BEEHIVE_struct.c`. All functions regarding to the procfs can be found in `ipt_BEEHIVE_proc.c`, ACPI functionality for reading the battery state lies in `ipt_BEEHIVE_acpi.c`.
All files are joined together by `#include<>` directives inside `ipt_BEEHIVE.c`.

We start with the description of three functions that implement the core functionality of a netfilter module.

These functions are:

- `static int __init init(void)`
- `static void __exit fini(void)`
- ```
  static int ipt_beehive_checkentry(const char *tablename,
  const struct ipt_entry *e,
  void *targinfo,
  unsigned int targinfosize,
  unsigned int hook_mask)
  ```

The first two are the initialisation code which is executed by the kernel on loading or unloading the Beehive kernel module. The macro `__init` shows the kernel that this function can be unloaded from memory once the module has been loaded correctly. For initialisation a `struct ipt_target` is filled with function pointers and module name and registered with the kernel in `init()`. Inside these functions we call the initialise or rather the tidying up functions of our submodules.

The third function `ipt_beehive_checkentry` is called every time a netfilter rule is inserted with the `BEEHIVE`-target. It can make some sanity checks whether this rule is to be accepted or not. To test this functionality the module verifies that the rule is intended for the `mangle`-table. Otherwise it prints an error message and returns an error code, so the rule is not inserted into the specified netfilter table.

IP datagram manipulation is done inside the `ipt_beehive_target` function which we describe after handling the internal data structures and the API.
  ```

7.2.4. Data structures

Structures

All C `structs` are defined in the header file `ipt_BEEHIVE.h`.

The most important data structure in the kernel is the hash table.

```
struct list_head *forager_hash;
```

Listing 7.3: Main hash table.

It is an array of size `HASHSIZE` (currently 256) entries. Every entry is the head of a linked list. In these linked list entries of the form `struct daddr_list_entry` are stored.

```
struct daddr_list_entry {
        struct list_head list;

        __u32 daddr;
        int count[IPT_BEEHIVE_OPT_MAX];
        int countsum;
        unsigned long timestamp;
        struct forager* forager[IPT_BEEHIVE_ARRAYSIZE *
           IPT_BEEHIVE_OPT_MAX];
};
```

Listing 7.4: daddr_list_entry struct

For every destination IP with stored foragers one of these `struct daddr_list_entry` exists. They contain the destination IP for quick reference and housekeeping information about the stored foragers. `countsum` is the total amount of foragers stored of any type whereas `count[]` is set to the number of foragers of the given type (energy, delay, throughput). The member `timestamp` helps for the garbage collection when overaged entries are removed. The last member is the actual array containing pointers to the stored `struct forager`.

```
struct forager {
        struct source_route *route;
        __u32 daddr;
        unsigned long  timestamp;
        __u32 info;
        __u8 nr_dances;
};
```

Listing 7.5: Forager struct

These structs contain all the information we collect from the foragers that "land" on our node: Source route, destination address (or from the point of the landed forager it's source address), timestamp of its arrival, information regarding energy, throughput or delay and the number of dances used to recruit new foragers.

The `struct source_route` is just another linked list used to record all hops of the taken route.

```
struct source_route {
        struct list_head list;

        __u32 hop;
};
```

Listing 7.6: Source route struct

The BeeHive-API

To ease the programming and provide consistent locking an API was implemented. It consists of a few calls to aid in modifying the above structures.

- static int beehive_forager_get(__u32 daddr, __u8 opt_type, __u8 remove, struct forager** forager);
- static int beehive_forager_put(struct forager *forager, __u8 opt_type);
- static void beehive_struct_free_forager(struct forager*);
- static struct forager* beehive_struct_copy_forager(struct forager* forager);
- static void beehive_struct_debugp_forager(struct forager* forager);
- static void beehive_struct_debugp_hash(void);

All these calls are implemented in a single source file (`ipt_BEEHIVE_structs.c`).

On loading the module the function `beehive_struct_init()` is called. It allocates the memory for the hash table and initializes the linked list for every entry.

On unloading the function `beehive_struct_fini` is called. It calls the garbage collection `beehive_struct_gc` with a maximum age of 0 seconds. This way all entries are removed as they are considered out-dated. Afterwards the hash table is freed.

beehive_forager_get

The first essential call. It returns a `struct forager` (passed by reference) for the given `daddr`. If available a forager of type `opt_type` is returned (`IPT_BEEHIVE_OPT_ENERGY`, `IPT_BEEHIVE_OPT_DELAY` or `IPT_BEEHIVE_OPT_THROUGHPUT`). The option `remove` indicates if the forager should be deleted if appropriate (`IPT_BEEHIVE_FORAGER_REMOVE` or `IPT_BEEHIVE_FORAGER_KEEP`). Even if `IPT_BEEHIVE_FORAGER_REMOVE` is passed the forager is not necessarily removed from the kernel. If the route was good and it is still young enough it might dance and in this case a copy is made and returned instead of the original forager.

The return value indicates how many foragers are left for this destination address. If -1 is returned no forager was found and a scout should be sent out.

beehive_forager_put

The second essential call. As the name suggests it inserts a `struct forgager` into the data structure. The option `opt_type` indicates the type of the forager to insert (`IPT_BEEHIVE_OPT_ENERGY`, `IPT_BEEHIVE_OPT_DELAY` or `IPT_BEEHIVE_OPT_THROUGHPUT`). For a returned scout this call should be made three times with the three different `opt_type`s. Care should be taken not to make this call with the same `struct forager*` pointer, but instead with copies of the original forager. `beehive_struct_copy_forager` can help with this.

Before inserting the forager the garbage collection will eventually be called. The probability of these calls can be influenced by the constant `IPT_BEEHIVE_GARBAGE_PROBAB`. All destination addresses without traffic in the last `IPT_BEEHIVE_MAXAGE` seconds are removed and the used memory is freed.

The return value indicates how many foragers are left for this destination address and `opt_type`.

beehive_struct_free_forager

As the `struct forager` contains a linked list, all items need to be freed. To ease this job a simple call to `beehive_struct_free_forager` will do this.

beehive_struct_copy_forager

For the same reason mentioned above copying foragers should be left to this call. It copies all items of the `struct source_route`-list and returns a pointer to an identical forager.

beehive_struct_debugp_forager

As the name suggests this function is for debugging purposes. It prints a forager with all its information into the kernel log.

beehive_struct_debugp_hash

The second debugging call. It iterates over the complete hash table finding all destination addresses and prints information regarding the age of the entries and also about every forager. This call should be used sparingly (preferably inside an `#ifdef`-statement) as it clutters the screen and the kernel log.

Helper functions

The API-calls depend on a number of helper functions. The most important ones are explained below. These are not meant to be called directly. They are just included to help understand the implementation of the API-functions.

beehive_struct_gc

As the name suggests this is the garbage collection function. It loops through the whole hash table and examines every `struct daddr_list_entry`. If the entry is considerd too old (compared to the passed option `maxage`) it and all its foragers are deleted.

```
/*
 * Do Garbage collection.
 * Cycle through hash table and remove all daddr_list entries
 * unused for maxage seconds
 */
static void beehive_struct_gc(unsigned int maxage)
{
        static unsigned int i;
        static struct daddr_list_entry *daddr_list, *n;
        static unsigned age;

        WRITE_LOCK(&beehive_hash_lock);
        for(i = 0; i < IPT_BEEHIVE_HASHSIZE; i++) {
                list_for_each_entry_safe(daddr_list, n,
                                                &forager_hash[i],
                                                    list) {
                        age = get_seconds() - daddr_list->timestamp;
```

```
                        if(age >= maxage) {
                                remove_daddr_entry(daddr_list);
                                list_del((struct list_head*)
                                    daddr_list);
                                kfree(daddr_list);
                        }
                }
        }
        WRITE_UNLOCK(&beehive_hash_lock);
}
```

Listing 7.7: beehive_struct_gc

forager_rate_lifetime

This function determines the value of `nr_dances`. It does not set the value directly but instead returns the correct value.

First of all the length of the source route is calculated. If the destination is a direct neigbour the forager gets a maximum rating as there is no better route.

If this test fails the information gathered by the forager is examined. Every hop stores a 3-bit value in the info field. This is extracted and put into an array. From this array the minimum and the average is calculated. Based on these numbers an appropriate value is returned.

```
/*
 * Look at info and decide how often the forager should dance.
 * Direct neighbours get a maximum rating.
 */
static int forager_rate_lifetime(struct forager* forager)
{
        static unsigned int route_length;
        static __u32 info;
        static unsigned short values[IPT_BEEHIVE_MAXHOPS];
        static unsigned short i;

        route_length = get_route_length(forager);

        if(route_length < 2)     /* Direct neighbour */
                return IPT_BEEHIVE_MAX_DANCES;

        info = forager->info;
```

```
        for(i = 0; i < route_length; i++) {
                values[i] = info && 7;
                info = info >> 3;
        }

        unsigned short min = 7;
        unsigned int avrg = 0;

        for(i = 0; i < route_length; i++) {
                min = min(min, values[i]);
                avrg += values[i];
        }
        avrg = avrg / route_length;

        // Do the black magic.
        return (((IPT_BEEHIVE_MAXHOPS - (route_length-1)) * min *
            avrg) + 1) / 44;
}
```

Listing 7.8: forager_rate_lifetime

insert_forager_into_array

This is a helper function for `beehive_forager_put`. Given a forager, its `opt_type` and its `struct daddr_list_entry` the function checks if the array used to store the foragers is already full. In this case the forager is freed and discarded.

Otherwise a rating is obtained and the forager is stored at the approriate location of the forager array of the `struct daddr_list_entry`. This location is calculated from the offset for the given `opt_type` and the number of foragers already stored there. Afterwards `count[]` and `countsum` are incremented. Before returning the timestamp for the `struct forager` is set to the current time.

```
static int insert_forager_into_array(struct forager* forager,
                                     struct daddr_list_entry *list
                                         ,
                                     __u8 opt_type)
{
        list->timestamp = get_seconds();
        /*
         * Check if array is full.
         */
```

```
        if(list->count[opt_type] == IPT_BEEHIVE_ARRAYSIZE) {
                beehive_struct_free_forager(forager);
                return -1;
        }

        forager->nr_dances = forager_rate_lifetime(forager);

        list->forager[list->count[opt_type] + opt_type *
           IPT_BEEHIVE_ARRAYSIZE] = forager;
        list->count[opt_type]++;
        list->countsum++;
        forager->timestamp = list->timestamp;
        return 0;
}
```

Listing 7.9: insert_forager_into_array

get_opt_forager

This is a helper function for the API-call `beehive_forager_get`. It is already called with the correct `daddr_list_entry`, the `opt_type` and a flag if the forager is to be removed from the array.

As all foragers are stored in an array in their respective `daddr_list_entry` getting a random forager is relatively simple: Generate a random number $0 < random < count[\texttt{opt\_type}]$, add an `opt_type`-related offset and read the forager from the array.

Now we calculate the age of the chosen forager. If it is not too old and we don't want to keep it (`IPT_BEEHIVE_FORAGER_KEEP`), we can just return it.

If we want to remove the forager more care must be taken. First we check if it is a candidate for dancing. If so, we copy it and return the copy.

Otherwise we decrement the members `count[]` and `countsum`, and copy the last forager in the array to the slot where we removed our random forager. The last slot is overridden with a `NULL`-pointer to easy error detecting. This way the array is always filled from the start and selecting one at random stays very simple. At last we check the foragers lifetime and discard it if it is already too old unless it is the last forager for this destination.

It might happen that we discard the last forager for a specific `opt_type`, but more foragers of different types are still available. In this case we return `NULL` and the calling function is responsible for getting a forager with another `opt_type`.

Again at the end of the function we update the timestamp of the `daddr_list_entry` to aid the garbage collection.

```
static struct forager* get_opt_forager(struct daddr_list_entry *list,
                                       __u8 opt_type,
                                       __u8 remove)
{
        static struct forager *forager;
        static unsigned int random;
        static unsigned int age;
        static unsigned int offset;

        offset = opt_type * IPT_BEEHIVE_ARRAYSIZE;
        do {
                /*
                 * I know modulo 'count' is not really random,
                 * but it's good enough for small 'count'
                 */
                random = (net_random() % list->count[opt_type]) +
                    offset;
                forager = list->forager[random];
                age = get_seconds() - forager->timestamp;
                if(remove == IPT_BEEHIVE_FORAGER_KEEP) {
                        if(age < IPT_BEEHIVE_FORAGER_LIFETIME)
                                break;
                } else {
                        if((forager->nr_dances > 0) &&
                                (age < IPT_BEEHIVE_FORAGER_DANCETIME)
                                    ) {
                                forager->nr_dances--;
                                forager = beehive_struct_copy_forager
                                    (forager);
                                break;
                        }
                }
                list->count[opt_type]--;
                list->countsum--;
                list->forager[random] = list->forager[list->count[
                    opt_type] + offset];
                list->forager[list->count[opt_type] + offset] = NULL;
                if(age < IPT_BEEHIVE_FORAGER_LIFETIME) {
```

```
                        /* We found him. Let's exit */
                        break;
                }
                if(list->countsum == 0) {
                        break;
                }
                beehive_struct_free_forager(forager);

                if(list->count[opt_type] == 0) {
                        return NULL;
                }
        } while(1);

        list->timestamp = get_seconds();
        return forager;
}
```

Listing 7.10: get_opt_forager

7.2.5. The module core: ipt_BEEHIVE.c

As mentioned before all threads run together inside `ipt_BEEHIVE.c`. It is home of the module functionality, the packet mangling and some helping functions.

Concurrency

Ordinary netfilter code does not include concurrency. A packet is put into a chain and leaves it at the end if not dropped or rejected. Because we have to queue packets while waiting for a route there was a necessity to think about events happening almost parallel to the standard packet handling thread. Using tasklets is the solution. Tasklets are a form of soft IRQ suitable for small tasks. If scheduled by the kernel, their code is executed once, after the hardware interrupt service routines are finished next time.

Our three tasklets have serve the following purposes:

- whenever a scout comes in, the `queue_tasklet` runs through the `bufferqueue` and tries to build packets containing the destination address of the incoming scout. Packets, ready for takeoff are put on the `sendqueue`.
- the `send_tasklet` is responsible for sending out packets from the `sendqueue` into the wireless network.

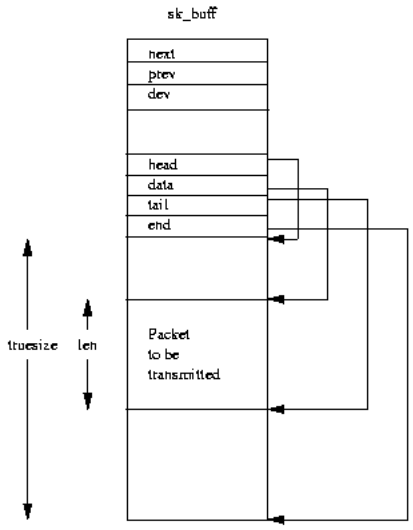


Figure 7.4.: sk_buff

- everytime the module is does not have a matching route to the destination, the scout tasklet is scheduled to send out scouts to the userspace daemon.

Core initialisation

As in every submodule we have to do some initialisation stuff inside the core. Inside `static int __init beehive_tasklet_init(void)` we initialise the tasklets, two linked lists, and create a socket, which will be used later for sending out data to the scout daemon. The linked lists are the `bufferqueue` and the `sendqueue`. We use the first one for queueing packets, waiting for a route; the second one will contain all packets which are ready to send.

The ipt_beehive_target function

The core functionality is implemented in the `ipt_beehive_target()`-function. This function is called with a pointer to a so called sk_buff (see 7.4). A sk_buff is a data region and a collection of pointers (see fig. 7.4). Its pointers point to the network headers of the different layers in the data region. This way the payload can be copied into the region. Network headers of different layers can now be copied in front of (and eventually after) the payload data without moving or copying it.

This sk_buff also contains a pointer to an IP options struct, but unfortunately we cannot use this. The struct is filled while the packet is generated by the kernel. Afterwards the packet is build, so that the data area of the sk_buff is already filled with the RFC-compliant IP-header directly followed by the payload data.

Independent from the hook from which the function is called, the first action is to recognise the type of bee it is actually handling from the IP header TOS field.

```
__u8 tos = iph->tos;
__u8 bee_type = IPT_BEEHIVE_OPT_ENERGY;
if ((IPTOS_TOS(tos) && IPTOS_LOWDELAY)==IPTOS_LOWDELAY)
        bee_type = IPT_BEEHIVE_OPT_DELAY;
if ((IPTOS_TOS(tos) && IPTOS_THROUGHPUT)==IPTOS_THROUGHPUT)
        bee_type = IPT_BEEHIVE_OPT_THROUGHPUT;
if ((IPTOS_TOS(tos) && IPTOS_MINCOST)==IPTOS_MINCOST)
        bee_type = IPT_BEEHIVE_OPT_ENERGY;
```

After getting the bee type we branch depending on the hook. The most interesting case is an outgoing packet which means we are inside the OUTPUT chain.

Netfilter OUTPUT-chain

The main code in the output chain is really compact. It asks the data structure for a forager to the matching destination address with a correct bee type. If it gets such a forager it is able to build the route into the packet by the function `ipt_beehive_build_packet(forager, sk_buff)` and to return NF_ACCEPT. This should be the most usual case, so that we do not produce much calculation overhead in our code (we will show you the overhead of the build function later).

If we do not get back a suitable forager we set `nextScoutDest` to the destination address, schedule the `scout_tasklet`, store the corresponding sk_buff in the bufferqueue and report it as `NF_STOLEN`. Fate of that packet is now in the hands of our module, we take it out of its ordinary way through the kernel.

```
struct forager *my_forager ;
if (beehive_forager_get(iph->daddr, bee_type,
        IPT_BEEHIVE_FORAGER_REMOVE,
        &my_forager)== -1) {
                nextScoutDest = iph->daddr;
                tasklet_schedule(&scout_tasklet);
                skb_queue_tail(bufferqueue, skb);
                return NF_STOLEN;
                } else {
```

```
                        if (ipt_beehive_build_packet(
                            my_forager, skb)
                                        ==-1) {
                                return NF_DROP;
                        } else {
                                skb->nfcache |= NFC_UNKNOWN;
                                return NF_ACCEPT;
                        }
                }
```

Listing 7.11: Output chain code

The `scout_tasklet` uses the socket, created in the init phase, to send out a scout packet for the address, contained in `nextScoutDest` to the daemon when running next time. If a scout comes back its destination address is written into `LastForagerDest` and the `queue_tasklet` is scheduled.

Functionality of the `queue_tasklet` wrapped inside a loop over the bufferqueue. As it knows a scout came in carrying a route to the address stored in `LastForagerDest`, it takes every packet having that destination out of the bufferqueue and does nearly the same as the OUTPUT-chain code then: It tries to get a forager and puts the route into the packet. If a packet cannot be built because of a missing forager a new scout will be sent out and the loop breaks - no more packets to the particular destination can be sent. The difference to the OUTPUT-chain code is, that we are outside the standard thread; every packet has to be sent manually. To manage this, completed packets are put onto the sendqueue and the `send_tasklet` is scheduled if one or more packets are built successfully.

```
static void QueueTaskletFunction (unsigned long data)
{
        __u32 ip = LastForagerDest;
        do {
                next_sk_buff = temp_sk_buff->next;
                iph = temp_sk_buff->nh.iph;
                if (iph->daddr==ip) {
                        if (beehive_forager_get(iph->daddr,
                                        IPTOS(iph->tos) ,
                                        IPT_BEEHIVE_FORAGER_REMOVE,
                                        &my_forager)== -1) {
                                nextScoutDest = iph->daddr;
                                tasklet_schedule(&scout_tasklet);
                        } else {
```

```
                                skb_unlink(temp_sk_buff);
                                ipt_beehive_build_packet(my_forager,
                                                         temp_sk_buff)
                                                            ;
                                skb_queue_tail(sendqueue,
                                    temp_sk_buff);
                                need_to_send = 1;
                        }
                }
                counter--;
                temp_sk_buff = next_sk_buff;
        } while (counter>0);
        if (need_to_send == 1 )
                tasklet_schedule (&send_tasklet);
        }
}
```

Listing 7.12: QueueTaskletFunction - Read the queue and build packets

Sending out queued packets is done by the `send_tasklet`. It removes the first sk_buff from the sendqueue and instructs the kernel to reroute it. This has to be done because we manipulated the destination address. While rerouting the kernel for example looks up the correct MAC address for the next HOP. Now the packet will be sent out. If the sendqueue is not empty it schedules itself for sending out the next packet. We did not use a loop over the sendqueue to save system ressources by keeping this interrupt action as short as possible.

```
 static void SendTaskletFunction (unsigned long data)
{
        int counter = skb_queue_len(sendqueue);
        struct sk_buff *temp_sk_buff;
        struct iphdr* iph;
        if (counter>0) {
                temp_sk_buff = skb_dequeue(sendqueue);
                iph = temp_sk_buff->nh.iph;
                skb_unlink(temp_sk_buff);
                temp_sk_buff->nfcache |= NFC_ALTERED;
                ip_route_me_harder(&temp_sk_buff);
                dst_output(temp_sk_buff);
                if (skb_queue_len(sendqueue)>0)
                        tasklet_schedule( &send_tasklet);
        }
```

```
}
```

Listing 7.13: SendTaskletFunction - Sending out packets

The central function - invoked by the OUTPUT-chain code as wall as by the `queue_tasklet` code is `ipt_beehive_build_packet`. Its job is to take the sourceroute from the given forager and to put it into the IP packet header. In addition to this it has to add the beehive option exactly there, too. The main problem was to figure out the proper way to add the IP options. Unfortunately we cannot be sure that there is enough room free in front of the transport layer header to move it to the front and add our IP options afterwards. So we need to make some room in front of the IP header if necessary. This is done with a call to the function `skb_cow(struct sk_buff *skb, int headroom)`. (The parameter `headroom` is the amount of bytes to reserve in front of the data area. If the headroom is already large enough this call does nothing.) Now we have to `memmove` the IP header to the beginning of the buffer, so that exactly the number of bytes we need between the header and the payload data is freed. After changing some pointers so that the new beginning of the network header is known and after changing the length of the packet and the IP header, we can go on filling the freed space with our own options.

```
opt = &(IPCB(my_skb)->opt); // pointer to options array
if (opt)
        oldoptlen=opt->optlen;
inclen = (route_length>1) ? IPT_BEEHIVE_OPTION_LENGTH
        + route_length*4 - 1 : IPT_BEEHIVE_OPTION_LENGTH ;
fillen = (4 - (inclen + oldoptlen)%4)%4;
inclen+=fillen;
iphlen = iph->ihl + (inclen >> 2);

/* testing, if options length is still OK*/
if (iphlen > 15) {
        printk( "Sorry. Headerlength %i is too big \n", iphlen);
        return -1;
}
if (skb_cow(my_skb, 60 - iphlen)) {
        DEBUGP("skb_cow failed\n");
        return -1;
}
new = skb_push(my_skb, inclen);
if (opt) {
        memmove(new, new + inclen, sizeof(struct iphdr) +
                                        opt->optlen);
```

```
} else {
        memmove(new, new + inclen, sizeof(struct iphdr));
}
```

Listing 7.14: creating space for IP options inside an sk_buff

If the length of the route is equal to one, which means we are sending data to our neighbour host, we just have to include the beehive option. if it is greater than one we have to include the beehive option and source routing information into the header. Of course we have to manipulate the destination address in this case. To achieve options length to be a multiple of four we will add padding in form of NOPs.

```
ipopts[0] = IPOPT_BEEHIVE;
ipopts[1] = IPT_BEEHIVE_OPTION_LENGTH;
ipopts[2] = 0;  // BeeHive data
ipopts[3] = 0;  // BeeHive data
ipopts[4] = 0;  // BeeHive data

if (route_length>1) {
        ipopts[5] = IPOPT_SSRR;
        ipopts[6] = (route_length*4)-1;
        ipopts[7] = 4; /* pointer */

        /* set destination address to next hop: */
        iph->daddr = route[1];
        opt->faddr = route[1];

        /* insert hops into ipopts */
        for (count=2; count<=route_length; count++) {
                int* test = (int*)&
                        ipopts[IPT_BEEHIVE_OPTION_LENGTH
                        +3+(count-2)*4];
                *test=route[count];
        }
}

/* fillup options with NOOPs to reach   length % 4 = 0 */
for (count=0; count<fillen; count++){
        ipopts[inclen-1-count]=IPOPT_NOOP;
}
```

Listing 7.15: Inserting IP options

Netfilter FORWARD-chain

Inside the FORWARD-chain bees we put relevant information about the HOP into each bee, flying by, dependent on the bee type. In contrast to the simulation we are only able to collect energy data in reality. Energy data, represented by the remaining battery capacity, stored in `ipt_beehive_battery_status`, is put into the IP packet by the `energy_check` function.

For mapping the real data to a three bit value we use the mapping function similar to the one inside simulation. Next step is to read the value of the existing beehive option. As we are not able to work on 24 bit values directly (remember: the beehive option data field consists of 24 bit, three bit for each host), we additionally take the length octet and put these four bytes into an __u32, which has to be converted from networking to host byte order. After the length octet is stored in an extra variable, because we have to restore it, we just shift beedata left about three bit and put the actual battery data of our host into the very right three bit. Bee data can now been written into the IP packet, the procedure is finished after restoring the length octet. Handling of other data, like delay or throughput data, is works in a similar way.

```
static void energy_check (struct sk_buff *skb)
{
        int batStand = ipt_beehive_battery_status;
        unsigned char batLevel=0;

        if (batStand <= 9) batLevel = 0;
        else if (batStand > 9 && batStand <= 14) batLevel = 1;
        [.....]
        else if (batStand > 65) batLevel = 7;

        struct ip_options *opt;
        opt = &(IPCB(skb)->opt);
        unsigned char* bee_ptr = (unsigned char*) skb->nh.iph +
                        opt->beehive +1;
        __u32* bee_data = (__u32*) bee_ptr;
        unsigned char saveme = *bee_ptr;
        __u32 beedata = ntohl(*bee_data);
        beedata = beedata <<3;
        beedata |= batLevel;
        *bee_data=htonl(beedata);
        *bee_ptr= saveme;
```

```
}
```

Listing 7.16: Energy check function

Netfilter INPUT-chain

In `LOCAL_IN` we have to produce a forager from every incoming IP packet. First of all we extract the bee data field from the IP options. As we have to extract a 24 bit data field we have to do some shifting to get the correct value. If the source of the packet is a computer, connected directly to this host, we are now able to create a forager, containing bee data and the source address of the IP packet, otherwise we have to extract source routing information and to add them to the forager. The next step is to add the forager, together with the bee type (see above, TOS field) to the data structure by using `beehive_forager_put(newForager, bee_type)`.

We have noticed that most applications answer source routed packets by using the reverse source route on the way back. To prevent this we will have to delete source routes from all incoming packets and fill up the options with NOOP options.

Using ACPI

If ACPI is deactivated `ipt_beehive_battery_status` will allways pretend the remaining battery capacity to be 100%. But if the kernel is patched (see 7.2.2) and the system is capable of using ACPI we are able to read the correct battery state at every moment.

All implementation details about using the ACPI battery state can be found in `ipt_BEEHIVE_acpi.c`. During initialisation the battery handle has to be found. We use `acpi_get_devices` for this task. The first parameter is the internal name for ACPI batteries, the second is the name of a little call back function, doing nothing but putting the handle of the first found battery into the battery struct, which is handed over as a third parameter. Now we are able to access the actual battery data. For reading the remaining battery capacity regularly we use a kernel thread. There are two reasons for this: a) we can do the reading independent from all other iptables code and b) we cannot access battery data from the interrupt context (taking a kernel timer would have been our choice then). Starting the kernel thread is the last thing which is done during initialisation of this submodule.

```
static int __init
beehive_acpi_init (void)
{
// Battery HID is PNP0C0A
```

```
        battery = kmalloc(sizeof(struct acpi_battery), GFP_KERNEL);
        // Look for acpi_battery
        acpi_get_devices("PNP0C0A", battery_probe, battery , NULL);

        init_waitqueue_head(&wq);
        ThreadID=kerneBatteryWatcherFunctionrFunction, NULL,
            CLONE_KERNEL);
        if (ThreadID==0)
                return -EIO;
        return 0;
}
```

Listing 7.17: ACPI initialisation

A kernel thread is handled like an ordinary userspace process (you can even see it inside the process table) but is able to access every kernel data structure. After it started it is daemonized, which means put into the background. Otherwise the insmod process would be blocked until the thread dies. To be able to kill it - and to unload the module - sending `SIGTERM` has to be allowed. As we do not want to run it all the time and to block everything else, the thread is sleeping for most of its lifetime on a waitqueue. Every five seconds it wakes up, puts the remaining capacity into `ipt_beehive_battery_status` and sleeps again. It does so until it catches a `SIGTERM`.

```
static int BatteryWatcherFunction( void *data )
{
        daemonize("BatteryWatcher");
        allow_signal( SIGTERM );
        while (TRUE) {
                timeout=HZ * 5;
                timeout=wait_event_interruptible_timeout( wq, (
                    timeout==0), timeout );
                acpi_battery_get_status( battery, &bat_stat );
                acpi_battery_get_info( battery, &bat_info );

                int bat_max = (int) bat_info->design_capacity;
                int energy = (int) bat_stat->remaining_capacity;

                if (bat_max == 0){      // no battery -> power from
                    line
                        ipt_beehive_battery_status =
                            IPT_BEEHIVE_BATTERY_FULL;
                } else {
```

```
                        ipt_beehive_battery_status =
                            IPT_BEEHIVE_BATTERY_FULL
                                * energy / bat_max ;
                }
                if (timeout==-ERESTARTSYS ) {
                        ThreadID=0;
                        break;
                }
        }
        complete_and_exit ( &OnExit, 0);

}
```

Listing 7.18: Battery watcher kernel thread

7.3. Communication between kernel and scout-daemon

By Gero Kathagen (gero.kathagen@uni-dortmund.de)

7.3.1. ProcFS Entrys

The scout-daemon is taken into userspace, because it has a few advantages not to be inside the kernel. The main benefits are, that in userspace its more easy to program, there are a lot of usefull data structures we can use, and its easier to send and to handle UDP-packets which have data bytes in the body of the packet. Because the scout runs in userspace whereas the other functionality is inside the kernel, they need a way to communicate and interact. This can happen with syscalls or via the ProcFS system. One reason to do it via the ProcFS is, that its easier to implement and a subgroup could work on its own part without need to interact with the other subgroups of the project group everytime. We have implemented and associated the Scout-Daemon without need a ProcFS-interface. We could independently test our ProcFS extensions via cat or echo, without need for a functioning Scout-Daemon.

The ProcFS is a virtual filesystem mounted onto the filesystem-tree. It is provided by the kernel itself. Some of the entries only display information, some display settings and allow to change them, and the some entries are just able to write informations to the kernel.

At the beginning of the programming work we have to decide, where to put the model inside the procfs. For our purposes its usefull, to put it into one of the "net" sub-directories, there are `/proc/net` or `/proc/sys/net`. Because for creating and using

/proc/sys/ subdirectory there are a system controll tables, which are built before compiling. So we decided to create a folder called /proc/net/beehive, which consists an input-file, called /proc/net/beehive/route_in, in which new routes can be written by the scoutdaemon. For each destination it will create one entry, and if anyone reads it, it gives a random choice of any route to this destination stored inside the kernel.

```
/proc | ...
      |-net| ...
           |-beehive | route_in to write new route to the kernel
                     | dest_1 to read a random-route to dest_1
                     | dest_2 ...
                     | ...
```

7.3.2. scoutdaemon to kernel

As mentioned above, a source-route is pushed from the scout-daemon to the kernel via the route_in entry. This entry will be created at module loading time.

```
beehiveDir = proc_mkdir( "beehive", proc_net );
inputFile = create_proc_entry( "route_in", S_IWUGO, beehiveDir );
...
inputFile->write_proc = ProcWrite;
```

Listing 7.19: initialisation of beehive in procfs

With this (incomplete) listing of commands the directory will be created, that has (at the moment) for all users write permissions. If anyone writes something into the route_in-file, the function ProcWrite is called.

The scout-daemon has to give the route in the following format:

```
DDCCBBAA:HHGGFFEE:LLKKJJII:.... :ZZYYXXWW,
in words:
Hop1    :Hop2    :Hop3    :.... :Destination
```

where the signs between the ":" are IP-addresses in IPV4 hexadecimal network-byteorder. For example: 192.168.1.70 in network-byteorder is 70.1.168.192, and its translation to hexadecimal it is 4601A8C0. So, if the string: 4601A8C0:4701A8C0:4501A8C0:4401A8C0 is written to the route_in-Entry, it will create a forager for 192.168.1.68 over the Hops 192.168.1.70, 192.168.1.71, 192.168.1.69.

ProcWrite uses the KernelBuffer, where the text written to route_in is found, and does some little correctness-checks on it. If it passes sanity check then eight characters

are joined into a `__u32` value. Then it appends it to a list. At the end of the given string, there is a complete source-route, which have to be inserted into a forager-struct.

```
while (i<Hops){
                [...]
                __u32 address_u32 = simple_strtoul (Adresse, NULL ,
                    16);
                struct source_route *nextHop = kmalloc(sizeof(struct
                    source_route),GFP_KERNEL);
                nextHop->hop = address_u32;
                [...]
                list_add_tail (&nextHop->list , &route->list);

                if (i == Hops-1) {
                         __u8 opt_type;
                        for(opt_type = 0; opt_type <
                            IPT_BEEHIVE_OPT_MAX-1; opt_type++) {
                                struct forager *otherforager =
                                    beehive_struct_copy_forager(
                                    newForager);
                                beehive_forager_put(otherforager,
                                    opt_type);
                        }
                        beehive_forager_put(newForager,
                            IPT_BEEHIVE_OPT_MAX - 1);

                        LastForagerDest= newForager->daddr;
                        tasklet_schedule(&queue_tasklet);
                }
                i++;
        }
}
```

Listing 7.20: functionality of route_in in /proc/net/beehive/

In this section the route is packed into a list of hops and finally a forager is created. This forager is copied for each possible optimisation type and inserted into the data structure with beehive_forager_put.

If there is no forager other for this destination in the structure, then it creates a new entry with the following function:

```
static void BeeProcInitFunction(void *_destaddr)
{       [...]
```

```
        newEntry = create_proc_entry( DestAddr,S_IRUGO, beehiveDir );
        newEntry->owner = THIS_MODULE;
        newEntry->read_proc = BeehiveProcRead;
        newEntry->data = kmalloc(sizeof(__u32), GFP_ATOMIC);
        memcpy(newEntry->data, destaddr, sizeof(__u32));
        [...]
}
```

Listing 7.21: creation of route-entries in procfs

This creates a new entry called as a string in DestAddr with the parent directory beehiveDir (`/proc/net/beehive`). Then it sets the "data" field to the identifier for this entry. This is important because at this field one can decide which entry has called the BeehiveProcRead function. It is called once a scout-daemon writes to the `route_in` or a forager arrives from an unknown destination.

7.3.3. kernel to scout-daemon

Every time, a process reads out a forager via the entry in the ProcFS, the read-function is called. This can be done by the scout-daemon, or a simple "cat" command. With the data-field, we know, which file is read and we can ask the structure to get us a random route. This route have to be written in a string and given out.

```
static int BeehiveProcRead( char *buf, char **start, off_t offset,
                            int size, int *eof, void *data)
[...]
int erf = beehive_forager_get(*(__u32*)data, IPT_BEEHIVE_OPT_ENERGY,
                                                    IPT_BEEHIVE_FORAGER_KEEP
                                                        , &
                                                        newForager
                                                        );
[...]
list_for_each_entry(hop, newForager->route, list)
                                BytesWritten += snprintf(buf +
                                    BytesWritten, size - BytesWritten,
                                     "%08x:", hop->hop);
[...]
```

Listing 7.22: reading a route out

These are the essential commands of the BeehiveProcRead-function. We simply get a copy of a forager for the destination and print it to the output buffer (BytesWritten).

With the return-statement the length of the written string is given and the kernel prints it to the file-handle. The output is formatted in the same way as input except entry for destination is omitted from string.

In the normal transfer-behaviour its possible, that the last forager is sent, and there is no other forager for his destination. In this case, we do not know any route to the destination, so there is no need for a entry in the proc-fs anymore. In the communication-API for the procfs we have also a function for removing an entry, and this is called if the last forager is read out. The function is called BeeProcRemove.

7.4. Userspace

7.4.1. iptables shared libraries

Iptables BEEHIVE-target

Writing the shared library for iptables was the easiest part. It does not need to check any parameters, so there is very little logic inside the code. Consequently the code was copied from the `TARPIT`-target and modified to support the `BEEHIVE`-target. The code for this library can be found in the repository at the following location:
`/oseg/BEEhiveTools/iptables/extensions/libipt_BEEHIVE.c`

The code consists of mostly empty (required) functions and a struct with pointers to these.

Iptables IPV4OPTIONS-match

The shared library for the ipv4options match is included in the standard iptables distribution. We just had to modify it to be able to direct iptables to match against our beehive option. The usage stays the same, we just added `--beehive` for matching beehive packets and `! --beehive` for matching anything but beehive packets into the known command line options of this module.

Using iptables

We want our algorithm to work on every outgoing IP packet on the specified interface to the beehive network. Exceptions to this rule are packets directed to ourself (to the IP of our outgoing beehive interface), packets to the broadcast address and packets to UDP port 1124. All other packets on the beehive directed interface have to pass the BEEHIVE target. Reasons for these exceptions are obvious: a) We do not to ask the

```
# outgoing traffic
iptables -A OUTPUT -t mangle -d {IP} -j ACCEPT
iptables -A OUTPUT -t mangle -d {BCAST} -j ACCEPT
iptables -A OUTPUT -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A OUTPUT -t mangle -d {IP}/{NETMASK} -j BEEHIVE

#incoming traffic
iptables -I INPUT -t mangle -m ipv4options --beehive -j BEEHIVE

#traffic to be forwarded
iptables -A FORWARD -t mangle -m ipv4options --beehive -j BEEHIVE
```

Figure 7.5.: Configuring netfilter for our needs

way to ourselves, b) taking influence on the route of broadcasted packets does not make sense, c) our scout daemon is listening on port 1124, we do not have to care about scout packets.

Only incoming packets carrying the beehive option are interesting for our algorithm to build foragers from them. All others can be ignored by us.

It is the same principle for forwarded packets. Just the ones carrying the beehive option have to be modified to insert the specific bee data for the actual host.

The resulting configuration is shown in fig. 7.5.

7.4.2. scout-daemon

The scout daemon takes care of an essential part of the beehive routing protocol. It is responsible for searching and finding new routes. This task is not done inside the kernel, but in the user-space by the scout daemon.

On every host of our network the scout daemon is running. If the kernel wants to send a packet to a certain host and recognizes that no route to that host is known it finds a route to that destination by sending an initial scout to the local scout daemon. A scout is a udp packet containing and collecting information about the route which it is searching for, that means the source, the hops, and the destination.

```
typedef struct {
    uint8_t mode, id;
    uint8_t ttl, nh;
    iaddr src, dest;        // network byte order
    iaddr hop[MAXHOPS];     // network byte order
```

```
} packet;
```

Listing 7.23: the scout datastructure

Every scout daemon which has received a scout adds its ip address to the route inside the packet and rebroadcasts it, so that all reachable hosts can receive it. If the scout reaches the destination it is sent back to the source via source routing where the scout daemon imparts the route to the kernel via the proc filesystem.

To reduce the number of scouts sent, we use three extensions:

1. Every scout gets a unique ID from its source host and every host ignores scouts with an ID and source it has already seen.

2. On every host we have a look at the dance-floor via the proc filesystem if that host already knows a route to the destination. If so we take that route and do not continue broadcasting.

3. We start scouting with a small ttl (time to live). If the source daemon does not receive a reply within a certain amount of time the ttl is increased and the scout is sent again and so on. Relating to the simulation group this decreases the total number of scouts sent.

implementation of the scout daemon

The implementation of the scout daemon consists of a couple of files:

- **scoutd.h** The paths to the configuration file and dancefloor-directory are stored in this file as well as the optimization parameters of the beehive algorithm.

- **scoutd.c** This is the main program. Here the daemon starts and the configuration file is read.

- **config.l config.y** These are files for `flex` and `bison` used to parse the configuration file.

- **dancefloor.h dancefloor.c** This is the interface to the proc filesystem. The two functions `search_on_dancefloor` and `write_to_dancefloor` are implemented here. While writing a route to the dancefloor is rather simple, the search is more complicated, as we have to check if the new route contains loops. This is possible because the new route consists of the part from the dancefloor and the part already found by broadcasting.

```
 // search on dancefloor for a route and store it in p;
 // returns 0 on success, -1 if no route is found
int search_on_dancefloor (packet *p);

// write route stored in p to dancefloor
void write_to_dancefloor (packet *p);
```

Listing 7.24: dancefloor functions

- **list.h list.c** Implementation of a simple list used by scouttimer
- **scouttimer.h scouttimer.c** The functions here are needed for resending scouts after a timeout with an increased ttl. A datastructure is provided in which the scouts wait for their next flight.

```
typedef struct {
    list l;
} scout_timer;

typedef struct {
    struct timeval send_time;
    packet p;
} st_node;

// initialize scout_timer
void scout_timer_init (void);

// returns 1 if next scout is due to fly,
// otherwise 0 and the time after he is due
// is stored in waiting_time
int next_scout_due (struct timeval *waiting_time);

// delete next scout from list and store it in p
void get_next (packet *p);

// append scout to list
void append (packet *p);

// remove scout for dest from list
// (only one, more shouldn't be in)
void remove_dest (iaddr dest);
```

```
// returns 1 if scouting for dest is in progress, 0 otherwise
int search_dest (iaddr dest);
```

Listing 7.25: scouttimer functions

- **scoutlist.h scoutlist.c** Implementation of the scoutlist, which is in fact an array. Here we store the IDs and sources of the incoming scouts to avoid broadcasting the same scout again and again.
- **scouting.h scouting.c** This is the most important part of the scout daemon. The actual scouting is done here. Have a look at the following listing, which gives an idea about the algorithm. It is strongly shortened and will not function at all this way.

```
static void send_back_scout (int sock, struct sockaddr_in *back,
                             packet *p, int nh) {
  p->mode = 2;
  if (nh) {             // there are hops
    ip_opts[0] = 1;     // noop
    ip_opts[1] = 0x89; // strict source routing
    ip_opts[2] = (nh << 2) + 3;
    ip_opts[3] = 4;
    for (i = 1; i <= nh; i++)
      memcpy(&(ip_opts[i<<2]), &(p->hop[--p->nh]), 4);
    if (setsockopt(sock, SOL_IP, IP_OPTIONS, ip_opts,
                   (nh+1) << 2) < 0)
      errmsg_exit("Setting ip_options failed, exiting.");
  }
  if (sendto(sock, p, sizeof(*p), 0, (struct sockaddr *) back,
             sizeof(*back)) < 0)
    DEBUGP("Could not send packet, error %d", errno)
  if (setsockopt(sock, SOL_IP, IP_OPTIONS, ip_opts, 0) < 0)
    errmsg_exit("Setting ip_options failed, exiting.");
}

void scouting (scoutd_config *sd_config) {
  while (1) {
    if (next_scout_due(&waiting_time)) {
      // scout in list which wants to be sent
      get_next(&p);  // get this scout and send it
      if (sendto(sock, &p, sizeof(p), 0, (struct sockaddr *) &bc,
```

```
                sizeof(bc)) < 0)
      DEBUGP("Could not send packet, error %d", errno)
    // put it back into list with increased ttl
    p.ttl += TTL_INCREASE;
    if (p.ttl <= TTL_MAX)
      append(&p);
  }

  timer.it_value = waiting_time;
  if (setitimer(ITIMER_REAL, &timer, NULL) < 0)
  errmsg_exit("Error with timer, exiting.");

  if ((l = recv(sock, &p, sizeof(p), 0)) < 0) {
    if (errno == EINTR) // interrupted by timer
      continue;
    else
      errmsg_exit("Error receiving packet, exiting.");
  }

  if (p.mode == 0) { // initial scout
    if (search_dest(p.dest)) {
      // scout with this destination already flying
      continue;
    }
    p.mode = 1;
    p.id = ++id;
    p.ttl = TTL_INITIAL;
    if (sendto(sock, &p, sizeof(p), 0, (struct sockaddr *) &bc,
                sizeof(bc)) < 0)
      DEBUGP("Could not send packet, error %d", errno)
    // put it into list for possible later resending
    p.ttl += TTL_INCREASE;
    if (p.ttl <= TTL_MAX)
      append(&p);
  } else if (p.mode == 2) { // scout flying back to source
    if (p.src == ipaddress) { // scout back at source
      write_to_dancefloor(&p);
      remove_dest(p.dest); // remove scout from scouttimer
    }
  } else if (p.dest == ipaddress) {
    // we are the destination -> route found
```

```
      send_back_scout(sock, &back, &p, p.nh);
    } else { // we are a hop
      if (!in_scout_list(&p)) {
        p.hop[p.nh++] = ipaddress;
        if (search_on_dancefloor(&p) == 0) {
          // we found a route to destination on dancefloor
          send_back_scout(sock, &back, &p, nh);
        } else if (--p.ttl) {
          // no route found on dancefloor -> keep searching
          if (sendto(sock, &p, sizeof(p), 0,
              (struct sockaddr *) &bc, sizeof(bc)) < 0)
            DEBUGP("Could not send packet, error %d", errno)
        }
      }
    }
  }
}
```

Listing 7.26: the scouting algorithm

- **createscout.c** This is a standalone program used for testing. The `cs` command creates initial scouts and sends them to the local scout daemon. This task is normally done by the kernel.

using the scout daemon

The scout daemon is simply started by `scoutd` and stopped by `scoutd stop`. It needs a configuration file (default is `/etc/scoutd.conf`) which includes the ip address of the local host. One can also change the netmask and port here if they differ from the defaults `255.255.255.0` and `1124`. For example the file could look like this:

```
address 192.168.0.71
netmask 255.255.255.0
port 1124
```

The daemon communicates with the kernel via the proc filesystem. Before a scout is broadcasted the daemon searches for a route in `/proc/net/beehive/<destination>`. Only if a route to the destination host does not exist (that means the file does not exist) broadcasting is done. Found routes are stored in `/proc/net/beehive/route_in`.

8. Testing

8.1. Testing-Design overview

The testing we have to do can be divided into two parts: functional testing and performance-testing. And we have to test two environments: Our module in an UML-environment and in reality on the notebooks. The functional testing is possible in both environments, but the measurement of the performance is quite difficult. To test our implementation, we have to create reproducible conditions.

If we want to test a static situation, we can do this in a reproducible way, but the interesting part of the algorithm is the mobile behaviour. To reproduce a mobile behaviour with our budget is impossible. For this reason only functional tests are done in reality.

In the UML-environment we have other problems: the CPU-power of the hostsystem is shared by all UML-machines. But the behaviour of the simulated network and the transfers are reproducible by time-related scripts and the UML-Switch.

8.2. Testing-Environment

8.2.1. Reality

For testing with real devices the Lehrstuhl gives us five Notebooks with 802.11b wireless lan integrated, 2.4 GHz Intel Pentium 4 CPU. We have developed the following scenario in fig. 8.1 for testing the functionality of finding new Routes.

8.2.2. UserModeLinux

Switch daemon

After calling `testenv <number>` the given number (from 1 to 9) of UML computers start. Possible are 1 to 9 instances. However this is easily adaptable. The network is realized by a switch daemon. The UML switch available does not support dynamic connections: all UML instances are linked with each other. So we had to change that.

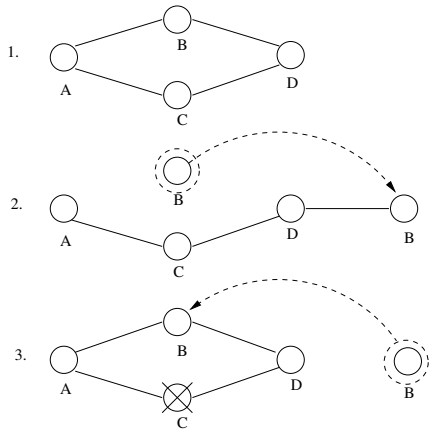


Figure 8.1.: Scenario for functional testing with real equipment

Now the switch daemon reads a configuration file when it is starting with an appropriate parameter (`uml_switch -wlan <filename>`). In this file the connections are listed which should exist. Here is an example:

```
default 0

merkur <-> venus 100
venus <-> erde 100
erde <-> jupiter 100
merkur <-> mars 100
mars <-> jupiter 100
venus <-> mars 100
```

Between `merkur` and `venus` 100% of the packets arrive, as well as between `venus` and `erde` and so on. All not explicitely given connections use the default value. By this we realized a network which looks like the one shown in figure 8.2. Actually not only the values `0` and `100` are possible, but also any value in between. So for example `merkur <-> venus 60` will result in 60 % of the packets between `merkur` and `venus` arriving and 40 % being dropped. The packets which are dropped are randomly chosen. However, while testing we only used rates of 0 % and 100 %.

As the switch internally works with mac addresses, there must be a mapping from hostnames to mac addresses. For that purpose with every hostname there must be declared an ip address, which is mapped by the switch daemon to a mac address the

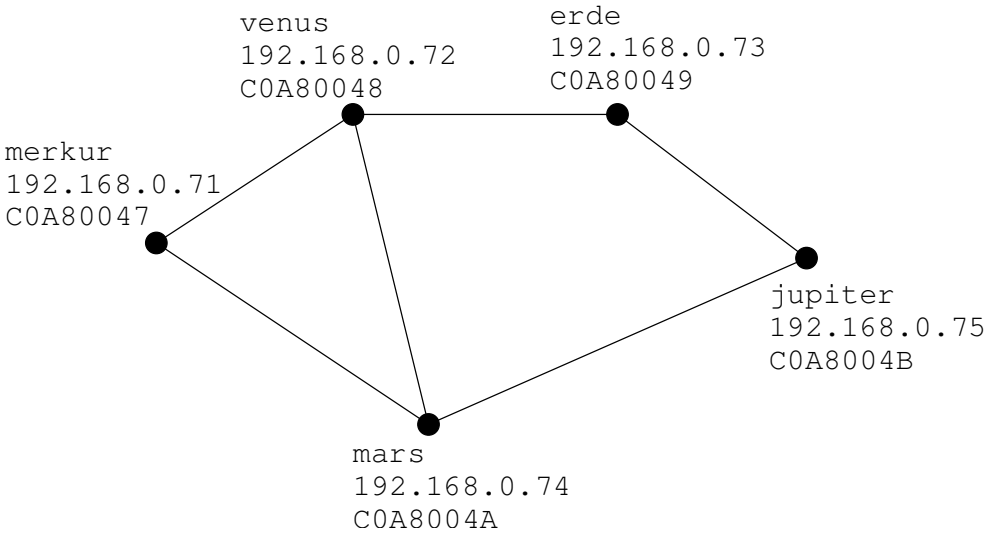


Figure 8.2.: example UML network

same way as the UML computers do. If that is not sufficient, one can declare the mac address directly. An example:

```
merkur:  192.168.0.71
venus:   192.168.0.72
jupiter: a0:c1:34:91:2b:f2
saturn:  192.168.0.76
```

If the configuration file is changed, the switch will read it and the changes will immediately become valid.

It is doubtful if the packet dropping performed by the switch daemon sufficiently simulates wireless networks with different connection qualities and transmission rates, but we can at least easily simulate networks in which a computer is not connected with all the other ones. This is very good for testing the principle functionality of our algorithm. A reasonable performance test is not possible with UML anyway.

Scenario editor

For testing our algorithm in UML it will be useful to have a scenario of different arrangements of the UML computers, i.e. different configurations of the switch daemon. The scenario editor helps to build such scenarios. Figure 8.3 shows a screen-shot of

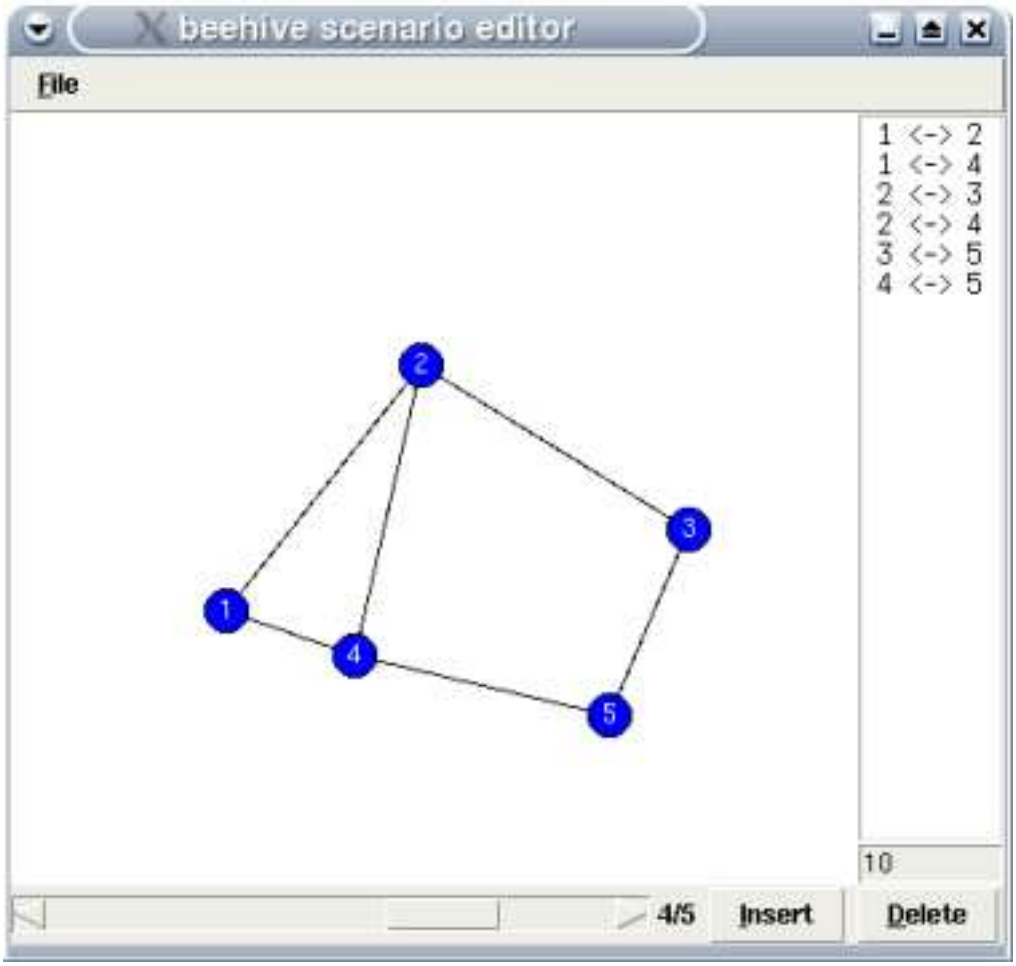


Figure 8.3.: the scenario editor

the editor. With the scrollbar at the bottom of the window one can move within the arrangements, the `Insert` button creates a new arrangement, and the `Delete` button deletes the current one. With the mouse you can move the computers. The lines between them and the table at the right side show which of the computers are connected. The text-field at the bottom right corner specifies the number of seconds this arrangement is in use. With the corresponding commands of the menu one can open and save scenarios. The `Export` command creates a scenario file and the configuration files for the switch daemon. A separate script is developed to read that scenario file and copies in time the configuration files to the `wlan.conf`, which is immediately read by the switch daemon.

Performance and UserModeLinux

For testing in UserModeLinux we have developed a testing environment, in which we start one to nine machines on one hostsystem.

These "virtual" machines have identical settings and on each machine we start a Perl-Script, which starts ftp-transfers at defined times, and measures the response time in the meantime with a parallel ping-command. The results (transfersize, transfertime, responsetime) are written to files on the mounted HostFS. At the end of the run on each machine will be executed a `iptables` command to find out, how many packets are send out and received, and how many scouts are needed.

After the run, we run a script to collect the information from the written files and printed out for a overview.

While doing the tests, we had enormous bad results for a simple static scenario with a couple of transfers at the same time. There are many things, that point on the theory, that the CPU of the hostsystem is to slow:

- In the scenario the load of the CPU is growing to 4 or more
- The UML-Switch can not deliver all packets to their targets
- If each machine is connected to each other, there are no problems
- Packets to destinations far away causes a lot CPU-time, because every hop on the way have to handle them, for a n hop route, a packet needs n-times more CPU-time.

8.2.3. Test-Scripts for UML and real networks

In order to measure the throughput of the network, we use ftp. For our scripts, on the machines a ftp-server (like `wu-ftpd`) and a ftp-client (we use `wget`) are needed. The

starter-script is written in `perl` and needs a few modules for proper working and exact measurement. This startscript, called `starter.pl` parses a scenariofile, in witch are given the points in time, the host, the destination-host are given, and starts at the given time the ftp-transfer.

The ftp-transferscript does two things: it starts a ftp-get from the target-host, and at the same time, it starts a ping-session to the target-host, as long as the ftp-transfer runs. When it has finished, the ping-command stops. The results of the ping-command (responsetime) is written to a file named host.targethost.pinglog, the result of the ftp-transfer (size, duration and targethost) is written to a file host.ftplog.

At the end of a testrun, the number of packets send, the number of scouts, etc, are read out with `iptables -L -t mangle -v -x`. The interesting numbers are parsed of the output and written in another logfile.

With an extern perlscript the logs are read out, the pingtimes are written to an array. Over this array it calculates some results, like mean, geometrical mean, stdandard deviation of the responsetime and it gives out the mean of throughput.

8.3. Results

8.3.1. UserModeLinux

We have run tests in static and in mobile environments simulated with the UserModeLinux-tools. As our test scenario we have created a scenariofile for runtime of 300 seconds. In this time, there are 32 ftp-transfers between the five UML machines. As described above, echo-pings measures the responsetime in parallel in 0.2 sec cycles. Each FTP-transfer has a transfervolume of 1 Megabyte. We have choosen this way of measurement, because it is a realistic test, with all influences of the TCP-features and functions. For the Results we have no comparison with other algorithms, because we found no other functional developments for the linux kernel.

In our static environment the machines are connected as follows:

```
merkur <--> venus <--> erde <--> mars <--> jupiter
```

In the mobile simulation the machines are connected in a cohesive graph and the connections change every 5 seconds.

The results for the static environment looks as follows:

```
scouts sent:           1266
scouts received:      11356
```

```
success rate:       99.7569 %
average ping:         18.09 ms
geometric mean:         8.5 ms
trimmed mean:         14.16
throughput:          480504 bytes/sec
```

In the mobile environment the results are very different:

```
scouts sent:          773.2
scouts received:       5975
intern scouts:          740
success rate:       98.4674 %
average ping:         21.93 ms
geometric mean:         6.5 ms
trimmed mean:         12.37 ms
throughput:         10446.12 bytes/sec
```

One reason, that the throughput of the mobile environment are worse then the of the static are, that the TCP-connections have a resend timeout, if a packet is lost, as described in the TCP-Section of this final report.

8.3.2. Real wireless Networks

Testing with real networks has caused much more problems as testing in UserModeLinux.

- uncontrollable environment, much influences by other electrical devices, weather, etc, as a result an erratic range.
- packet loss and resulting TCP slowdowns
- problems with other ethernet-packets like ARP

For this reasons, the results are not coparable with the UserModeLinux-Results and the NS2 simulation.

But we have re-enacted the scenario above (Fig. 8.1) with the laptops and used the statistical script to get some results. The number of foragers and scouts is not countable, because we need other iptables-rules for the real testing, but its possible to get responsetime and throughput. In order to test if in every case of the scenario a way from each host to each other host is found, we do 12 ftp transfers per run, every host to each other in both directions. The transfervolume is set to 1 Megabyte and pings are done in parallel in 0,2 sec cycles.

The Results:

1.	average ping:	145.53	ms
	geometric mean:	77.68	ms
	throughput:	215805.45	bytes/sec
2.	average ping:	290.80	ms
	geometric mean:	64.34	ms
	throughput:	60908.75	bytes/sec
3.	average ping:	306.21	ms
	geometric mean:	98.67	ms
	throughput:	81052.85	bytes/sec

shows, that the connections are found, and the ping time is acceptable.

Part III.

Energy Efficient Hierarchical Scheduling

9. Introduction

By Kai Moritz (kai.m.moritz@gmx.de) and Rene Zeglin (rene.zeglin@udo.edu)

9.1. Motivation

The main goal of the BEEhive Project–Group is to develop an *energy efficient ad-hoc network architecture*. To reach this goal, two jobs have to be done. First, an energy efficient routing protocol has to be developed. Our approach to that objective has been outlined in the first sections of this report. Second, the single hosts themselves, that now can communicate with each other without wasting too much energy, should consume as little energy as possible while fulfilling their tasks. This part of the report describes our approach to reach that goal.

9.2. Where to Save Energy

Energy can be saved at many points in modern computer architectures. Most often by simply turning off components that are currently not used. So a modern laptop can turn of its display, hard-disk or wireless network card if there is no interaction with the user, the system does not need to fetch any data or there is no network traffic for a certain period of time. It could even completely turn off itself after a certain period of time. The turn-off of the hard-disk, the display or the complete computer nowadays is efficiently managed by the BIOS. But the obvious disadvantage of saving energy by completely turning off the computer is, that a turned-off computer is not very useful. Especially it is not very responsive. Often a computer system has to do some work continuously, for example a small number of periodic tasks, and hence it cannot be turned off. On the other hand, it needs its full computation power only during short periods, so running at full speed all the time would mean wasting a lot of energy during the times when the computer is idle.

There are two solutions for this problem. The first one is to turn off some units of the CPU temporarily while idling. The advantage of this method is that it needs little

modifications to the operating system: it only has to inform the CPU when it is idling. Due to this fact it is already implemented in most present-day computer architectures and operating systems. The second solution is to reduce the clock rate of the system, filling up the idle-time by stretching the queued work.[1] The advantage of this method lies within the option to reduce the voltage level which will increase the energy-savings. As a rule of thumb a system running at halved voltage will consume only a quarter of the energy needed by the system running at maximum voltage level, even taking into account the CPU turn-offs during the idle times. This is because the same number of cycles are executed in both systems, but the system running at halved voltage reduces the energy consumption by reducing the operating voltage.

Modern processors are able to reduce the voltage level together with the clock rate of the processor. As explained in [JRL03], the support of dynamic voltage scaling by today's operating systems is mostly limited to interval-based DVS algorithms. These algorithms adjust the clock frequency to the current load by splitting the time into intervals and setting the frequency of the upcoming interval according to the utilization of the past intervals. These decisions can only be an approximation of the required performance and it cannot be guaranteed that an urgent task finishes its work in due time. Additionally, the behavior of the system may be customized by the definition of rules that prescribe a certain performance if a set of criteria is fulfilled. For example, the system could run at the maximal frequency stage when a certain important and urgent application is executed. However, these rules does not allow for efficient energy savings.

A better strategy is the task-based scheduling which regards the computer's work as tasks with a guaranteed amount of computing time in a certain period of time. This way, the outstanding work up to a certain deadline is always known and the frequency can be set to a level that saves as much as energy as possible while it is still guaranteed that the tasks meet their deadlines. However, this approach is only applicable in the case of reservation based scheduling but a general purpose operating system additionally requires other forms of scheduling for interactive and batch processes.

Hence, our goal is to develop a DVS algorithm that combines both of these strategies and is applicable to general purpose operating systems and to implement it into the Linux kernel. A good DVS algorithm promises noticeable power savings during everyday use of a computer system, because the maximum CPU-frequency is normally only required during short periods, while most of the time the system can run at a low frequency and thus reduce the voltage level. On the other hand dynamic voltage scaling ideally

[1] This two solutions are not exclusive. It is no problem to scale the clock rate and additionally save energy by turning off the CPU during idle-loops if there are still some.

does not slow down the responsiveness of the system, because the CPU-speed will be dynamically increased when needed.

9.3. Our Approach to Dynamic Voltage Scaling in General Purpose Operating Systems

A central problem while designing a DVS algorithm for a general purpose operating system is that the schedulers of such systems are most often fairly complex. As general purpose operating systems are designed to be used in different scenarios, the designers of such systems have only limited knowledge of which programs the system has to run and which of these are the important ones from the user's point of view. Hence, the design of a scheduling algorithm for a general purpose operating systems has to be based on ingenious heuristics, which most often leads to complex dependencies between the scheduling algorithm and various parts of the system.

Because of this, implementing a DVS algorithm for a general purpose operating system is a challenging task. Furthermore, the developed algorithm has to be tied together an operating system, because it has to work closely with the special scheduling algorithm of that operating system to exchange information about process states etc.

These considerations lead to our central design decision: the design and implementation of our DVS Algorithm will be based on *Hierarchical Scheduling* (HS). HS splits up complex, monolithic schedulers in a hierarchical composition of small and easily maintainable schedulers. These schedulers build a tree with a root-scheduler, which receives the overall computing time of the system. Each scheduler in the hierarchy distributes the CPU time that it receives to its children which can be schedulers again or tasks. In such a system the root and the inner nodes of the hierarchy tree are schedulers and the leaves of the hierarchy tree are the tasks which are scheduled by the system.

The combination of HS and DVS brings several advantages. The first advantage of HS in regard to our goal is the reduction of complexity through decomposition. Using HS, complex scheduling behaviors are modeled by a composition of small and simple schedulers. The main idea is that every scheduler in the hierarchy runs its own DVS algorithm and the global DVS decision is reached from these locally computed decisions. This way, the problem to develop a complex DVS algorithm for a multipurpose scheduler is divided into two – hopefully smaller – problems: First, DVS algorithms for some simple schedulers must be developed; and second, a way must be found to reach a reasonable global DVS decision from the locally computed DVS needs. As a side effect of the decomposition it becomes possible to compose the hierarchy of well known schedulers

together with dedicated DVS algorithms from literature. Therefore the first problem vanishes.

Another advantage of HS is that it introduces the possibility to give special tasks guarantees about how much computation time they will receive and when they will get it. This establishes the opportunity to create a DVS algorithm that can lower the operating frequency of a system in order to save energy and, at the same time, ensure that special tasks will receive a guaranteed amount of computation time, so that they will not fail when the frequency is lowered. This is a clear advantage over a monolithic DVS algorithm that is based on heuristics about the scheduling behaviour.

Last but not least, HS makes the scheduling behaviour of the system customizable. Scheduling hierarchies are composed of independent scheduling modules. Thus, a new hierarchy that enforces a different scheduling policy can be created very easily by re-arranging the available schedulers. Furthermore, the implementation of new scheduling algorithms is simplified, because the HS framework provides a simple API for that purpose. That is, end-users become able to tune the scheduling behavior of their system to fit their special needs.

10. A Framework for Hierarchical Scheduling and Voltage Scaling

By Kai Moritz (kai.m.moritz@gmx.de) and Rene Zeglin (rene.zeglin@udo.edu)

10.1. Hierarchical Scheduling

As said above, HS splits up the computation of a scheduling decision in smaller parts. In a system that uses HS the monolithic scheduler is replaced by a collection of smaller and simpler schedulers that are arranged in a scheduling hierarchy. The available computation time is distributed from the root of this hierarchy to its leaves by means of the individual schedulers in the hierarchy. Each scheduler in the hierarchy simply takes the incoming computation time and distributes it to its children. That is, every time a scheduler is provided with computation time (generally speaking, every time it is scheduled), it just schedules one of its children. This modular design has several advantages:

- the complexity of the scheduling decision is reduced
- the single schedulers can be kept very simple because complex scheduling policies can be achieved through clever combinations of this modules.
- additional schedulers can simply be integrated
- the scheduling policy becomes customizable because it is much easier to build up a new hierarchy, than to replace a monolithic scheduler, which is usually deeply embedded in the operating system

10.2. Enforcement of Scheduling Policies Through Modularized Schedulers

The schedulers in a hierarchy are designed as independent and reusable modules. They are autonomous, that is they do not have to know anything about the state of the

scheduling hierarchy as a whole to do their job. Also they do not have to know anything about the scheduler that provides them CPU time nor do they have to know if their children are schedulers or tasks. They just take the granted CPU time and distribute it to their children according to their local scheduling policy.

But although a single scheduler does not have to take into account the scheduling policies of the other schedulers in order to compute its scheduling sequence, it cannot fulfil its job independently from the other schedulers in the hierarchy. More precisely, a single scheduler can compute its scheduling sequence independently from the other schedulers, but the validity of a scheduling policy assured to a specific task relies upon a correct scheduling hierarchy. The guarantee that a task at a leave of the hierarchy receives depends not only on the scheduling policy of its direct parent scheduler, but on the scheduling policies of all schedulers on the path from the task up to the root of the scheduling hierarchy. Hence, not every combination of schedulers leads to a useful hierarchy. For example a real time scheduler cannot guarantee anything to its children if it is scheduled by a time sharing scheduler.

Whereas a scheduler does not have to know anything about its parent to calculate a scheduling order, the validity of the scheduling policies which are meant to be provided by a scheduling hierarchy must be proved before the hierarchy is set in operation. To solve this problem Regehr has developed a theory for proving that a given scheduler hierarchy is able to fulfill the scheduling requirements of a set of applications.

10.3. Validating Scheduling Hierarchies

10.3.1. Describing Scheduling Policies through Guarantees

In order to judge the validity of a scheduling hierarchy the scheduling requirements of the applications are specified in form of scheduling guarantees. A guarantee is a formal statement describing the allocation and distribution of CPU time. The requirements that are needed by a certain scheduler and the scheduling policy which it enforces, both can be described as guarantees. So, a certain scheduler can be satisfactorily described by an incoming guarantee, which it needs to work correctly, and a set of outgoing guarantees, which it provides. That is, by means of guarantees one can describe a scheduler while abstracting from a certain algorithm: in terms of Regehr's theory of guarantees a scheduler can simply be seen as a guarantee converter. The purpose of a scheduling hierarchy then becomes to convert the ALL guarantee, which represents the overall CPU time, into a set of guarantees matching the requirements of the applications. Thus, the question if a scheduling hierarchy provides a certain set of scheduling policies

can be solved by verifying that all nodes in the hierarchy tree receive sufficient guarantees.

In [Reg01] Regehr defines a set of guarantees and shows which schedulers match which guarantee conversion. Furthermore he claims some rules for direct guarantee conversions through rewrite rules and substantiates his claims with formal proofs. As some of the later augmentations are based on the knowledge of that theory, we will briefly describe some of its fundamentals. For more details about guarantees and guarantee conversions we refer to [PGc, PGa, Reg01].

10.3.2. Guarantee Types

As first step, some basic types of guarantees will be described.

The ALL Guarantee represents the assignment of 100% of the available CPU time. It is given to the root scheduler of a hierarchy by the operating system. Obviously it is an acceptable incoming guarantee for any scheduler.

The NULL Guarantee states, that there can't be made any guarantees about the amount of CPU time that will be provided. A normal general–purpose scheduler (like the one implemented in Linux or Windows) can only promise this guarantee.

CPU Reservation Guarantees (RES) describes soft real time scheduling behavior. A RES guarantee ensures that a specific *amount* of CPU time is provided during each *period*. Thus *RES* guarantees are useful for applications that will fail if they receive less processing time than they require. The basic *RES* guarantees are constructed by combining the five properties *basic*, *continuous*, *hard*, *soft* and *probabilistic*:[1]

- **Basic** CPU Reservations ensure that the promised amount of CPU time is provided during each period. They don't make any statement if it is provided at the beginning of a period or at the end. So the scheduler could arbitrary arrange CPU time within a period.
- **Continuous** CPU Reservations ensure that *every* arbitrary period–sized time interval will contain the reserved amount of CPU time. Just recall that a basic reservation scheduler is allowed to provide the promised CPU time at the beginning of one period and at the end of the next. Obviously the time between this two schedules is longer than one period. Claiming a continuous reservation, this case is forbidden.

[1] It is essential to not confound the terms *hard* and *soft* with the corresponding real–time terms.

- **Hard** CPU Reservations provide exactly the requested amount of CPU time. No extra time will be given to the process.
- **Soft** CPU Reservations may receive extra CPU time in addition to the guaranteed reservation.
- **Probabilistic** CPU Reservations are a special case of soft CPU reservations. A specified minimum execution rate and granularity is guaranteed to the scheduled objects. In addition they have the chance to get extra CPU time from a shared *overrun partition* on a probabilistic basis.

Regehr constructs the following RES guarantees from these properties.

- $RESBS\ x, y$ denotes a basic soft CPU reservation with amount x and period y.
- $RESBH\ x, y$ denotes a basic hard CPU reservation with amount x and period y.
- $RESCS\ x, y$ denotes a continuous soft CPU reservation with amount x and period y.
- $RESCH\ x, y$ denotes a continuous hard CPU reservation with amount x and period y.
- $RESPS\ x, y, z$ denotes a probabilistic soft CPU reservation guarantee with amount x and period y. z is the size of the *overrun partition* from that scheduled objects can receive extra CPU time on a problematic basis.

Proportional Share Guarantees can be divided into two classes:

- **Proportional Share Guarantees with Bounded Error (PSBE)** have the syntax $PSBE\ s, \delta$. s denotes the share of the CPU time promised to the receiving object. s is specified as an absolute procentual value ($n \in [0..1]$), because this simplifies localized analysis of a hierarchy. δ is a constraint on the error–term in the provision of CPU time. For any time t the schedulable object is guaranteed to receive at least a $s\dot{t} - \delta$ share of the CPU time. δ is highly dependent on the used scheduling algorithm and may be difficult to calculate.
- **Weak Proportional Share Guarantees (PS)** are proportional share guarantees where no deterministic bound on the error–term could be made by the scheduling algorithm. They have the syntax $PS\ s$ with $s \in [0..1]$. Like above s is the absolute procentual value of the promised CPU share. However, since

no bound on the error–term can be held the provided share is only an approximation of the promised one. Hence PS is a much weaker guarantee than $PSBE$.

The set of basic guarantees presented above is by no means complete. For example Regehr additionally names the *non–preemptive CPU reservation* ($RESNH$), the *synchronized CPU reservation* ($RESSH$) and the *uniformly slower processor* ($RESU$), that are all special kinds of the CPU reservation guarantee [Reg01, p. 49].

10.3.3. Guarantee Conversion through Schedulers

Like said above, from the point of view of the guarantee formalization a scheduler takes a guarantee of one type and converts it into a set of (other) guarantees. (This was exactly the reason why the guarantee formalization was introduced.) Table 10.1 shows some conversions that can be achieved through well known scheduling algorithms. The fact that a scheduler converts a guarantee of type A into a set of guarantees of type B is noted as: $A \mapsto B^+$, where A is the weakest acceptable incoming guarantee for the scheduler to perform the conversion. *any* is a placeholder for an arbitrary guarantee. Exemplary discussion of two of the conversions noted above:

Scheduling Algorithm	**Guarantee Conversion**
Fixed Priority	$any \mapsto (any, NULL^+)$
Proportional Share	$PS \mapsto PS^+$, $PSBE \mapsto PSBE^+$,
EEVDF	$ALL \mapsto PSBE^+$,
Basic CPU Reservation	$ALL \mapsto RESBS^+$
Probabilistic CPU Reservation	$ALL \mapsto RESPS^+$
Round Robin	$NULL \mapsto NULL^+$
Linux	$NULL \mapsto NULL^+$

Table 10.1.: Conversions that can be achieved through well known scheduling algorithms

Fixed Priority: A preemptive fixed priority scheduler gives no guarantee of its own, rather it passes what ever guarantee it receives to its highest priority child. What amount of CPU time the other children will receive can not be predicted, because it depends on the CPU usage of the highest priority child. Hence the other children only receive the NULL guarantee. Because of that a fixed priority scheduler can

accept any type of guarantee as incoming one, though an incoming guarantee of type NULL will not make any sense.

Time Sharing: A time sharing scheduler does not make any guarantees to its children. Hence it can accept any type of guarantee as incoming guarantee.

10.3.4. Direct Guarantee Conversions through Rewrite Rules

In [Reg01] John Regehr has also shown that it is possible to convert certain guarantees into others by rewrite rules. A rewrite rule simply interprets a given guarantee as another. However, this is not possible for all guarantee pairs, as they must be semantically similar. As rewrite rules are only interpreting the incoming guarantee as another one, they do not really change it. Thus, rewrite rules cannot be used to enforce a certain scheduling policy. Their purpose is to ease the combination of schedulers.

ALL	√	√	–	√	√	√	√
RESBS	–	√	√ (10.1)	√	√ (10.3)	√ (10.2)	√
RESPS	–	√	√ (10.1)	√	√ (10.3)	√ (10.2)	√
PSBE	–	√ (10.4)	√ (10.4)	√	√	√	√
PS	–	–	–	–	–	√	√
NULL	–	–	–	–	–	–	√
↦	*ALL*	*RESBS*	*RESCS*	*RESPS*	*PSBE*	*PS*	*NULL*

Table 10.2.: Direct guarantee conversions by means of rewrite rules (deviated from [Reg01, p. 56])

Table 10.2 shows the possible direct conversions. In general guarantees can only be converted into weaker ones or at most in equally powerful ones. The non-trivial rewrite rules stated in the table are listed below. For their proof please refer to [Reg01].

Theorem 10.1. *The guarantees $RESBS\ x, y$ and $RESBH\ x, y$ can each be converted into the guarantee $RESCS\ x, (2y - x + c)$ for any $c \geq 0$.*

Theorem 10.2. *Any CPU reservation with amount x and period y may be converted into the guarantee $PS\ \frac{x}{y}$.*

Theorem 10.3. *The guarantees $RESBH\ x, y$ or $RESBS\ x, y$ may be converted to the guarantee $PSBE\ \frac{x}{y}, 2\frac{x}{y}(y - x)$.*

Theorem 10.4. *The guarantee $PSBE\ s, \delta$ can be converted into the guarantee $RESCS$ $(ys - \delta), y$ or $RESBS$ $(ys - \delta), y$ for any $y \geq \frac{\delta}{s}$.*

10.3.5. Prerequisites for Using Guarantees

In order to use the rules for guarantee conversion described above to validate scheduling hierarchies, three simple assumptions must hold.

- The requirements of the applications are known and can be expressed in form of guarantees.
- All schedulers that are used in a given hierarchy are implemented correctly and are proved to provide the agreed outgoing guarantees under the condition that they are given the necessary incoming guarantee.
- The scheduling scenario, i.e. the set of applications and associated guarantees, is static.

The first two assumptions can be regarded as given. The validity of the third assumption is not that obvious. In a normal operating systems the set of applications that has to be scheduled by the system is frequently changing and cannot be foreseen. So, in order to be able to regard a scheduling scenario as static Regehr differentiates between long-term and short-term decisions. Short-term decisions are made by a scheduling algorithm in millisecond-granularity to enforce its scheduling policy. They do not affect the scheduling scenario. The fact which applications are running and which guarantees are provided to them is considered as a long-term decision. A change to these long-term decisions corresponds to a transition from one scheduling scenario to another one. In-between two long-term decisions, a scheduling scenario can be regarded as static and thus, be validated using Regehr's Guarantee System.

10.4. Hierarchical Computed Dynamic Voltage Scaling Decisions

So far, the basic approach and theoretical fundamentals of HS have been explained. Now we have to show how HS can be utilized to compute a DVS decision.

The central advantage of HS in connection with our plan to implement a DVS algorithm is the decomposition mentioned above, which decreases complexity. The goal is to use the modularity of HS for the design of a DVS algorithm. The main idea is, that the voltage scaling decisions are made locally by the schedulers in the hierarchy. In a scheduling hierarchy each scheduler knows about its incoming and outgoing guarantees. That is, it can perform a local DVS algorithm to decide the percentage of its incoming guarantee that is actually needed. Naturally, a single scheduler cannot decide if it

makes sense to change the CPU frequency of the overall system because in accordance with the principle of modularization each scheduler has only access to a limited set of information. Hence, the local scaling decisions made by the particular schedulers have to be assembled to make up a global voltage scaling decision.

To assemble the global voltage scaling decision we have decided to transfer locally computed scaling information to the parent scheduler so that they cumulate at the root of the scheduling hierarchy in the end. If every local DVS algorithm considers the DVS decisions of its child schedulers all local DVS decisions of a whole sub tree are automatically assembled in the local DVS decision at the root of this sub tree. Obviously the local DVS decision at the root of the hierarchy assembles all locally made DVS decisions and can be used to adjust the frequency of the system accordingly.

10.5. Extensions to Regehr's Theory of Guarantees

Although the correctness of the DVS decision that is computed by the hierarchical algorithm described in the previous section seems to be obvious, some extensions have to be made to the theory of guarantees proposed by Regehr in order to prove this correctness.

As said in section 10.3 three assumptions must be validated, before Regehr's theory of guarantees can be used to judge about hierarchies. The third assumption, which says that the hierarchy under examination has to be static, is affected by our hierarchical DVS algorithm, because a consequence of our extension of hierarchical scheduling by a DVS algorithm is a new dynamic component that is added to the scheduling system. The DVS algorithm may modify a scheduling scenario at an arbitrary time by adjusting the incoming guarantee of a scheduler to the current load. Hence, the assumption that the scheduling scenario is static between two long-term decisions is violated.

In order to be able to use Regehr's guarantee system to verify that the decisions made by the DVS algorithm does not perish the guarantees the hierarchy is providing to the scheduled tasks, this new dynamic component must be considered. In [Reg01] Regehr mentioned the following three events as long-term decisions, which lead to a transition between two scheduling scenarios:

- a process forks,
- a process exits and
- a process requests another guarantee.

By defining the decisions that are computed by our hierarchical DVS algorithm as an additionally long-term decisions, the dynamic changes to the scheduling hierarchy which were introduced by our hierarchical DVS algorithm are reduced to a sequence of static scheduling scenarios. This way, the decisions made by our algorithm can be proved as correct if each of that static scheduling scenarios can be proved correct in the sense of the guarantee system.

Thus, by this extension it becomes feasible to validate a hierarchical DVS scheduler if the following assumption holds in addition to the prerequisites mentioned in section 10.3:

- the decisions made by the DVS algorithms are correct, so that the schedulers are able to provide the agreed guarantees when they are given the adjusted incoming guarantee.

10.6. Composing Valid Hierarchical DVS Algorithms

In order to acquire a valid hierarchical DVS algorithm some further restrictions regarding the combination of the DVS algorithms used by the individual schedulers must be considered while the scheduler hierarchy is composed. In this section we will explain these additional restrictions and propose some rules to cope with them.

10.6.1. Reclamation of Returned Computation Time is Forbidden

A typical reservation based DVS algorithms tracks unused computation time and exploits it to reduce the clock frequency of the processor. The algorithm always knows how much work has to be done until a certain deadline so that the frequency can be set as minimal as possible without violating the reservations that are provided to the tasks. The outstanding work can be calculated from the reservations that are provided to the tasks. Additionally, the tasks indicate when their work for the current period is done so that the outstanding work up to the task's deadline can be reduced by the portion of the task's reservation that has not been consumed.

The central assumption is, that a task will not expect any more computation time within the actual period after it has signaled to the scheduler, that its work has been done. So, if tasks hand back their remaining reservations because they have finished their job prematurely, they must be aware that they cannot claim the released computation time back, because the DVS algorithm may have consumed it by lowering the CPU frequency. However, in a system that uses HS a scheduler can also schedule other

schedulers which are running their own DVS algorithm. In this case a reservation is handed back if the DVS algorithm of the child scheduler decides that it does not need its full incoming guarantee at the moment. Just like a normal task, this DVS algorithm has to be aware that it cannot claim the released computation time back within the current reservation period. Unfortunatley, not every DVS algorithm complies with this assumption.

Typical interval-based DVS algorithms for general purpose operating systems are able to fulfill this requirement. Information about future needs of processes like deadlines or periods is generally not available in a general purpose operating system. Thus, a DVS algorithm cannot be based on slack time estimations like a real time DVS algorithm. To adjust the clock frequency to the current load, DVS algorithms for general purpose operating systems usually split up time into intervals and regard the utilization of the past intervals as a prediction for the upcoming interval. If such a DVS algorithm becomes the child of a real time DVS algorithm in a scheduling hierarchy, the length and positioning of the intervals can be aligned to the internal reservation period of the parent scheduler. The positioning can be done, for example, by making the parent scheduler send a signal to the DVS algorithms of its children at the beginning of a period. This way, the requests to adjust the incoming guarantee are made exclusively at the beginning of a period and are not changed and particularly not incremented within a period.

But for example real time DVS algorithms require that the clock frequency can be changed immediately and all the time. If such an algorithm runs as the child of a reservation based DVS algorithm it may happen that it tries to increase its incoming guarantee shortly after reducing it so that both events occure within the same scheduling period. Therefore, it must be ensured that schedulers which have to change their needs at an arbitrary time are given the ALL guarantee.[2] The ALL guarantee ensures to its receiver that it is scheduled immediately every time it wants to and is never interrupted. A scheduler that provides the ALL guarantee (like for example a *preemptive fixed priority* scheduler, which provides its incoming guarantee to the child with the highest priority and therefore can forward the ALL guarantee) must ensure that the child that receives the ALL guarantee is scheduled immediately every time it becomes runable. Hence, it must not assign a period to that child. This allows the DVS alrogithm which is associated with the scheduler that receives the ALL guarantee to change its DVS decision at any arbritary time. Claiming the ALL guarantee for reservation based schedulers looks like

[2] Another guarantee that would allow its receiver to change its DVS at any arbitrary time is the $RESU$ guarantee. $RESU$ stands for REServation Uniformly Slower Processor. That is, receiving the $RESU$ guarantee with a share of 50% is the same as receiving the ALL guarantee on a system that is only half as fast.

a hard restriction. But it is not that worse, because reservation based schedulers (like for example the *earliest deadline first* scheduler) require the ALL guarantee as their incoming guarantee anyway.

10.6.2. Adjusting Guarantees

Once a DVS decision has been calculated by a scheduler, it has to be transferd up to the parent scheduler. Since the DVS decisions are calculated in percent of the incoming guarantee we decided to hand up that value and let it up to the parent scheduler to adjust the single parameters of the associated guarantee accordingly. So far, we assumed that the adjustment of the guarantee is realized by the parent scheduler simply by adjusting the amount of computing time that is provided within one period whereas the period itself is not changed. This assumption is adequate, because all schedulers that provide a meaningful guarantee (that is not the $NULL$ guarantee) must maintain a period to keep track of time and some sort of amount to ensure that their children are provided with the promised guarantees. However, this assumption may be violated if rewrite rules are used to convert incoming guarantees in order to fit the needs of a scheduler.

The adjustment of a converted guarantee is realized by an adjustment of the original guarantee. The problem in doing so is that the time limit up to which the converted guarantee provides a certain amount of computing time may depend on the amount of computing time that is provided by the original guarantee. For example, the guarantees $RESCH\ x, y$ or $RESCS\ x, y$ can be converted to the guarantee $PSBE\ \frac{x}{y}, \frac{x}{y}(y-x)$. That is, a reservation of x time units over a period of y time units can be interpreted as a CPU share of size $\frac{x}{y}$. The error term $\frac{x}{y}(y-x)$ denotes that for any time t the schedulable receives at least $st - \delta = \frac{xt}{y} - \frac{x}{y}(y-x)$ units of computing time. Since it depends on the amount of computing time x that is provied by the original guarantee, the error term changes when the original guarantee is changed. The maximum of $\delta = \frac{y}{4}$ is reached for $x = \frac{y}{2}$ or $s = 0.5$. So, if a scheduler that is given the converted $PSBE\ s_g, \delta$ guarantee reduces its share from $s_g > s_e \geq \frac{1}{2}$ to s_e the reserved computing time is decreased from $x_g = ys_g$ to $x_e = ys_e$ as expected but, in parallel, the bounded error increases to $\frac{x_e}{y}(y - x_e) > \frac{x_g}{y}(y - x_g)$.

A way to circumvent this problem is to weaken the converted guarantee by setting the bounded error to its maximum δ_{max} although the actual value will be more than it when the requested share is set to $s_e \neq min(s_g, 0.5)$.

$$\delta_{max} = \begin{cases} ys_g - ys_g^2 & \text{if } s_g < 0.5 \\ \frac{y}{4} & \text{if } s_g \geq 0.5 \end{cases}$$

This way, the bounded error stays constant when the reserved amount of computing time x_e is modified. However, in the case of $s_g > 0.5 \wedge s_e \neq 0.5$ more computing time than actually requested is reserved for the schedulable.

With some conversions the upper bound of the length of the period within which the agreed computing time is provided is not as good as $\frac{y}{4}$ but practically useless. For example, the guarantee $PSBE\ s, \delta$ can, for any $y \geq \frac{\delta}{s}$, be converted into the guarantee $RESCS\ (ys - \delta), y$ or $RESBS\ (ys - \delta), y$. In this case, the length of the reservation period y is inversely proportional to the reserved CPU share s and increases rapidly for small values.

10.7. Validating the Assembled Dynamic Voltage Scaling Decisions

As said in section 10.5, the computation of a DVS decision has to be defined as an additional long-term decision in order to be able to judge about its validity. That is, each DVS decision leads to a new scheduling scenario whose validity in terms of the guarantees system has to be proved, before it is set into operation. Fortunately, this is not a necessity in practise, as can be seen by the following argumentation.

A scheduler adjusts its incoming guarantee by adjusting the guarantee's parameters, i.e. the type of the guarantee is not modified. The parameters can be set at most to the settings of the primary incoming guarantee, that is the adjusted guarantee cannot exceed the primarily agreed one. Therefore, a parent scheduler is always able to provide the adjusted guarantees to it children, because they are just relaxations of the primary agreed one. Thus, it is sufficient to prove that a scheduler is able to provide the required outgoing guarantees after its incoming guarantee has been adjusted to the current utilization. However, the correctness of these DVS decisions of the individual schedulers follows from the correctness of the implemented DVS algorithms and the assumption that the restrictions mentioned in the previous section hold.

This way, the validity of the whole resulting scheduling hierarchy can be recursively proved from the leafs to the root. Since the local DVS decision of the root scheduler is equal to the global decision of the scheduling hierarchy, the new scheduling scenario is valid and the calculated frequency will suffice to provide the guarantees required by the schedulers and processes at the leafs.

11. Our Implementation of a Hierarchical Scheduling and Voltage Scaling Framework

By Kai Moritz (kai.m.moritz@gmx.de) and Rene Zeglin (rene.zeglin@udo.edu)

11.1. The Scheduling Framework

11.1.1. Data Structures

The two basic data types of our framework are the scheduler and the schedulable structures. A scheduler distributes computing time among its children according to the implemented scheduling policy. A child of a scheduler is a schedulable and represents either a process or a scheduler since in HS both of them can be scheduled. Therefore, processes as well as schedulers are each connected to a schedulable structure. When a new process is forked or a scheduler is created an appropriate schedulable is constructed and connected to the process or scheduler respectively. It is destroyed when a scheduler or the task structure of a process is released.

The schedulable data structure (listing 11.1) stores the pointers `sched` and `task` to a scheduler and a process. Since a schedulable represents either a process or a scheduler one of the two pointers is a $NULL$-pointer. So that a process or scheduler is able to receive computing time, the associated schedulable must be connected to a scheduler. It is connected to exactly one scheduler which is called its parent scheduler and is referenced by the `parent` pointer. Scheduler specific data of a schedulable, e.g. a `list_head` structure that is queued in the runqueue of the parent scheduler, and parameters, e.g. the priority in the case of a fixed-priority scheduler, are stored in the `sched_data` and `sched_param` arrays. Although there are schedulers that do not use timeslices we decided to integrate `time_slice` and `first_time_slice` into the schedulable structure for reasons of simplicity. Instead of maintaining them in nearly all schedulers it is less costly to assign this job to the framework. The pointer `progname` points to the absolute path

of the executable in the file system if the schedulable represents a process. It determines the scheduler the process will be connected to.

```
struct schedulable {
    scheduler_t *parent;

    unsigned int status;

    unsigned int time_slice, first_time_slice;

    /* scheduler specific data */
    int sched_data[SCHED_DATA_SIZE];

    /* scheduler specific parameters */
    int sched_param[SCHED_PARAM_SIZE];

    char *progname;

    scheduler_t *sched;
    task_t *task;

    struct schedulable *next;
    struct schedulable *prev;
};
```

Listing 11.1: Schedulable structure

The modifications of the process data structure consist only of an added pointer `this_schedulable` to a schedulable and the removal of the timeslice elements that have been moved to the schedulable data structure (listing 11.2).

```
struct task_struct {
...
#ifdef CONFIG_SAADI
        schedulable_t *this_schedulable;
#endif
...
#ifndef CONFIG_SAADI
        unsigned int time_slice, first_time_slice;
#endif
...
```

```
}
```

Listing 11.2: Modifications of the process descriptor

A scheduler (listing 11.3) references its schedulable by the pointer `this_schedulable`, just like a process. By means of the function interface which is described in 11.2 the framework accesses the actual implementation of the scheduling algorithm. The `rq_data` array stores scheduler specific data, e.g. the `list_head` of a runqueue. So that the framework knows whether a scheduler is runnable or idle it maintains the number of runnable schedulables in `nr_running`. By means of the ioctl-like system call `saadi_schedctl()` a process can send messages to its scheduler. The system call is forwarded to the function `schedctl()`. In order that potential arguments can be copied from user-space to kernel-space the size of the parameters is stored in the array `schedctl_param_length`.

```
struct scheduler {
    unsigned int id;
    char *name;
    hierarchy_root_t *hr;
    struct list_head sched_list_entry;

    unsigned int flags;

    /* number of runnable schedulables */
    int nr_running;

    /* scheduler specific functions */
    link_to_func_t link_to;
    unlink_with_func_t unlink_with;
    link_func_t link;
    unlink_func_t unlink;
    join_func_t join;
    leave_func_t leave;
    scheduler_tick_func_t scheduler_tick;
    dispatch_func_t dispatch;
    yield_func_t yield;
    destructor_func_t destructor;
    schedctl_func_t schedctl;
    schedmsg_func_t schedmsg;
    set_minfreq_func_t set_minfreq;

    /* lengths of schedctl parameters */
    long schedctl_param_length[SCHEDCTL_PARAM_LENGTHS];
```

```
    /* scheduler specific data */
    int rq_data[RQ_DATA_SIZE];

    scheduler_t *childs;
    scheduler_t *sibling;

    schedulable_t *this_schedulable;
};
```

Listing 11.3: Scheduler structure

The interrelationship between the three data structures is depicted in figure 11.1.

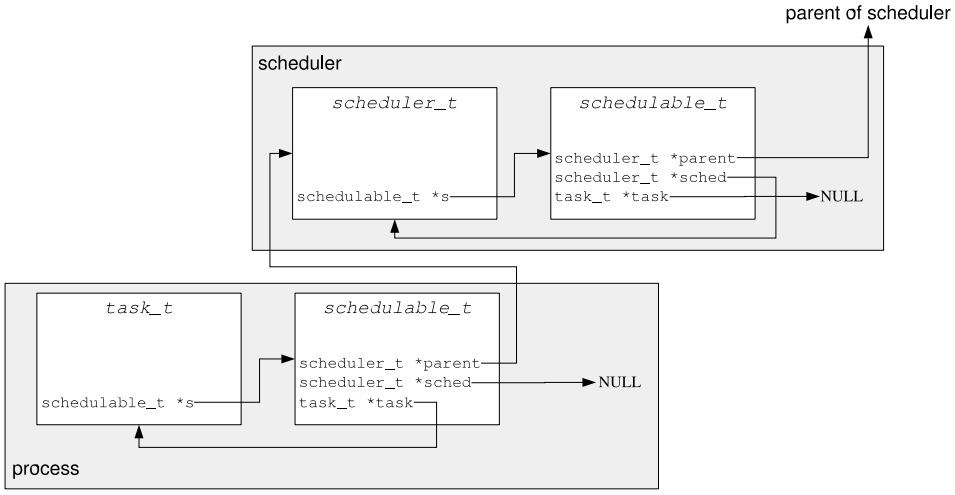


Figure 11.1.: Task, Scheduler and Schedulable Structures

The basic data structure in the Linux scheduler is the runqueue which maintains the list of runnable processes. We replaced the runqueue structure by a `hierarchy_root` structure (listing 11.4) that references the root scheduler `root_sched` of a hierarchy. By defining the name of the runqueue structure in Linux as an alias for our structure many of the macros and scheduling functions used in the official kernel, e.g. the code for runqueue locking, could be used without any modifications. Therefore, the hierarchy root structure includes, apart from the SAADI elements, all elements of the official runqueue structure that may be accessed by the reused code. The pointer `root_parent_dummy` and the frequency adjustment are described in 11.3. The last four elements store mappings (and a reference to mappings) from processes or kernel threads to schedulers of the hierarchy.

The mapping determines the scheduler to which a newly forked process or a process thats begins executing a new executable is connected to (see 11.1.5).

```
struct hierarchy_root {
    /* global scheduling hierarchy lock */
    spinlock_t lock;

    unsigned long nr_running;

    unsigned long long nr_switches;
    unsigned long expired_timestamp, nr_uninterruptible;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    atomic_t nr_iowait;

    scheduler_t *root_sched;

    scheduler_t *root_parent_dummy;
    struct freq_adjust_struct *freq_adjust;

    struct list_head *task_mapping;
    struct saadi_asm default_mapping;
    struct saadi_asm kthread_mapping;
    struct list_head all_schedulers;
};
```

Listing 11.4: Hierarchy root structure

11.1.2. Selection of the next-to-run Process

When kernel code wants to sleep or a process is to be preempted another runnnable process must be selected to run as its successor. To select the next-to-run process the function `schedule()` in `kernel/sched.c` is called.

In Linux, the process on the runqueue with the highest priority is chosen whereas the selection in HS depends on the scheduling decisions of the hierarchy schedulers.

Each scheduler in the hierarchy provides a dispatch function, which selects and returns the schedulable object from the maintained ones that should be running next according to the local scheduling policy. If the returned schedulable is connected to a scheduler, the scheduler hierarchy is descended by one level and the referenced scheduler is queried

to select a schedulable object. This process continues until a schedulable is returned that is connected to a process which is finally allocated the CPU.

If the root scheduler returns a $NULL$ pointer there is no runnable process and and the idle task, which is always runnable, is dispatched to consume the superfluous computing time.

It is not necessary to start the selection process at the root scheduler everytime. If no change in the local scheduling situation of a scheduler has occurred, the previously made decision is still valid. If the scheduler would be queried again, it would make the same decision as before because the set of runnable processes has not changed. Since the framework is notified by the hierarchy schedulers when a change in their local scheduling situation takes place, the selection process can be started at the desired scheduler.

From the set of schedulers with modifications in the local scheduling situation the upmost one belonging to the path from the current process to the root of the hierarchy is selected. The selection process starts at the upmost scheduler because, in HS, the decision of a superior scheduler precedes the decision of an inferior one.

The `schedule()` function can be called explicitly by kernel code to yield the CPU to other processes. In addition to this, preemptive scheduling requires that it is also called when a process runs out of timeslice or when a process with a higher priority than the currently running one wakes up. Therefore, the `need_resched` flag can be set to signal a necessary rescheduling to the kernel. The flag is checked upon returning from a system call or interrupt handling and induces a rescheduling if it is set. In SAADI, the flag can be set in `saadi_join()` (11.1.3), which is called when a process wakes up, and in the `scheduler_tick()` functions of the schedulers (11.1.4), which are called in the course of a timer interrupt.

11.1.3. Sleeping and Waking Up

Generally speaking, a process that waits for an event to occur sets its state from running to sleeping and removes itself from the runqueue before the scheduler selects another process to run. This way, the scheduler cannot select a process that does not want to run. In Linux, a task structure is removed from the runqueue by calling `deactivate()`.

We replaced this function by `saadi_leave()` which removes a schedulable from the runqueue of its parent scheduler. If the last runnable schedulable of a scheduler is removed from the runqueue the scheduler itself is not runnable any more and must be removed from the runqueue of its parent to prevent it from being selected to run. Therefore, the framework maintains the number of runnable schedulables for each scheduler. If it reaches 0 the schedulable connected to the scheduler is removed from the runqueue

of the parent scheduler. This process continues until either a scheduler with further runnable schedulables or the root scheduler is reached.

Waking up is done by the function `try_to_wake_up()` which sets the state of the process to running and appends the process to the runqueue by calling `activate()`. If the priority of the woken up process is higher than the priority of the currently running one a rescheduling is necessary which is indicated by setting the `need_resched` flag.

Our framework replaces `activate()` by `saadi_join()` which enqueues a schedulable to the runqueue of its parent scheduler. If the scheduler has been idle, i.e. there have not been any runnable schedulables on its runqueue, it must be appended to the runqueue of its own parent. This process continues until a scheduler is reached that is already queued in the runqueue of its parent and terminates because the root scheduler is always queued.

If the last scheduler of this process is currently running and the woken up schedulable has a higher priority than the currently running one according to the local scheduling policy, a rescheduling is necessary and the `need_resched` flag is set. So that the rescheduling begins at this scheduler or above in the hierarchy the `SAADI_NEED_RESCHED` flag is set in the state bitmap of the scheduler.

11.1.4. Aging Schedulables

By means of the system timer the Linux scheduler maintains statistics and activates a rescheduling if a process runs out of timeslice. The system timer generates interrupts at a fixed frequency which are handled by the timer interrupt handler.[1] The interrupt handler then calls `scheduler_tick()` to pass the event on to the scheduler.

As regards HS, each of the schedulers in the hierarchy may internally work with timeslices or may be interested in the periodic signal for other reasons. Therefore, the `scheduler_tick()` function of each scheduler on the path from the currently running process to the root of the hierarchy is called in the course of a timer interrupt.

If a rescheduling is necessary according to the local scheduling policy, the affected scheduler returns a certain constant. Thereupon, the `need_resched` flag is set and the `SAADI_NEED_RESCHED` flag is set in the scheduler's state to indicate a necessary rescheduling.

[1] The frequency of the timer interrupt is defined differently from architecture to architecture. It is usually $1000Hz$ or $100Hz$.

11.1.5. Fork, Exec and Exit

HS requires a mapping of applications or processes to the schedulers of the hierarchy. In SAADI, this is realized by mapping absolute paths of program binaries to schedulers. A mapping consists of the path of the program binary, the target scheduler and scheduling parameters, like e.g. the priority of a process that is mapped to a scheduler using static priorities. A newly forked process is mapped to the scheduler its parent is connected to, since it executes the same executable as its parent. Furthermore, it is assigned the same scheduling parameters as its parent process.

When a process starts to execute another program by calling the `exec()` system call, it may be necessary to move the process from its current scheduler to the target scheduler specified in the mapping of the new program binary. If a reassignment is necessary the schedulable is disconnected from the current scheduler and connected to the new one after the scheduling parameters have been updated.

An exiting process is disconnected from its parent scheduler.

Apart from user processes the scheduler maintains kernel threads. Kernel threads are processes that do not execute user-space programs but exclusively kernel code and do not leave the kernel-space. A special mapping assigns all kernel threads to a single scheduler and stores their scheduling parameters.

11.1.6. System Call Interface

SAADI-aware processes can call scheduler specific functions and set or get the current scheduling parameters by means of two system calls.

If the scheduler of a process implements the `schedctl()` function, a call to `sys_saadi_schedctl()` is forwarded to this function after an optional argument has been copied from user- to kernel-space. Multiple operations can be differentiated by the `cmd` parameter which also determines the size of the parameter data structure.

Scheduling parameters can be updated by calling `sys_saadi_sched_param()`. If scheduling parameters are updated, the schedulable is disconnected from its parent scheduler (unlink) and re-connected (link) after the new scheduling parameters have been set. If `update` is false the current parameters are returned.

```
asmlinkage long
sys_saadi_schedctl(int cmd, pid_t pid, __user void *data_us)

asmlinkage long
sys_saadi_sched_param(int update, pid_t pid, __user void *
```

```
    sched_param_us)
```

Listing 11.5: SAADI System Call Interface

11.2. Scheduler Programming Interface

A scheduler that is to be integrated into the framework defines specific data structures and implements a function interface. The scheduler can be compiled into the kernel or as a loadable kernel module which can be dynamically inserted into the kernel and removed as well when it is not needed any more. The appendix contains the listing of an exemplary implementation of the programming interface.

11.2.1. Data Structures

A scheduler maintains global information and individual data for each of the connected schedulables. Considering a priority scheduler, the global data would, for example, consist of a priority array of linked lists and each of the maintained schedulables would be assigned a list head for insertion into one of the lists. It showed to be advantageous, to additionally define a parameter data structure, that stores individual parameters for each schedulable. As regards the example, each schedulable would be assigned a priority as its scheduling parameter. This way, scheduling parameters may be dynamically changed without touching the data structure which might contain scheduler internal data that must not be modified. Thus, each scheduler defines three specific data structures which store scheduler global data (`rq_data`) and data (`sched_data`) and parameters (`sched_param`) of a single schedulable. These structures are embedded in the scheduler and schedulable data structures.

11.2.2. Function Interface

After a scheduler has been created a constructor function is called to initialize the state of the scheduler, register timers etc. Similarly, the finalize function is called to clean up, e.g. by unregistering timers, before a scheduler is destroyed.

```
/* scheduler constructor */
typedef void (*scheduler_construct_t) (scheduler_t * sched);

/* scheduler destructor */
typedef void (*destructor_func_t) (scheduler_t * sched);
```

The following four functions handle connections and disconnections of schedulables to the scheduler. `link()` is called, when a schedulable is to be connected to the scheduler. While the parameters are set when a schedulable is being linked the scheduler has to initialize the `sched_data` structure in the course of the function call. If applicable, a schedulability test must be accomplished and the schedulable rejected, if accepting it would overload the capacity of the scheduler. Whether the schedulable is accepted or not is indicated by the return value of the function.

`unlink()` is called to disconnect a schedulable from its parent scheduler.

`link_to()` and `unlink_with()` are helper functions that are called when this scheduler is to be connected to a parent scheduler. They can contain specific actions necessary in cases like the Join scheduler [2] but if there is no need for these functions default implementations can be used which just call `link()` or `unlink()` where the actual work is done.

```
/* cause a scheduler to link to a parent-scheduler */
typedef int (*link_to_func_t) (scheduler_t * self, scheduler_t *
    parent);

/* cause a scheduler to unlink from a parent-scheduler */
typedef void (*unlink_with_func_t) (scheduler_t * self, scheduler_t *
     parent);

/*
 * a schedulable object requests to be scheduled
 * by given scheduler, returns success or failure
 */
typedef int (*link_func_t) (scheduler_t * sched, schedulable_t * s);

/* a schedulable object leaves its parent scheduler */
typedef void (*unlink_func_t) (schedulable_t * s);
```

The `join()` and `leave()` functions are called when a schedulable becomes runnable or when it blocks waiting for an event, respectively. Blocked schedulable objects do not take part in the competition for computing time and must be removed from the runqueue so that they are not returned when the scheduler is queried for a schedulable to run. In a preemptive scheduler, a woken up schedulable may invalidate a previously made

[2] A Join scheduler merges the computing time of multiple parents although a scheduler has actually only a single parent. When a Join scheduler is linked to a scheduler it creates a helper scheduler and connects it, as a proxy of itself, to the parent. Internally, the Join scheduler is connected to multiple helper schedulers.

scheduling decision. This happens if the schedulable has a higher priority than all other runnable schedulables. By returning the constant `SAADI_NEED_RESCHED` the scheduler can indicate that a change in its scheduling situation occurred so that the framework will perform a rescheduling if it is necessary.

```
/* a schedulable object joins the runqueue of its scheduler */
typedef int (*join_func_t) (schedulable_t * s);

/* a schedulable object leaves the runqueue of its scheduler */
typedef void (*leave_func_t) (schedulable_t * s);
```

The `dispatch()` function is called when the scheduler is to select the runnable schedulable from the maintained ones that is to run next. If there are no runnable schedulables this is indicated by returning a *NULL* pointer.

```
/* dispatch next schedulable */
typedef schedulable_t *(*dispatch_func_t) (scheduler_t * sched);
```

When the scheduler is currently running, i.e. it lies on the path from the currently running process to the root of the hierarchy, the function `scheduler_tick()` is called in the course of the timer interrupt. A preemptive scheduler can signal the necessity of a rescheduling by returning the constant `SAADI_NEED_RESCHED`.

```
/* process a scheduler tick */
typedef int (*scheduler_tick_func_t) (schedulable_t * s);
```

The task of the remaining functions is to receive and handle messages from processes and other schedulers.

The `sys_sched_yield()` system call is a method for a process to voluntarily release the CPU so that other processes get the chance to run. The system call is forwarded to the `yield()` function of the process's parent scheduler.

Scheduler specific messages can be sent using the `saadi_schedctl()` system call which copies a potential argument from user- to kernel-space and forwards the request to the `schedctl()` function of the parent scheduler of the process.

Messages between schedulers are exchanged by calling the `schedmsg()` function of the recipient, whereby there is a special purpose function for exchanging DVS related information. When the DVS algorithm of a scheduler detects that it needs to adjust the portion of its incoming guarantee it communicates the required value to the parent scheduler by means of `set_minfreq()`.

Messages to processes are sent using the asynchronous signal mechanism.

```
/* yield function */
```

```
typedef void (*yield_func_t) (schedulable_t * s);

/* scheduler control */
typedef void (*schedctl_func_t) (int cmd, schedulable_t * s,
                                 __user void *data);

/* scheduler messages (hierarchy internal) */
typedef void (*schedmsg_func_t) (scheduler_t * sender,
                                 scheduler_t * receiver, int msg,
                                 void *data);

typedef void (*set_minfreq_func_t) (scheduler_t * child, unsigned int
    p);
```

11.3. Implementation Details

The scheduler hierarchy can be modified at runtime by registering hierarchies with the framework. Each hierarchy (and an associated mapping of processes to schedulers) is registered under a unique name and the active hierarchy can be chosen by writing the name of a hierarchy into a file[3] in the virtual proc-file system. Hierarchies can be compiled into the kernel or loaded at runtime. Since a hierarchy depends on its schedulers, the corresponding modules must be compiled or loaded into the kernel before the hierarchy.

We make use of CPUFreq to set the clock frequency of the processor. CPUFreq is a modular driver which provides a standard architecture independent way to set the clock frequency of the CPUs in the system[Bro]. The actual frequency transition is done by architecture drivers which are available for a variety of platforms. Requests for frequency changes are accepted by so called governors which forward the requests to the CPUFreq core according to an implemented policy. We created a governor as an interface between the scheduler and CPUFreq which just forwards our requests to the CPUFreq core. Implementation details of our governor are described in [PGb].

As a consequence of the decision to use CPUFreq for frequency scaling the hDVS scheduler is applicable on every platform with an existing CPUFreq architecture driver.

CPUFreq has been designed to be called in process context, because the relevant functions may sleep. Therefore, we created a kernel thread that is woken up using a wait queue when a change of the clock frequency is required and accomplishes the

[3] /proc/hs/active

requests by means of our governor. A work queue could not be used in this case because of serialization reasons. The DVS decisions that activate a frequency transition are made in interrupt context while the scheduler spinlock is held and the ordinary code path to wake up a process would occupy this lock a second time. The wait queue mechanism allows setting a customized wake up function which circumvents the problem. [4]

So that the required frequency adjustments are made as soon as possible the kernel thread has a higher priority than all other processes. It is scheduled by a dummy scheduler doing fixed priority scheduling that is also the parent of the root scheduler of the hierarchy. The kernel thread has a higher priority than the root scheduler and preempts the hierarchy whenever it is unblocked. This does not affect the *ALL* guarantee that is provided to the root scheduler because the computing time consumed by the thread can be equated with the stolen time of hardware interrupts.

11.4. Pitfalls

It is not clear how long the frequency transitions take on different architectures. Since the DVS algorithm might change the clock frequency very frequently it may be necessary to defer and combine multiple requests for lower frequency stages. Requests for higher frequencies cannot be buffered without affecting the correctness of the decisions made by the DVS algorithms.

The definition of a scheduler hierarchy requires the definition of rules describing to which scheduler an application should be connected. In addition, it may be necessary to set some scheduler specific parameters, e.g. a priority level. At the moment, the mapping between applications and schedulers depends exclusively on the path of the application in the file system and is applied when a process calls the `exec()` system call to execute another program. When a process forks a child by means of `fork()` the schedulable of the new process is connected to the scheduler of the parent process because both processes execute the same program. Both the parent and the child are each connected to the scheduler with the parameters defined in the mapping.

This is a pitfall because forking a new process might result in an invalid scheduling situation because of overload. A solution to this problem would be to share the application guarantee between the parent process and its children either by assigning each process a part of the guarantee or by inserting a scheduler that distributes it among the processes.

[4] Although the work queue implementation is based on wait queues, the programming interface does not permit setting a customized wake up function.

12. Testing

By Kai Moritz (kai.m.moritz@gmx.de) and Rene Zeglin (rene.zeglin@udo.edu)

12.1. Introductory Considerations

It is difficult to compare a system based on hierarchical scheduling to other systems because the scheduling behavior of such a system depends on the actual composition of the scheduling hierarchy. A normal general purpose operating system has a fix scheduling policy, which must be designed to do its best in common cases while trying not to be to bad in special situations. Using HS, the system can easily be equipped with a new special hierarchy for every scenario it is deployed in. Since scheduling requirements may be contradictory there is no optimal hierarchy and the hierarchy will always be customized to the actual scenario. Thus, comparing the scheduling behavior of a system based on HS is tricky, because it can have thousand different faces.

We decided to take one fix scheduling hierarchy for testing and compare the behavior of this hierarchy to the behavior of an unmodified Linux kernel under varying load situations. To be fair, each program that requires soft real time abilities calls `sched_setscheduler()` when running on the unmodified Linux kernel to acquire the special real time scheduling priority this kernel offers. To achieve comparable results the testing environment was designed to be able to simulate the chosen scheduling scenarios in a reproducible manner. This way the SAADI/Linux kernel and the unmodified Linux kernel can be confronted with exactly the same load situation.

To measure the performance of our kernel level DVS algorithm the `powernowd` daemon is run on the unmodified Linux kernel and the power savings of both systems are compared. We chose `powernowd` because its DVS decision depends only on the CPU load. That is, it behaves in a similar fashion to the DVS algorithms we implemented for our time-sharing schedulers and thus, the results should be fairly comparable. Other frequency scaling daemons consider additional information like battery status, AC status, temperature, fan status, etc. in their DVS decisions, what might mess up the results. For more details see [pow].

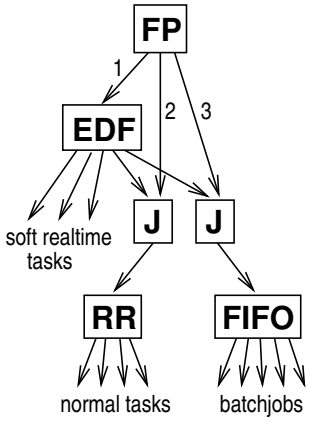


Figure 12.1.: Scheduling hierarchy used for testing

12.1.1. The Testing Scenarios

To show the usefulness of hierarchical scheduling and voltage scaling and to test our implementation we designed a set of scheduling scenarios which make typical demands on the scheduling policy. Each scenario runs for 60 seconds and each simulation consists of 10 test runs. The presented results are the average values of these 10 runs.

12.1.2. The Scheduling Hierarchy Used for Testing

The hierarchy used for testing was designed to cope with common everyday scheduling situations. In addition, it provides some extra features that a normal general-purpose scheduler cannot offer. It supports periodic soft real-time tasks, which require a certain amount of computing time in a specified time period. In contrast to the Linux scheduler the hierarchy preempts soft real-time tasks when the agreed timeslice of a period has been consumed. This way, the CPU cannot be monopolized and starvation is prevented. Furthermore, the hierarchy provides a first-in-first-out (FIFO) queue for batch jobs.

The hierarchy is shown in figure 12.1. A fixed-priority (FP) scheduler plays the role of the root-scheduler. It schedules an earliest-deadline-first (EDF) scheduler with highest priority. Hence, soft real-time schedulables are always guaranteed to receive the reserved CPU-time. The FP scheduler also schedules two join (J) schedulers. These two schedulers are also fed by the EDF scheduler with a small CPU-reservation to prevent starvation. The join scheduler that is scheduled with priority 2 schedules a round-robin (RR) scheduler. The RR scheduler is the default scheduler all normal tasks are mapped to. The join scheduler that is scheduled with the lowest priority by the FP scheduler

schedules the first-in-first-out (FIFO) scheduler batch processes are mapped to.

12.2. Our Framework for Testing and Evaluating

To be able to test our scheduling system under various load situations in a reproducible manner we created a testing environment. It consists of a set of test-programs that run on the target machine and simulate certain categories of programs and a controller program that runs on a second machine. The controller program starts the test-programs over a TCP/IP network connection and stores the acquired results in a database. In addition to these results, kernel events, like for example the forking of a new process, that have been traced by the mLTT on the target machine can also be recorded.[1] The collected data can be analyzed later by scripts that read the results and sampled events from the database. Different load situations can be simulated in a reproducible manner by using so-called scenario files. A scenario file defines which of the test-programs are started at what time and with which options. A simulation runs for a specified duration of time and may consist of multiple identical runs.

12.2.1. Implemented Test-Programs

In literature generally three basic program-classes are distinguished: *interactive programs*, *batch-jobs* and *(soft) real-time processes*. We have implemented a test-programs for each of these categories.

Simulating Interactive Programs (iclient and iserver)

Simulating interactive programs is the most tricky part since we have to generate events that make the tested system think a real user is interacting with it. Modern operating systems apply heuristics to judge if a given program is interactive and I/O-bound or CPU-bound. The most common rule is that an interactive program spends most of the time waiting for user input.

For example for every typed character the hardware generates an interrupt to inform the operating system about this new event. The operating system receives the typed character on behalf of the program waiting for input and than wakes it up. The program does whatever computations it wants to do with the new input and than goes back to sleep waiting for more input. Interactive programs are expected to have only little work to do, so they will fall asleep again very soon. For example a text editor will decide

[1] The mLTT is a device driver that samples kernel events and makes them available to a reading process.

whether it has to echo the character to the display or not and than will wait for more characters. Furthermore the breaks between single user inputs are long periods in terms of computation time. Even the latency between two characters typed is a long time for modern computers. Hence the rule that an interactive program will sleep most of the time and generate a lot of interrupts compared to the amount of computation time it needs.

To simulate the interactivity we have built a server-client application that mimics the behavior of a user typing commands in a shell and reading the results. It consists of two programs: the `iserver` that simulates the shell running on the tested system and the `iclient` that is running on the second machine and simulates a remotely logged in user.

iserver On the test-candidate a small server-program called `iserver` is running that just echoes the characters, which it reads, back to the sending machine after doing some computations in busy-loops. If this program reads a newline character it will execute a larger number of busy-loops after the newline character has been sent back and send an additional special character (the prompt). This is done to simulate the execution of a small program.

iclient On the second machine a program called `iclient` is executed. This program connects to the `iserver` and writes single characters with an adjustable random latency. To measure the responsiveness of the test-candidate it tracks down how much time the remote machine needs to echo the characters. Additionally it will stop writing new characters if it has not received any echo after writing an adjustable number of characters and record to its output that it has to interrupt writing because of a lack of responsiveness. Finally it will start the simulation of a command-execution after writing a limited random number of characters by sending a newline character to the `iserver`. It than will measure how much time the simulation of the command-execution takes by waiting for the prompt to be sent back. When it receives the prompt-character it will fall asleep for a limited random time to simulate the reading of the command-output by the user.

In the final version of our testing environment the functionality of the `iclient` has been integrated into one single program that starts up all the tests on the target machine. This way, no details will be lost, because the evaluated data is received and inserted into the database in raw form instead of a summarized statistic.

Simulating Batch-Jobs

To simulate a batch-job we have written a simple program that just computes CPU-intensive busy-loops for a certain duration of time and than outputs the time it took to do that.

Simulating Soft Real-Time Processes (srt)

The `srt` program tries to mimic the behavior of a common multimedia application which belongs to the class of soft real-time processes. This type of application has to do a certain amount of computations each period before a critical deadline is reached. For example a mp3-player has to decode the next frame of audio data before it has to be sent to the audio-controller or a video-player application has to decode the next video-frame. If these applications miss their deadlines, they have to throw away the done work and start over with the work scheduled for the next period. Obviously, it is desirable that they miss as little deadlines as possible, but if a deadline is missed the system can go on doing its work.

`srt` takes a period, an amount and a number of iterations as arguments and tests whether it is able to busy-loop the given amount of time each period. The program can be run in vanilla or a SAADI-aware mode. If it is run in vanilla mode, it tries to use the soft real-time abilities of a normal Linux kernel which are only available if the program runs with root-privileges. In order to be signaled the beginning of a period it sets up a periodic timer. The option `--SAADI` runs `srt` in SAADI mode and it expects to be mapped to the EDF scheduler. By means of the `sched_param()` system call the program claims the given period and amount as its reservation. The EDF scheduler supports sending a signal at the beginning of a period so that it is not necessary to set up a periodic timer.[2]

In each period the program busy-loops for the given amount of time. If it finishes the computation before the end of the period it records a successful iteration and goes asleep. In vanilla mode this is done by calling a sleep function so that the process sleeps until it is woken up again by the periodic signal. In SAADI mode the EDF scheduler is signaled that the process has finished its computation for the current period so that it will not be scheduled until the begin of the next period. If `srt` is interrupted while busy-looping it records the number of loops that could not be done and restarts its computation trying to reach the next deadline.

[2] A periodic timer that is not in time with the scheduler internal one would make no sense anyway: The EDF scheduler provides a *RESBH* guarantee, i.e. the scheduler prescribes the tact of the periods in which the agreed amount of computing time is provided. For details we refer to [Reg01]

After the given number of iterations has been done or when the program is terminated it prints for each period whether the deadline has been reached or not. In addition to this the number busy-loops that could not be done in due time is printed, too.

Further Test-Programs

For proving the privilege separation provided by the SAADI-Framework we wrote a simple program called `hungry`, that eats up all CPU time for a certain while. If this program is run under a normal Linux kernel it sets its scheduling priority to `RT_PRIORITY`. If the program is run in the SAADI-Framework it claims a certain period and amount.

Notes

The `batch` and `srt` programs have to be calibrated for exactly the computer architecture and kernel version it will run on to get correct results. This is necessary because it counts the amount of work to do in busy-loops and since the computation time of one busy loop is dependent on the performance of the computer and kernel a correct conversion factor between busy-loops and time has to be computed.

12.3. Testing Results

In the first four scheduling scenarios SAADI uses the hierarchy shown in figure 12.1 and an alternative version of it. In the simulations called *SAADI/1* the SFP-scheduler is used as root-scheduler instead of the FP-scheduler. The SFP-scheduler is a simple FP-scheduler, which considers only the needs of the child with the highest priority in its DVS decision. In other words, only the child-scheduler with the highest priority (the EDF-scheduler) can force the root-scheduler to raise the CPU-frequency.

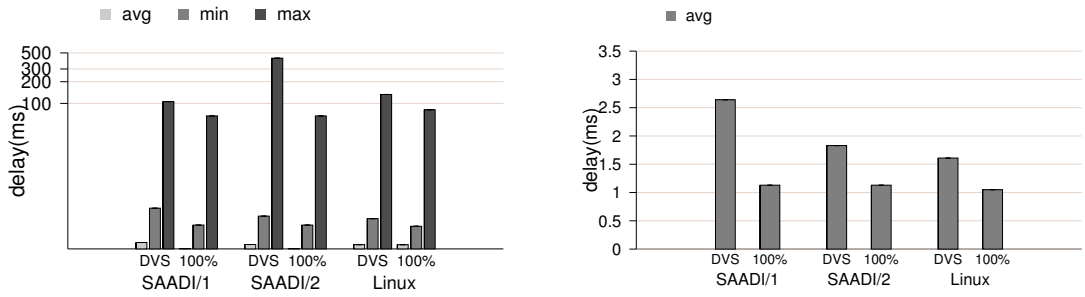
In the test runs called *SAADI/2* the FP-scheduler is used as root scheduler which additionally considers the needs of the RR scheduler in its DVS decision. This means that the CPU-frequency is also increased when the load of the RR scheduler exceeds a certain threshold. However, the load of the FIFO scheduler is not considered in the DVS decision. The intention behind this is that most people probably accept longer turnaround-times of the batch processes if this leads to less energy consumption.

12.3.1. Load Variation

In this scenario two iclients and a soft real-time process with an amount of 10ms and a period of 100ms are running for the whole time. At the 10th, 20th, 30th and 40th

second 6 further iclients and 4 batch processes are started simulating a peak load. The iclients run for 5 seconds and each of the batch processes works for 1 second.

figure[H]



Response times of interactive processes

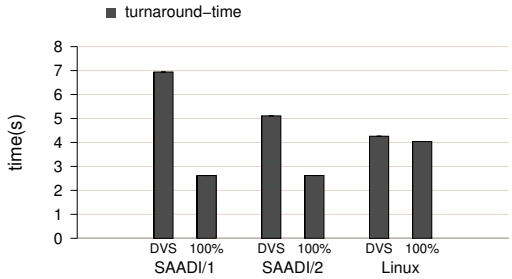


Figure 12.2.: Average turnaround-times of batch processes

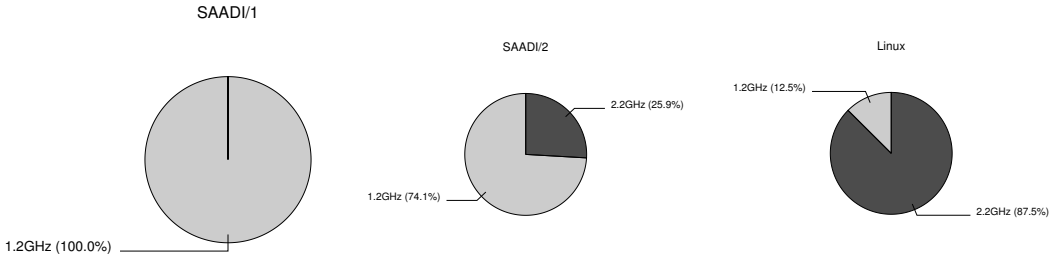


Figure 12.3.: Distribution of non-idle cycles

		SAADI/1	SAADI/2	Linux	SAADI (100%)	Linux (100%)
non-idle	1.2 GHz	100	74.08	12.54	0	0
cycles(%)	2.2 GHz	0	25.92	87.46	100	100
response	min	0.23	0.15	0.15	0.01	0.14
time(ms)	max	105.95	422.80	132.80	66.92	81.53
	avg	2.64	1.83	1.61	1.13	1.05
turnaround	avg	6.94	5.11	4.26	2.62	4.04
time(s)						
missed	#	0	0	0	0	0
deadlines	%	0	0	0	0	0
loops todo	%	0	0	0	0	0

Table 12.1.: Results

The disadvantage of the low frequency used in SAADI are longer average response and turnaround-times. Compared to the results determined with Linux, they are not as bad as might be expected comparing the frequency pie-charts. Regarding the turnaround-times, the reason is that SAADI schedules batch processes first-in-first-out instead of round-robin. This also explains the low turnaround-times of SAADI without DVS. The response times of Linux are increased by the soft real-time task which is always preferred to interactive and batch processes.

12.3.2. Interactive and Batch Processes

In this scenario two iclients run for the whole time. At the 10th second 10 batch processes are started. Each of these batch processes works for 4 seconds.

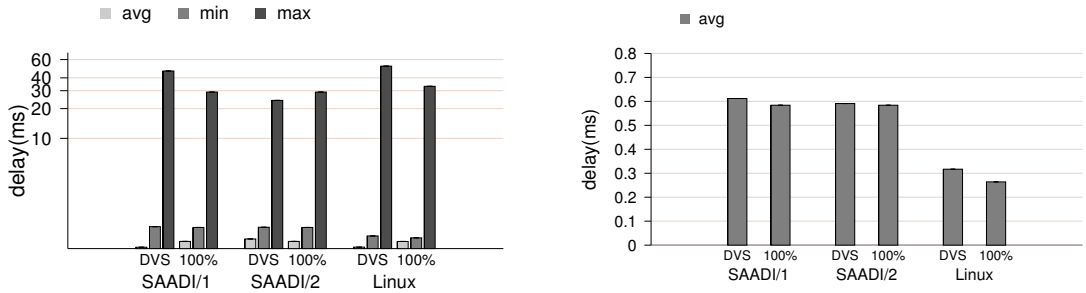


Figure 12.4.: Response times of interactive processes

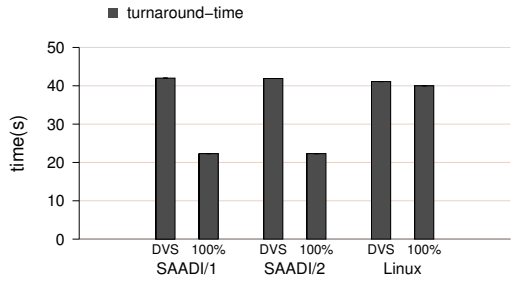


Figure 12.5.: Average turnaround-times of batch processes

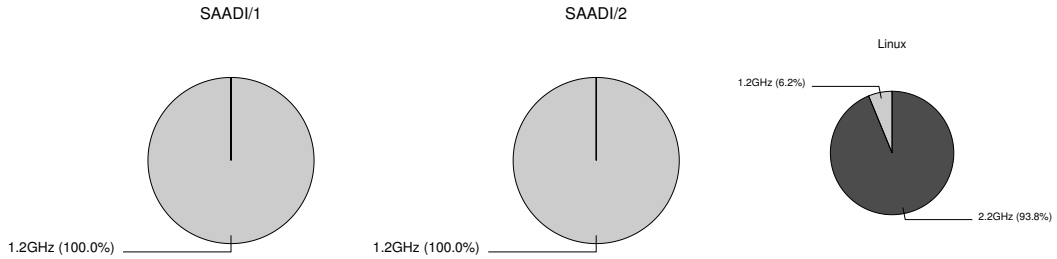


Figure 12.6.: Distribution of non-idle cycles

		SAADI/1	SAADI/2	Linux	SAADI (100%)	Linux (100%)
non-idle	1.2 GHz	100	100	6.21	0	0
cycles(%)	2.2 GHz	0	0	93.79	100	100
response	min	0.03	0.23	0.03	0.17	0.17
time(ms)	max	46.59	24.08	51.93	29.10	33.08
	avg	0.61	0.59	0.32	0.58	0.26
turnaround	avg	42.00	41.94	41.14	22.26	39.99
time(s)						

Table 12.2.: Results

The disadvantage for the low frequency used in SAADI are worse response times of the interactive processes. Again, the scenario shows the advantage of a FIFO-scheduler for batch processes. The average turnaround-times are nearly identical.

12.3.3. Real-time Process and an Increasing Interactive Load

In this scenario a soft real-time process with a period of 100ms and an amount of 20ms runs for the whole time. At each 5th second an iclient is started that runs till the end of the test run. In addition, a further iclient is started at the 45th, 50th and 55th second.

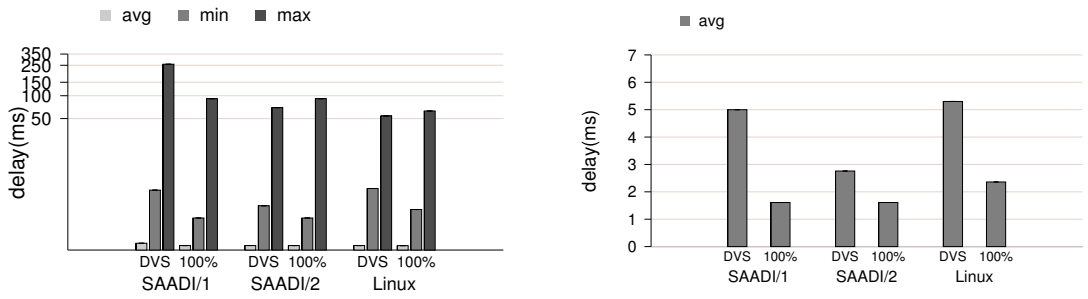


Figure 12.7.: Response times of interactive processes

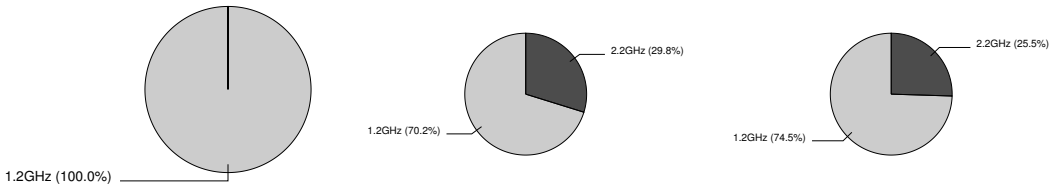


Figure 12.8.: Distribution of non-idle cycles

		SAADI/1	SAADI/2	Linux	SAADI (100%)	Linux (100%)
non-idle	1.2 GHz	100	70.21	74.51	0	0
cycles(%)	2.2 GHz	0	29.79	25.49	100	100
response	min	0.23	0.15	0.14	0.14	0.14
time(ms)	max	257.50	69.77	54.18	91.49	62.99
	avg	5.00	2.76	5.30	1.61	2.36
missed	#	0	0	0	0	0
deadlines	%	0	0	0	0	0
loops todo	%	0	0	0	0	0

Table 12.3.: Results

This scenario shows the advantage of load isolation. The Linux scheduler allows real-time processes to monopolize the CPU which leads to increased response-times or even starvation of the time-sharing processes. The scheduling hierarchy prevents this by reserving a small amount of computing time (in an appropriate period of time) for the RR and FIFO schedulers.

12.3.4. Interactive Processes and an Increasing Real-Time Load

In this scenario two iclients run for the whole time. At the 10th second a soft real-time process with a period of 100ms and an amount of 10ms is started. At the 20th second another real-time process with a period of 100ms and an amount of 50ms is started that runs for 30 seconds. At the 30th second a third real-time process with a period of 120ms and an amount of 10ms is started. The first and the third real-time process run till the end of the test-run.

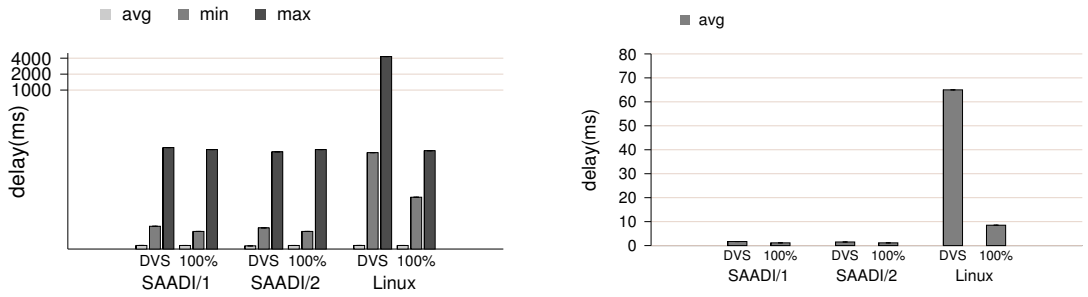


Figure 12.9.: Response times of interactive processes

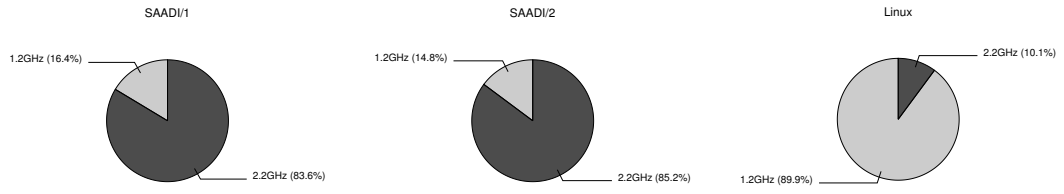


Figure 12.10.: Distribution of non-idle cycles

		SAADI/1	SAADI/2	Linux	SAADI (100%)	Linux (100%)
non-idle	1.2 GHz	16.38	14.82	89.86	0	0
cycles(%)	2.2 GHz	83.62	85.18	10.14	100	100
response	min	0.17	0.15	0.17	0.17	0.14
time(ms)	max	81.22	67.40	4292.34	74.38	62.99
	avg	1.69	1.51	65.03	1.15	2.36
missed	#	0.30	0	269.40	0	0
deadlines	%	0.03	0	30.79	0	0
loops todo	%	1.72	0	21.13	0	0

Table 12.4.: Results

This scenario shows the advantage of task-based voltage scaling, as done by the EDF-scheduler, over an interval-based approach that is used by the cpufreq user-space daemon. An interval-based DVS algorithm cannot guarantee the meeting of deadlines because the reaction to an increased load may take some time. As in the last scenario, the lacking load isolation of the Linux scheduler leads to conspicuously increased response times of the interactive processes. After the second srt-process has been started the CPU is completely occupied by the real-time processes until the cpufreq user-space daemon reacts to the increased load and switches to the high CPU-frequency.

12.3.5. Real-Time and Interactive Programs and an Increasing Batch-Load

In this scenario one real-time process with a period of 100 and an amount of 15 and two iclients are running the whole time. At the 5th, 15th 25th, 35th and 45th second additionally batch-jobs are started. Each of these batch-jobs has work for 12 seconds.

In this and the following scenarios SAADI uses the default-hierarchy shown in figure 12.1. In the simulations marked as *SAADI 1* the FP-scheduler used as root-scheduler accounts for the requirements of all its child-schedulers in its DVS-Algorithm. The amount of the needs of the RR- and the FIFO-scheduler which is considered in the DVS decision of the FP-scheduler is limited. But under heavy load the FP-scheduler can be forced to raise the CPU-frequency by the RR- and the FIFO-scheduler. In the simulations marked as *SAADI 2* the SFP-scheduler is used as root-scheduler. The SFP-scheduler is a simple FP-scheduler, which considers only the needs of the child with the highest priority in its DVS decision. In other words, only the child-scheduler with the highest priority (the EDF-scheduler) can force the root-scheduler to raise the CPU-frequency.

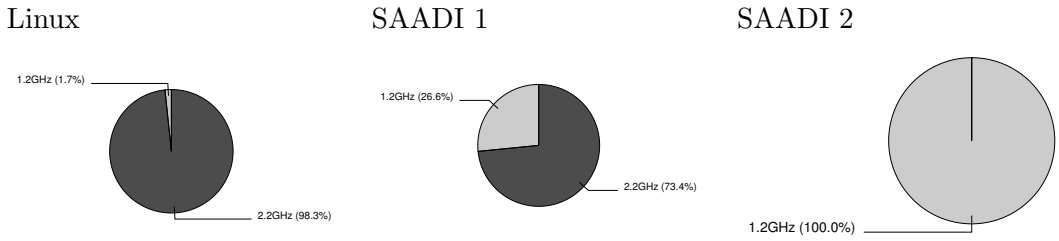


Figure 12.11.: Distribution of Non-Idle Cycles

	min.	max	avg.
Linux	0.04 ms	50.78 ms	0.88 ms
SAADI 1	0.11 ms	33.87 ms	0.51 ms
SAADI 2	0.15 ms	44.90 ms	0.76 ms
Linux 100%	0.17 ms	28.12 ms	0.80 ms
SAADI 100%	0.06 ms	25.75 ms	0.43 ms

Figure 12.12.: Response Times

batchjobs						
delay	work	Linux	SAADI 1	SAADI 2	Linux 100%	SAADI 100%
5 s	12 s	15.66 s	21.20 s	23.33 s	14.36 s	12.27 s
15 s	12 s	21.58 s	26.78 s	96.26%	19.46 s	14.54 s
25 s	12 s	32.53 s	31.36 s	0.01%	30.88 s	19.04 s
35 s	12 s	71.17%	24.25%	0.01%	74.40%	23.51 s
45 s	12 s	38.60%	0.00%	0.01%	40.39%	9.31%
avg. time		23.26 s	26.45 s	23.33 s	21.57 s	17.34 s
unfinished		40.00 %	40.00 %	80.00 %	40.00 %	20.00 %

Figure 12.13.: Turnaround Times for Batchjobs

- Neither Linux nor SAADI has missed any deadlines.

As can be seen in table 12.14 the price for the low frequency used in SAADI 2 is that less batchjobs can be done in the same time. But SAADI 1 gets the same number of batchjobs done as Linux although it is running more time at lower frequency. This effect comes from the FIFO-Scheduler that is used for the batchjobs. Running at full CPU-Power SAADI outperforms Linux, because of its FIFO-scheduling for the batchjobs. It is remarkable, that the delays are a little bit better with SAADI than with Linux. We suppose that the reason for this is the real-time process which is always preferred over the iclients when running on the unmodified Linux.

12.3.6. Real-Time and Interactive Programs and Peak Batch-Load

In this scenario one real-time process with period 100 and amount 10 is running. Two iclient are running for the whole time. At the 15th second 8 batchjobs are started. Each of these batchjobs has work for 2 seconds.

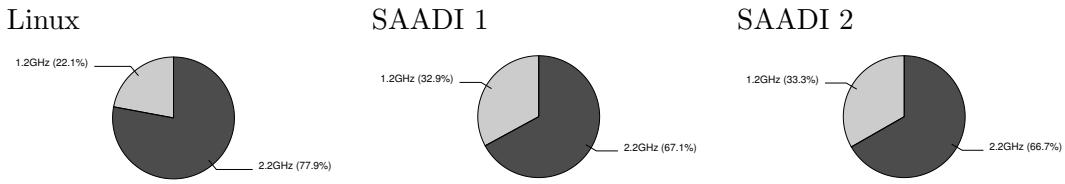


Figure 12.14.: Distribution of Non-Idle Cycles

	min.	max	avg.
Linux	0.19 ms	26.61 ms	1.63 ms
SAADI 1	0.17 ms	33.41 ms	0.67 ms
SAADI 2	0.11 ms	24.53 ms	0.68 ms
Linux 100%	0.15 ms	17.13 ms	0.84 ms
SAADI 100%	0.06 ms	42.66 ms	0.53 ms

Figure 12.15.: Response Times

batchjobs						
delay	work	Linux	SAADI 1	SAADI 2	Linux 100%	SAADI 100%
15 s	2 s	19.23 s	6.01 s	6.44 s	17.86 s	3.68 s
15 s	2 s	19.21 s	6.69 s	6.80 s	17.79 s	5.69 s
15 s	2 s	19.21 s	9.24 s	9.33 s	17.86 s	5.01 s
15 s	2 s	19.08 s	11.31 s	12.32 s	17.92 s	9.51 s
15 s	2 s	19.14 s	13.60 s	13.89 s	17.83 s	11.57 s
15 s	2 s	19.16 s	16.78 s	15.30 s	17.90 s	12.70 s
15 s	2 s	19.14 s	18.15 s	18.25 s	17.83 s	15.67 s
15 s	2 s	19.20 s	20.70 s	20.78 s	17.76 s	17.95 s
avg. time		19.17 s	12.81 s	12.89 s	17.85 s	10.22 s
unfinished		0.00 %	0.00 %	0.00 %	0.00 %	0.00 %

Figure 12.16.: Turnaround Times for Batchjobs

- Neither Linux nor SAADI has missed any deadlines.

As can be seen in table 12.17 SAADI again outperforms Linux because it schedules its batchjobs first-in-first-out. The time the system spends on the lower frequency does not differ much between the simulations, because most of the time there is nearly nothing to do. Nevertheless SAADI 1 and 2 spend a little more time on the lower frequency. The user-space daemon which dose the voltage scaling on Linux cannot react as fast, because of its interval-based DVS-algorithm. Again the delays are a little bit better with SAADI, because a real-time process is running.

12.3.7. Interactive Programs and Increasing Batch-Load

In this scenario two iclients are running the whole time. Every 5 seconds a new batch-job with work for 10 seconds is started until 10 batchjobs were started.

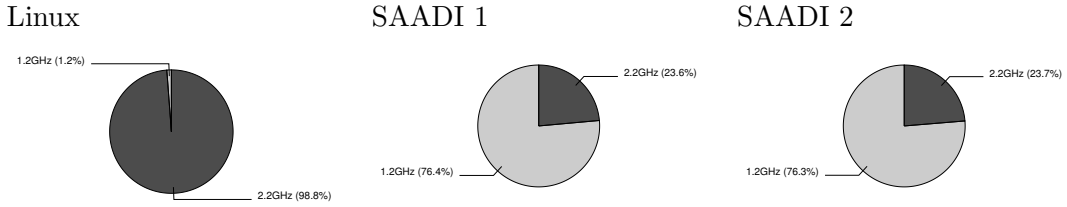


Figure 12.17.: Distribution of non-idle cycles

	min.	max	avg.
Linux	0.17 ms	11.84 ms	0.27 ms
SAADI 1	0.18 ms	16.32 ms	0.46 ms
SAADI 2	0.19 ms	25.76 ms	0.47 ms
Linux 100%	0.16 ms	21.70 ms	0.28 ms
SAADI 100%	0.09 ms	18.43 ms	0.34 ms

Figure 12.18.: Response Times

batchjobs						
delay	work	Linux	SAADI 1	SAADI 2	Linux 100%	SAADI 100%
0 s	10 s	22.49 s	17.53 s	17.45 s	19.72 s	10.26 s
5 s	10 s	44.82 s	29.71 s	29.62 s	43.80 s	15.42 s
10 s	10 s	88.40%	41.96 s	41.93 s	89.71%	20.67 s
15 s	10 s	72.45%	45.39%	45.67%	74.02%	25.88 s
20 s	10 s	59.98%	0.00%	0.02%	61.19%	31.13 s
25 s	10 s	49.01%	0.00%	0.02%	49.37%	77.19%
30 s	10 s	39.79%	0.02%	0.00%	40.03%	8.73%
35 s	10 s	32.18%	0.02%	0.02%	32.12%	0.04%
40 s	10 s	25.13%	0.02%	0.01%	25.64%	0.02%
45 s	10 s	19.64%	0.02%	0.02%	19.77%	0.03%
avg. time		33.66 s	29.73 s	29.67 s	31.76 s	20.67 s
unfinished		80.00 %	70.00 %	70.00 %	80.00 %	50.00 %

Figure 12.19.: Turnaround Times for Batchjobs

As can be seen in table 12.20 SAADI again gets more batchjobs done in the same time as Linux although it is running remarkable more time on the lower frequency. In this scenario the delays are better with Linux because no real-time prevents the Linux-Scheduler from choosing the interactive programs.

12.3.8. Interactive Programs and Heavy Batch-Load

In this scenario two iclients are running the whole time. Each second a new batch-job with work for 10 seconds is started until 10 batchjobs are running.

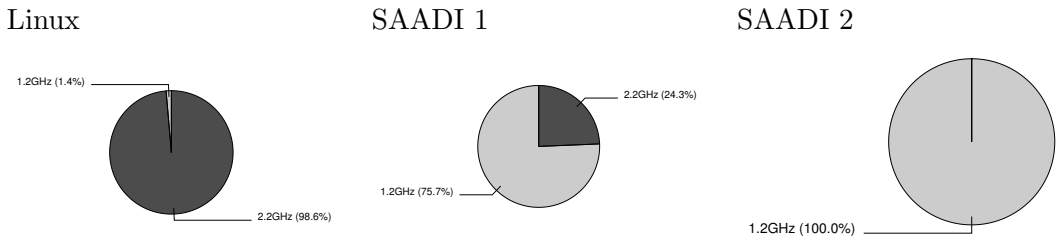


Figure 12.20.: Distribution of non-idle cycles

	min.	max	avg.
Linux	0.15 ms	9.95 ms	0.28 ms
SAADI 1	0.12 ms	17.01 ms	0.53 ms
SAADI 2	0.13 ms	21.87 ms	0.57 ms
Linux 100%	0.16 ms	28.04 ms	0.28 ms
SAADI 100%	0.14 ms	16.55 ms	0.36 ms

Figure 12.21.: Response Times

batchjobs						
delay	work	Linux	SAADI 1	SAADI 2	Linux 100%	SAADI 100%
1 s	10 s	69.08%	17.32 s	19.60 s	74.35%	10.35 s
2 s	10 s	64.39%	33.45 s	38.19 s	65.80%	19.47 s
3 s	10 s	61.78%	49.83 s	9/10 56.48 s 98.95%	61.69%	28.71 s
4 s	10 s	59.49%	40.91%	1.82%	59.21%	37.94 s
5 s	10 s	57.00%	0.00%	0.00%	56.89%	47.15 s
6 s	10 s	55.21%	0.01%	0.00%	55.27%	75.78%
7 s	10 s	53.88%	0.02%	0.01%	53.73%	0.00%
8 s	10 s	52.59%	0.02%	0.00%	52.43%	0.00%
9 s	10 s	51.77%	0.02%	0.02%	51.66%	0.03%
10 s	10 s	50.51%	0.02%	0.02%	50.84%	0.02%
avg. time		0.00 s	33.53 s	37.46 s	0.00 s	28.72 s
unfinished		100.00 %	70.00 %	71.00 %	100.00 %	50.00 %

Figure 12.22.: Turnaround Times for Batchjobs

This scenario again shows the advantages of the possibility to schedule batchjobs first-in-first-out.

12.3.9. A Hungry Real-Time Process

In this scenario a real-time process with period 512 and amount 62 is running. Two iclients are running the whole time. In the 5th, 10th, 15th, 20th, and 25th second new batchjobs each with work for 6 seconds are started. Additionally a *hungry* real-time process with work for 45 seconds is started in the 15th second. This process eats up all processing time it gets until it has done as much work as if it was working alone with the whole CPU-Power for 45 seconds. On the unmodified Linux-Kernel the *hungry* process is mapped like a real-time process. The idea is, to show Linux is not able to provide load-isolation for real-time processes.

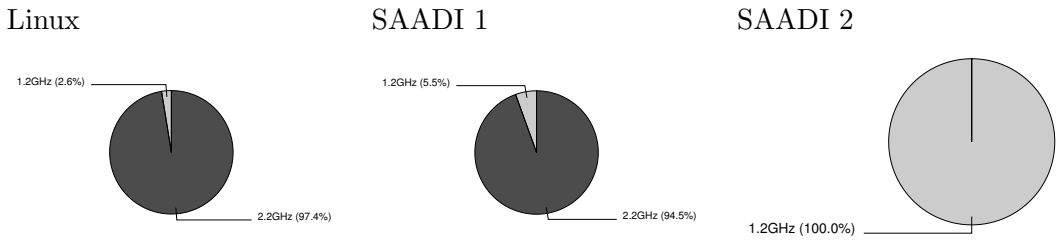


Figure 12.23.: Distribution of non-idle cycles

	min.	max	avg.
Linux	0.19 ms	4255.38 ms	57.35 ms
SAADI 1	0.17 ms	53.87 ms	0.61 ms
SAADI 2	0.16 ms	62.21 ms	1.38 ms
Linux 100%	0.14 ms	4204.13 ms	56.21 ms
SAADI 100%	0.09 ms	66.12 ms	0.52 ms

Figure 12.24.: Response Times

The response times shown in table 12.25 are not sufficient to show how the normal Linux-Kernel is blocked during the execution of the *hungry* real-time process because no responses are made during that time at all. Because of that, the iclients observe if they type more than three characters without receiving any response. In this case they stop typing and wait for the next echo. Just like a real user who stops typing commands if he cannot see them on the screen. Table 12.26 shows the results.[3]

[3]In all other scenarios are no wait times at all, thus that results are only shown for this scenario.

	iclient	
	1	2
Linux	39.11 s	34.97 s
SAADI 1	0.00 s	0.00 s
SAADI 2	0.00 s	0.00 s
Linux 100%	39.18 s	39.23 s
SAADI 100%	0.00 s	0.00 s

Figure 12.25.: Total Time Waiting for any Response

batchjobs						
delay	work	Linux	SAADI 1	SAADI 2	Linux 100%	SAADI 100%
5 s	6 s	5/10 36.74 s 96.40%	9.16 s	15.38 s	9.15 s	7.04 s
10 s	6 s	37.88%	11.12 s	0.00%	43.79%	9.06 s
15 s	6 s	2.22%	0.00%	0.03%	2.76%	0.00%
20 s	6 s	1.88%	0.00%	0.02%	1.91%	0.04%
25 s	6 s	1.47%	0.04%	0.00%	1.53%	0.03%
avg. time		36.74 s	10.14 s	15.38 s	9.15 s	8.05 s
unfinished		90.00 %	60.00 %	80.00 %	80.00 %	60.00 %

Figure 12.26.: Turnaround Times for Batchjobs

	iclient	
	1	2
Linux	39.11 s	34.97 s
SAADI 1	0.00 s	0.00 s
SAADI 2	0.00 s	0.00 s
Linux 100%	39.18 s	39.23 s
SAADI 100%	0.00 s	0.00 s

Figure 12.27.: Total Time Waiting for any Response

Linux	0.10	(0.32%)
SAADI 1	0.00	(0.00%)
SAADI 2	0.00	(0.00%)
Linux 100%	0.00	(0.00%)
SAADI 1 100%	0.00	(0.00%)

Figure 12.28.: Missed Deadlines (Average over all runs)

Again Linux gets more batchjobs done. Even SAADI 2 gets more batchjobs done than Linux although it is running at the lower frequency the whole simulation. As can be seen in table 12.29 Linux misses some deadlines, while SAADI 1 and 2 are catching all.

This scenario clearly shows the advantages of privilege separation. An inherit feature of hierarchical scheduling which cannot be achieved with a normal Linux-Kernel.

13. Conclusion

13.1. Conclusion and future work

We demonstrated in our project group, BeeHive, that ideas inspired from natural system provide a sufficient motivation for designing and developing algorithms for scheduling and routing problems. We have followed an engineering approach that allowed us to map concepts from a bee colony to a routing algorithm. We evaluated our algorithm in a simulation environment, however, our simulation model was developed by an early feedback from the Linux routing group. In this way, the algorithmic development phase took into account the constraints of the real routing framework. Such an approach smooth lined the implementation of BeeHive algorithm under Linux operating system. We have done extensive testing and evaluation under varying environmental parameters that represent the real network conditions. The results from all experiments reveal that the performance of BeeHive is of the order of the best algorithm (DSR) , however, this excellent performance is achieved at a much less energy expenditure. Hence, BeeHive is energy efficient, simple but delivers the best performance. In the second phase of our project, we implemented BeeHive inside the network framework of Linux operating system. We designed a testing infra-structure that simulated different scenarios in real time and then tested the algorithm with the help of this infra-structure. Unfortunately, the simulation scenarios could not have been mapped to this framework because of the complexity of modifying Linux network framework. However, with the help of the framework we have done the functional verification of the algorithm. In the final phase we tested the algorithm on real adhoc networks to verify its functional correctness. This approach gave us a good insight into the real adhoc network environments. Finally, in this project group, we have demonstrated that an energy efficient framework is incomplete without energy efficient scheduler. We have developed a hierarchical DVS scheduling algorithm for the Linux operating system that gives the user the ability to write scheduling policies. This system is able to guarantee certain scheduling requirements to the applications while at the same time trying to meet them with as little energy consumption as possible. The experiments have demonstrated that SAADI DVS is able to scale the frequency and

voltage of the processor in a sophisticated manner as compared to the standard Linux DVS algorithm. We believe that BeeHive opens new dimensions for the routing problem in adhoc networks. The work in this project group could be extended in the following manner

1. BeeHive algorithm could be modified in such a manner that it is able to scale to 1000+ nodes. This objective will make BeeHive algorithm to scale to large adhoc networks and hence suitable for sensor networks.

2. BeeHive algorithm is not secure at the moment. A challenging task is to make it secure so that the network is not susceptible to security attacks.

3. Developing a Multi-agent System that helps us in doing a multi-objective optimization in a simple and energy efficient manner.

4. SAADI DVS algorithm should be extended in such a manner that the hierarchy could be modified on the run if there is a change in the application requirements.

BeeHive has shown a novel approach to all of us for the routing and scheduling problems. We hope that the project will help the research community in exploring the honey bee colony for other problems as well.

A. Sample Scheduler-Implementation: The Simple Fixed-Priority Scheduler

A.1. include/linux/saadi/sched_sfp.h

```
#ifndef _SAADI_SCHED_SFP_H_
#define _SAADI_SCHED_SFP_H_

#ifdef __KERNEL__
#include <linux/module.h>
#include <linux/saadi/saadi.h>
#endif

struct sfp_sched_param {
short prio;
};

typedef struct sfp_sched_param sfp_sched_param_t;

#ifdef __KERNEL__

#define SFP_TYPE_NAME "sfp"

#define SFP_MAX_PRIO 128
#define SFP_BITMAP_SIZE ((SFP_MAX_PRIO + sizeof(long) - 1)/ sizeof(long))


struct sfp_rq_data {
schedulable_t **children;
unsigned long *linked;
```

```
unsigned long *active;
unsigned int gfrac_sum;
};

typedef struct sfp_rq_data sfp_rq_data_t;


struct sfp_sched_data {
short prio;
unsigned int curr_gfrac;
};

typedef struct sfp_sched_data sfp_sched_data_t;

extern void construct_sfp_scheduler(scheduler_t *sched);
#endif /* __KERNEL__ */
#endif /* _SAADI_SCHED_SFP_H_ */
```

A.2. include/linux/saadi/sched_sfp.c

```
/*
 *  kernel/saadi/sched_sfp.c
 *
 *  simplified fixed priority scheduler
 *
 *  This scheduler only accepts one child per priority!
 *
 *  (c) 2004 Kai Moritz <kai.m.moritz@gmx.de>
 *  (c) 2004 Rene Zeglin <rene.zeglin@udo.edu>
 *
 */

#include <linux/saadi/sched_sfp.h>

/* link a schedulable with the scheduler */
static int sfp_link(scheduler_t * sched, schedulable_t * s)
```

```
{
sfp_sched_data_t *sched_data = (sfp_sched_data_t *)s->sched_data;
sfp_sched_param_t *sched_param = (sfp_sched_param_t *)s->sched_param;
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)sched->rq_data;

/* sfp accepts only schedulers as childs */
if (unlikely(!s->sched))
panic("trying to link task to sfp!");

if (unlikely(sched_param->prio > SFP_MAX_PRIO))
panic("requested priority %i is to big!", sched_param->prio);

sched_data->prio = sched_param->prio;
sched_data->curr_gfrac = 0;

if (unlikely(rq_data->children[sched_data->prio]))
/* There is already a schedulable of this
 * priority linked to the scheduler!  */
panic("sfp only accepts one child for every priority!");

/* Save a pointer to the new schedulable */
rq_data->children[sched_data->prio] = s;
set_bit(sched_data->prio, rq_data->linked);

MLTT_SCHEDULING_TRACE(
SFP_LINK,
s->task ? 1 : 0,
sched->id,
s->task ?
   (unsigned long)s->task->pid :
(unsigned long)s->sched->id
);

return SAADI_ACCEPT;
}
```

```
/* unlink a schedulable with our scheduler */
static void sfp_unlink(schedulable_t * s)
{
sfp_sched_data_t *sched_data = (sfp_sched_data_t *)s->sched_data;
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)s->parent->rq_data;

rq_data->children[sched_data->prio] = NULL;
clear_bit(sched_data->prio, rq_data->linked);

MLTT_SCHEDULING_TRACE(
SFP_UNLINK,
s->task ? 1 : 0,
s->parent->id,
s->task ?
   (unsigned long)s->task->pid :
(unsigned long)s->sched->id
);
}

/* a schedulable joins the runqueue (and the competition) */
static int sfp_join(schedulable_t * s)
{
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)s->parent->rq_data;
sfp_sched_data_t *sched_data = (sfp_sched_data_t *)s->sched_data;
int pre_bit;

MLTT_SCHEDULING_TRACE(
SFP_JOIN,
s->task ? 1 : 0,
s->parent->id,
s->task ?
   (unsigned long)s->task->pid :
(unsigned long)s->sched->id
);

/* get priority of runnable schedulable with highest priority */
```

```
pre_bit = find_first_bit(rq_data->active, SFP_MAX_PRIO);

set_bit(sched_data->prio, rq_data->active);

if (sched_data->prio < pre_bit && pre_bit < SFP_MAX_PRIO)
/* The new schedulable has a higher priority,
 * so signal rescheduling.
 */
return SAADI_NEED_RESCHED;

return 0;
}

/* a schedulable leaves the runqueue (and the competition) */
static void sfp_leave(schedulable_t *s)
{
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)s->parent->rq_data;
sfp_sched_data_t *sched_data = (sfp_sched_data_t *) s->sched_data;

MLTT_SCHEDULING_TRACE(
SFP_LEAVE,
s->task ? 1 : 0,
s->parent->id,
s->task ?
   (unsigned long)s->task->pid :
(unsigned long)s->sched->id
);

clear_bit(sched_data->prio, rq_data->active);
}


static schedulable_t *sfp_dispatch(scheduler_t *sched)
{
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)sched->rq_data;
int idx;
```

```
/* Get priority of runnable schedulable with highest priority */
idx = find_first_bit(rq_data->active, SFP_MAX_PRIO);

if (idx == SFP_MAX_PRIO) {
MLTT_SCHEDULING_TRACE(SFP_EMPTY_RUNQUEUE, sched->id);
return NULL;
}

MLTT_SCHEDULING_TRACE(
SFP_DISPATCH,
rq_data->children[idx]->task ? 1 : 0,
rq_data->children[idx]->parent->id,
rq_data->children[idx]->task ?
   (unsigned long)rq_data->children[idx]->task-> pid :
(unsigned long)rq_data->children[idx]->sched->id
);

return rq_data->children[idx];
}

/* a yield has no effect for a fixed-priority scheduler */
static void sfp_yield(schedulable_t *s) { }

/* the child with highest priority will never be interrupted */
static int sfp_scheduler_tick(schedulable_t *s) { return 0; }


/* a child scheduler tells the sufficient guarantee fraction */
static void sfp_set_minfreq(scheduler_t *child, unsigned int gfrac)
{
schedulable_t *s = child->this_schedulable;
scheduler_t *sched = s->parent;
sfp_sched_data_t *sched_data = (sfp_sched_data_t *)s->sched_data;
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)sched->rq_data;
unsigned int new_gfrac;
```

```
MLTT_GUARANTEE_TRACE(TELL_G_FRAC, gfrac, child->id, sched->id);

rq_data->gfrac_sum -= sched_data->curr_gfrac;
rq_data->gfrac_sum += gfrac;
sched_data->curr_gfrac = gfrac;

if (rq_data->gfrac_sum > 128)
new_gfrac = 128;
else
new_gfrac = rq_data->gfrac_sum;

MLTT_DVS_TRACE(SFP_DVS, rq_data->gfrac_sum, sched->id);
sched->this_schedulable->parent->set_minfreq(sched, new_gfrac);
}

static void sfp_schedctl(int cmd, schedulable_t *s, void *data)
{
/* this scheduler implements no special schedctl commands */
SAADI_DEBUG("FP: unrecognized schedctl command %i\n", cmd);
}

static void sfp_schedmsg(
scheduler_t *sender,
scheduler_t * myself,
int msg,
void *data)
{
sfp_rq_data_t *rq_data;
schedulable_t *s;
int idx;

switch (msg) {
case SCHED_MSG_PERIOD:
/* propagate signal to child with highest priority */
rq_data = (sfp_rq_data_t *)myself->rq_data;
```

```
idx = find_first_bit(rq_data->linked, SFP_MAX_PRIO);

if (idx < SFP_MAX_PRIO) {
s = rq_data->children[idx];

/* sfp accepts only schedulers as childs,
 * so we do not have to check wether the
 * child is a task or a process...
 */
s->sched->schedmsg(
myself,
s->sched,
SCHED_MSG_PERIOD,
NULL);
}
break;
case SCHED_MSG_DESCHEDULE:
break;
default:
SAADI_DEBUG("SFP: unrecognized schedmsg %i\n", msg);
}
}

void construct_sfp_scheduler(scheduler_t *sched)
{
sfp_rq_data_t *rq_data = (sfp_rq_data_t *)sched->rq_data;
int j;

sched->name = "sfp";

sched->link_to = saadi_default_link_to;
sched->unlink_with = saadi_default_unlink_with;
sched->link = sfp_link;
sched->unlink = sfp_unlink;
sched->join = sfp_join;
sched->leave = sfp_leave;
```

```
sched->scheduler_tick = sfp_scheduler_tick;
sched->dispatch = sfp_dispatch;
sched->yield = sfp_yield;
sched->destructor = saadi_free_scheduler;
sched->schedctl = sfp_schedctl;
sched->schedmsg = sfp_schedmsg;
sched->set_minfreq = sfp_set_minfreq;
#ifdef CONFIG_SAAID_PROCFS
sched->schedinfo = default_schedinfo;
#endif

rq_data->children =
kmalloc(sizeof(schedulable_t *)*(SFP_MAX_PRIO), GFP_ATOMIC);
if (!rq_data->children)
panic("cannot allocate memory for sfp-scheduler (children)!\n");

rq_data->linked = kmalloc(sizeof(long)*(SFP_BITMAP_SIZE), GFP_ATOMIC);
if (!rq_data->linked)
panic("cannot allocate memory for sfp-scheduler (linked)!\n");
rq_data->active = kmalloc(sizeof(long)*(SFP_BITMAP_SIZE), GFP_ATOMIC);
if (!rq_data->active)
panic("cannot allocate memory for sfp-scheduler (active)!\n");

for (j = 0; j < SFP_MAX_PRIO; j++) {
rq_data->children[j] = NULL;
__clear_bit(j, rq_data->linked);
__clear_bit(j, rq_data->active);
}

rq_data->gfrac_sum = 0;
}

EXPORT_SYMBOL(construct_sfp_scheduler);

MODULE_DESCRIPTION("SAADI scheduler 'sfp'");
MODULE_LICENSE("GPL");
```

Bibliography

[adh] Ad hoc sim 1.0, nicola concer homepage: http://www.cs.unibo.it/ concer/.

[Bro] Dominik Brodowski. *CPUfreq documentaton inside the kernel-2.6.0-test4 sources*. You find this inside Documentation/cpufreq.

[CEPD02] Elizabeth M. Belding-Royer Charles E. Perkins and Samir R. Das. Ad hoc on-demand distance vector (aodv) routing. In *INTERNET DRAFT*, 2002.

[DBJH04] David A. Maltz David B. Johnson and Yih-Chun Hu. The dynamic source routing protocol for mobile ad hoc networks (dsr). In *INTERNET DRAFT*, 2004.

[DHSS] L. Donckers, P.J.M. Havinga, G.J.M. Smit, and L.T. Smit. Enhancing energy efficient tcp by partial reliability.

[Dye02] Fred D. Dyer. The biology of the dance language. *Annu. Rev. Entemol.*, 47:917–949, 2002.

[EB95] H. E. Esch and J. E. Burns. Honeybees use optic flow to measure the distance of a food source. *Naturwissenschaften (quoted from [Dye02])*, 82:38–40, 1995.

[EB96] H. E. Esch and J. E. Burns. Distance estimation by foraging honeybees (nach [Dye02]). *Journal of Experimental Biology*, 199:155–162, 1996.

[(Ed02] J. Reynolds (Editor). Assigned numbers. RFC 3232, January 2002.

[Fee99] Laura Marie Feeney. A taxonomy for routing protocols in mobile ad hoc networks. In *SICS Technical Report*, 1999.

[Flo99] S. Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 2582*, 1999.

[Fri23] Karl von Frisch. *Über die "Sprache der Bienen", eine tierpsychologische Untersuchung (quoted from [Fri65], page 283)*. Zoologisches Jahrbuch (Physiol.), Nr. 40, 1923.

[Fri65] Karl von Frisch. *Tanzsprache und Orientierung der Bienen*. Springer Verlag Berlin, Heidelberg, 1965.

[JRL03] Alan Jay Smith Jacob R. Lorch. Operating system modifications for task-based speed and voltage scheduling. *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, 2003.

[KSC99] R. Kravets, K. Schwan, and K. Calvert. Power-aware communication for mobile computers, 1999.

[net] *netfilter/iptables*. http://www.netfilter.org/.

[ns2] Ns-2 homepage: http://www.isi.edu/nsnam/ns/.

[omn] Omnet++ homepage: http://www.omnetpp.org/.

[otc] Otcl homepage: http://otcl-tclcl.sourceforge.net/otcl/.

[PGa] University of Dortmund Project Group 424. SAADI - integrated design approach of adapted schedulers (final report).

[PGb] University of Dortmund Project Group 439. BEEhive - an energy-aware scheduling and routing framework (interim report).

[PGc] University of Dortmund Project Group 439. BEEhive - an energy-aware scheduling and routing framework (seminarpaper).

[pow] *PowerNowd*. http://www.deater.net/john/powernowd.html.

[Reg01] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[rfc81] Internet protocol. RFC 791, Information Science Institute, University of Southern Carolina, December 1981.

[See92] Thomas D. Seeley. The tremble dance in the honey bee: message and meanings. *Behavioral Ecology and Sociobiology*, 31:375–383, 1992.

[See95] Thomas D. Seeley. *The Wisdom Of The Hive – The Social Physiology of Honey Bee Colonies*. Harvard University Press, Cambridge, Massachusetts, 1995.

[stl] Sgi stl homepage: http://www.sgi.com/tech/stl/.

[Sum00] David J. T. Sumpter. From bee to society: An agent-based investigation of honey bee colonies. *Phd Thesis submitted at the University of Manchester*, march, 2000.

[SV88] Thomas D. Seeley and P. Kirk Visscher. Assesing the benifits of cooperation in honebee foraging: Search costs, forage quality, and competitive ability. *Behavioral Ecology and Sociobiology*, 22:229–237, 1988.

[TL] V. Tsaoussidis and A. Lahanas. Exploiting the adaptive properties of a probing device for tcp in heterogeneous networks.