Reliable and energy efficient algorithms for sensor networks used in flood protection

Tobias Baumgartner

June 1, 2006

Contents

List of Figures

List of Tables

1 Introduction

Sensor networks consist of a large number of small devices, each containing one or more sensors, a processor, a radio transmitter, memory, and a limited amount of energy generally given by a battery, whereas each node merely measures the values given by its sensors. They communicate with each other via exchanging messages over their radio transmitter. Joined together, they build a powerful collective and are able to solve complex tasks. In the absence of a central control unit, which would enable access to a global view of the network, the necessary information has to be gathered by carrying the local information of each node together. Due to the restrictions in energy consumption, there is a need of a special kind of algorithms with a main focus on energy efficiency.

In general, the nodes are distributed randomly in a given region, so that algorithms must be self-organizable and scalable. In addition, localization hardware like GPS is too expensive and energy wasting to equip many nodes, or even each one with it. There are approaches to solve this with localization algorithms to enable a global position awareness on each node, but depending on ranging errors the results are imprecise and not satisfying so far. As a consequence it is preferable to make use of the benefits of strict local algorithms. In such an approach, the sensor nodes are not aware of their position, neither in an absolute nor in a relative global coordinate system, and merely use the information of the contiguous neighbors.

Sensor networks can be used, for example, for animal observation in large areas, movement analysis of fire fronts in forest fires, traffic supervision, habitat monitoring, or flood protection.

This thesis shall deal with the latter case. The sensors are therefore inserted into sandbags and measure the actual wetness of the surrounding sand. With this information the network is able to build a general view of the moisture penetration the sandbag structure is exposed to. There are three avenues of approach conceivable. Either a base station requests a status report of the nodes in the sandbag structure, or they report their state in periodic intervals independently, or potential leaked sections are announced. Such information must be routed through the network which reduces the overall energy level of the nodes.

By assuming that nodes below the waterline are no longer able to communicate with their neighbors, it is preferable to route messages especially over nodes in the near of the waterline. Those nodes run the risk of losing their ability to communicate in the foreseeable future, and thus can be exhausted instead of ones that are supposed to be part of the network over a longer period. The communication loss can be assumed, for example, in the consequence of physical restrictions of radio wave propagation through wetted sand on the one hand, or nodes that are not water resistant on the other hand.

However, the construction of such a communication backbone can be divided into multiple tasks. Firstly, the waterline must be identified by fault-tolerant techniques that are robust against measurement errors. Secondly, in the consequence of the lack of position awareness, the nodes must be able to coordinate themselves. Thirdly, the communication backbone is built in the near of the identified waterline, and must provide a continuous connected structure that also recognizes potential missing parts in the topology. Finally, a routing algorithm that uses the backbone structure must be designed.

Chapter 2 describes the related work which has already been done in the context of this thesis. In Chapter 3, a general overview of sandbagging and dikes in the sense of flood protection is given. Chapter 4 presents the designed algorithms. Chapter 5 introduces the used simulation environment and illustrates the idea of the implementation. In Chapter 6, the results of the simulations are shown. At last, Chapter 7 summarizes this thesis and gives a short forecast.

2 Related Work

This chapter gives an introduction to the related work covering the basic objectives of this thesis. At first, selected routing algorithms for sensor networks are presented, followed by approaches of organizing and coordinating a network. Then, literature about general as well as moisture measurement techniques is presented. At last, a basic overview of floods, dikes, and the usage of sandbags is given.

Intanagonwiwat et al. [CIE00] describe the *Directed Diffusion*, a data-centric routing algorithm which implies that the generated data is named by attribute-value pairs. For example, a node starts an interest which again is diffused throughout network. The requested data in turn is sent back to that node by the use of multiple paths and additional data caching. Hence, there are methods for the propagation of requests as well as ones for the aggregation of data.

Johnson et al. [JM96] present the *Dynamic Source Routing*. If one node sends a message to another one, it starts a route request that is flooded over the network. If the request reaches the destination, or a node is aware of an already known cached route, a route reply message is sent back to the initiator. A route contains the addresses of the nodes through which the message can be forwarded. In addition, the maintenance of cached routes is handled.

Nieberg et al. [NDH+03] have designed the *Collaborative Algorithms for Communication*. The basic idea is to cluster the network whereby the clusterheads form a dominating set. The non-clusterheads in turn act as gateways. To allow the routing of messages, the already above mentioned *Dynamic Source Routing* is adapted to fit the clustered structure of the network. For an additional improvement of reliability, messages are sent on multiple paths as well as divided into several packets on demand.

EQos [SCC04] is a two phase protocol, whereby the first phase creates a virtual communication backbone, turn the radios of some nodes off and let the nodes learn their rough coordinates. In the second phase redundant sensors turn their hardware off due to information of the rough coordinates and the local neighborhood.

ASCENT by Cerpa et al. [CE04] has been designed for energy efficiency reasons. The basic idea is to have only a few nodes active routing the messages through the network to perform multihop routing. The other ones are passive and check periodically if they should wake up and become active. If an active node recognizes that too many messages getting lost, it sends a *neighbor announcement message* to wake up passive node to be integrated into the backbone. All things considered, there are four different states a node could take: *sleep*, *passive*, *test* and *active*.

Another very simple approach is *SPAN* [CJBM01] which elects coordinators randomly. It ensures that there are enough coordinators, rotates the coordinators due to energy efficiency, attempts to minimize the number of coordinators, and elects the coordinators using only local information.

Youngis et al. have designed the clustering algorithm *HEED* [YF04] that periodically selects cluster heads according to their remaining energy level and a variable second parameter such as the neighborhood density. Unlike other clustering algorithms such it does not need any location information.

Another general problem in sensor networks is the appearance of measurement errors that must be recognized by the nodes. Krishnamachari et al. [KI03] present a Bayesian fault-recognition algorithm that considers the own reading as well as the ones of the neighborhood. In contrast, Ishar et al. [IPPR03] discuss methods of fault-tolerant feature extraction in sensor networks at a central unit which receives the readings of multiple nodes. In a more general manner Chugh et al. [CDA03] present a cluster-based method for event notification in appropriate regions. A similar subject is described by Koushanfar et al. [KSPSV02] who present an error-tolerant fusion of sensor data.

An overview of measurements in general is given in [PP92] that is an introduction in basic measurement techniques, theoretical fundamentals, and causes of potential errors. The latter is described in detail by Grabe [Gra05]. In addition, there is also specialized literature about moisture measurement, and can be found in [Kup97], for instance.

However, this thesis presents algorithms for sensor networks potentially used in flood protection. There is much basic literature about floods available. In [Fle02], an overview of the flood risk in the UK is given. Ohlig [Ohl04] presents historical observations with a focus on flood protection at the Elbe. The problems of floods with respect to seepage and stability reasons of dikes are presented in [Dav64], which in turn leads to the demand on the use of sandbags. Different application areas can be found in [Bra03], whereas Kobayashi et al. [KJ85] as well as Gadd [Gad88] have analyzed the stability and construction issues of sandbags for slope protection, but with a main focus on coastal regions. A more general idea is given by Burnett et al. [BW99] who have filed a patent on additional fill material for sandbags to reduce the leakage. A detailed overview of flood protection is given in the next chapter.

3 Flood Protection

River flooding is a natural disaster that can cause very costly damage by destroying human life, buildings, villages, or even whole districts. In the consequence of excessive rainfall or melting snow, and if the land is no longer able to drain the present amount of water due to a saturated ground, a river can overwhelm its banks and flood the bordering areas.

There are several examples of such disasters in history and recent time, respectively. For example, in Easter 1998 a flood affected an area in the United Kingdom from Worcestershire to Cambridgeshire for six days and caused a damage of £500-700 million [Fle02]. In Autumn 2000 there was the most intensive rainfall since records began over 270 years ago. In October, for instance, the rainfall was four times the average for a month. The related flood caused a damage of £1 billion [Fle02]. Another example occurred in Germany is the Elbe flooding in August 2002. More than 100 people died and the caused damage exceeded 15 billion Euro [RMS03].

To protect an area against flooding, a widely spread defense strategy is the building of dikes. Other methods are described in detail in [Fle02], for instance, but are beyond the subject of this thesis. However, a basic overview of dikes and potential problems of moisture penetration is given in [Dav64]. Figure 3.1 shows a simplified example.

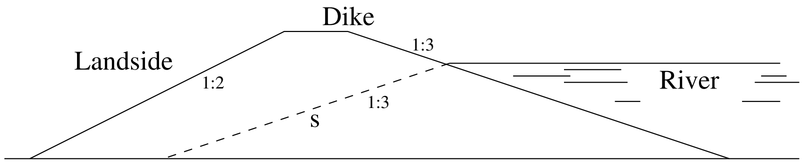
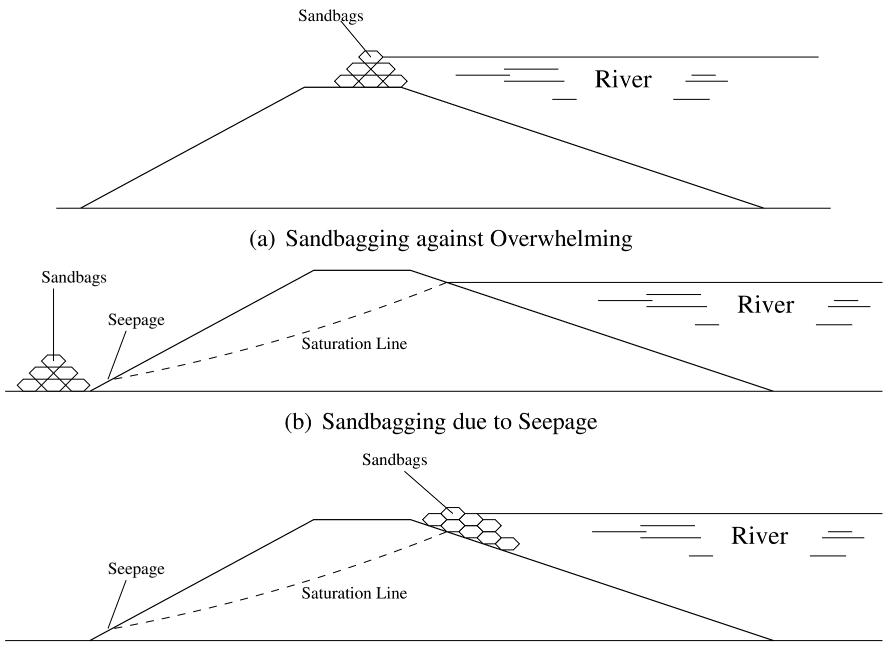


Figure 3.1: Example of a Dike. *The figure shows an example of a dike which has been built to protect an area against flooding, and is a simplification of an illustration that can be found in [Dav64]. On an increasing waterline, the structure runs the risk of getting soaked. Such a situation is indicated by the dashed line s that is also called a $saturation\ line$. If the end of the line is located at the landside of the dike instead of the ground, there is a leakage at the according position, and a dike breakage can occur.*

There are several reasons that can cause a risk for the dike. For example, in the consequence of the mentioned problem of moisture penetration, the dike runs the risk of breakage which in turn causes essential danger for the bordering regions. Furthermore, it is possible that the water level overtops the height of the dike in the foreseeable future, and would also result in flooding the concerning areas. Such situations require additional methods of dike defense. A widely spread practice is the usage of sandbags. Figure 3.2 shows potential application areas for sandbagging, and can also be found in [Bra03].



(a) Sandbagging against Overwhelming

(b) Sandbagging due to Seepage

(c) Sandbagging at the origin of the Seepage

Figure 3.2: Potential Applications of Sandbagging. *The figure shows different applications for sandbagging in the sense of dike defense. In (a), the dike is protected against overwhelming. That is, the increasing water level is assumed to get greater than the height of the dike. Subfigure (b) shows the problem of moisture penetration of the dike. In the consequence of a leakage at the landside, the sandbags are put on the appropriate position. In contrast, Subfigure (c) shows the positioning of sandbags at the origin of the seepage to avoid an ongoing moisture penetration.*

4 Algorithms

This chapter presents algorithms for solving the mentioned problems of managing a dike of sandbags which protect an area against flooding. Firstly, the preliminaries and assumptions of the environment are presented. Secondly, different methods of the identification of the waterline are shown, depending on the equipment of the used nodes. Then, solutions for some rough coordination to enable the decision, in which direction an incoming message must be sent, are given. Finally, different algorithms for building a reliable and energy efficient communication backbone are presented, each combined with a special routing process.

4.1 Preliminaries

On developing algorithms for sensor networks, some preliminaries and assumptions must be taken. Hence, this section gives an overview of the behavior of the nodes on which the below presented algorithms are based on.

To simplify the communication model, it is assumed that there are only bidirectional connections. If node u is able to communicate with node v, node v in turn is able to communicate with node u. Furthermore, nodes communicate only via broadcasting messages, and thus there are no unicast connections. Consequently, if a node sends a message, each of its neighbors receive it. This abstraction can be used for a simple recognition of message loss by sending and forwarding messages. Thus, if a node broadcasts a message, it should receive the same one again when the neighbors in turn broadcast the message.

To allow for a definite identification of nodes, each node in the network is equipped with an unique ID. Furthermore, by taking this assumption into account, the sent messages can also be made unique, at least by identifying them over the ID of the initial sender and a consecutively numbered value.

Although each node can be uniquely identified, it is not assumed that one sends a message to an arbitrary other one. Instead, it must only be assured that messages can be sent to a given base station. This is essential for the routing process inside the communication backbone, because the main focus is set on the construction of the backbone and the routing of messages along the structure.

Moreover, the nodes are not aware of their global position, so that the designed algorithms must operate in a strict local manner. This is assumed, because equipping the sensor nodes with localization hardware like GPS is both too expensive and energy wasting. Also the use of only a few position aware nodes and well known localization algorithms is not a suitable option, because by reason of ranging errors in distance measurement the results are imprecise and not satisfying. In addition, that would cause an unnecessary message overhead and thus an avoidable waste of energy.

The nodes themselves can have two different behaviors in the matters of soaked sandbags. On the one hand, in an absence of water resistance, they malfunction by the first occurrence of wetness. On the other hand, if they are watertight, they survive the wetted sand, but just have a very limited communication range in the consequence of physical restrictions.

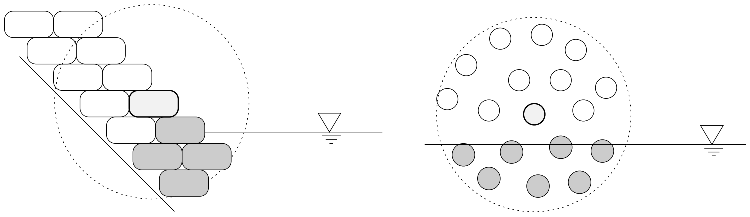
4.2 Waterline Identification

The first discussed problem is the identification of the waterline. Depending on the sort of used sensors, there are different approaches for solving this problem. If the nodes are equipped with appropriate sensors like wetness or temperature ones, they are able to handle the resulting readings. Otherwise, in case of the absence of such sensors, the decision must be taken without any information about the environment. That can be, for example, the consideration of the characteristic of a limited communication range due to soaked sandbags or the enclosing waterline.

4.2.1 Neighbor Loss

This approach assumes that the nodes are not equipped with any sensoring hardware, and either malfunction or lose their communication ability on soaked sandbags. Especially the former assumption can be reasonable for energy-saving causes, because each type of sensor consumes energy and can be seen as an expense factor.

The basic idea is indicated in Figure 4.1. A node which is able to identify the waterline has lost the connection to at most 50% of its neighborhood.



(a) Cross Section (b) Front View

Figure 4.1: Cross Section and Front View of a node identifying a neighbor loss. *The above Figures show a typical situation of an arbitrary node that should identify a noticeable neighbor breakdown. Due to the fact that the waterline has not yet reached the appropriate node, there are obviously less than 50% of the neighbors below the waterline.*

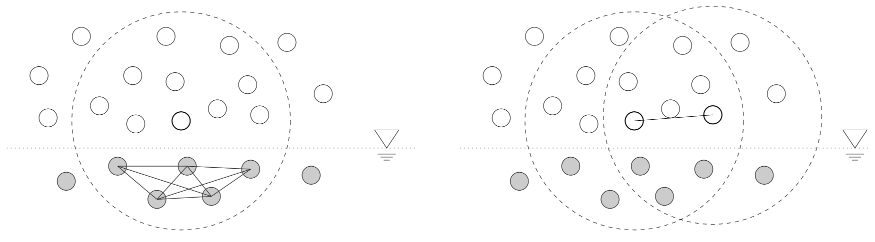
Taking this information into account, one must elect a valuable threshold for the decision whether a node is near the waterline or not. On the one hand, if the threshold is too high, a node runs the risk of getting soaked itself, and thus is no longer able to communicate with its neighbors. On the other hand, in case that the threshold is chosen too small, the decision process is very fault-prone due to malfunctions of actually healthy nodes, and a consequently false positive identification of the waterline.

The idea is to strike a balance between these two alternatives, set the threshold to 30%, and establish additional constraints which are presented below.

Eliminating probably Fault-Prone Nodes The first restriction is defined by the assumption that a node loss in consequence of an increasing waterline occurs in a much smaller time interval than a node loss due to arbitrary malfunctions. A node loss in turn is recognized, if the last activity of a neighbor is greater than the given time period $t_{min(LA)}$. Hence, it is required that each node guarantees a maximal idle time with respect to the sending of messages, and thus sends dummy messages in case of need. After that, a value for the time period $t_{max(LA)}$ must be given, which defines a maximal limit for the elapsed time since the last activity of a node. Both thresholds $t_{min(LA)}$ as well as $t_{max(LA)}$ depend essentially on the assumptions of the terms and conditions of the environment. The former value can be linked to the given maximal idle time t_{idle}, and thus a node is assumed to be dead if $t_{min(LA)} > 2 \cdot t_{idle}$. The latter threshold, $t_{max(LA)}$, depends on the mean alteration rate of the water level for a certain period during a flood, and thus can not be given exactly in this part. Of course, choosing this threshold too small would lead to rejecting regular malfunctioned nodes by a medium-term established water level. Otherwise, by allowing too wide space of time, this restriction would have no effect.

The second restriction takes the problem of a node loss due to a lack of energy into account. It is obvious that a node which is run out of energy can be taken for one below the waterline. For this purpose the threshold e_{min} is adopted, in order that all malfunctioned nodes for which a lower last energy level is known, are ignored in the decision process. Due to the definition of the energy level as well as the chosen value of the above mentioned idle time t_{idle}, it is set to 10% of the mean initial energy level of the sensor nodes.

Relationships of Lost Nodes As already mentioned above, nodes can not distinguish between an arbitrary malfunction and a wetness based node loss. The next idea to scale the probability of false positives down is to consider relationships between fancy nodes. That is, if one node gets lost, in all probability at least one of its neighbors gets lost, too. The idea is indicated in Figure 4.2 (a).



(a) Relationships of Lost Nodes (b) Neighborhood-Based Decision

Figure 4.2: Waterline identification with the aid of the neighborhood. *The left Figure shows the idea of “Relationships of Lost Nodes”. Nodes below the waterline should have the characteristic that a certain number of their neighbors are below the waterline, too. Taking this information into account, an upside node can differentiate between an arbitrary malfunction and wetness based node loss. The Figure on the right shows another approach, namely “Neighborhood Comparison”. That is, a node recognizing a noticeable node loss should verify that other nodes in its neighborhood have also recognized a similar breakdown.*

Hence, assuming a relatively small neighborhood density of about 15 to 30, the threshold must be chosen rather reserved. Thus, a positive decision with respect to a near waterline requires at least two neighbors which in turn had got at least one lost node as a neighbor, too. Additionally, all considered nodes must be different. For example, if a node has the connected neighbors p_1 and p_2, as well as q_1 and q_2, which have all got lost, it is $\{p_1, p_2\} \cap \{q_1, q_2\} = \emptyset$.

Neighborhood Comparison The next restriction regards neighbors which are still alive. The basic idea is that if a node decides to be near the waterline, some of its neighbors should have done so, too. An indication is shown in Figure 4.2 (b).

Again, the appropriate threshold must be chosen reserved. It is required that at least two neighbors have identified a significant node loss in the consequence of the above mentioned restrictions, too. In addition, two of the considered neighbors must be located on different sides. This is approximated by the requirement that two nodes are not allowed to be in the neighborhood of the respectively other one.

In general, the *Neighborhood Comparison* needs the adoption of an additional temporary state. At first, all nodes have not identified the waterline. If the former mentioned restrictions are fulfilled, a node changes to be a temporary accepting one, and only if the here described comparison succeeds, the waterline is finally identified.

Characteristics The *Neighbor Loss* method is a very simple one. There are only a few requisites needed, such as a nearly reliable losing of communication ability to nodes below the waterline and the knowledge of the 2-hop neighborhood. Moreover, the nodes do not need any type of sensor which would lead to a relevant energy consumption.

Nevertheless, the approach has some mentionable drawbacks. First, the requirement of the knowledge of the 2-hop neighborhood leads to a significant message overhead as well as additional memory consumption. Moreover, the nodes decide only locally, which could lead to false positives and should be taken into account in the below described backbone algorithms.

4.2.2 Local Measurement

The idea of this approach is that the nodes are equipped with capable sensors to identify a nearby waterline. Like the previous method *Neighbor Loss*, the *Local Measurement* decides only locally whether to be near the waterline or not, and works as follows.

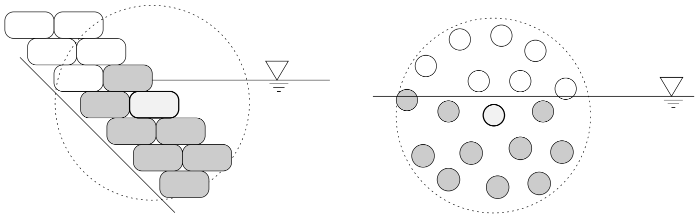
Each node measures periodically its surrounding environment, and compares the results to the definition of a nearby waterline. If the given parameters match, the waterline is identified and the node notifies its neighbors by broadcasting an appropriate message. Due to the probable occurrence of measurement errors, the decision process must handle those uncertainties by taking only the local available information into account. On the one hand, nodes which had not identified the waterline although they should do so, must recognize this and set themselves to be near or below the waterline. On the other hand, false positives must be eliminated as reliable as possible.

The nodes compare the own measurements with the ones of their neighborhood and accept the waterline only if certain conditions are fulfilled. For this purpose each node is able to accept the waterline temporarily before the decision process is finished. The ideas are presented in detail below.

States of the Nodes For enabling a temporarily acceptance of the waterline, each node can take the three different states *Rejected*, *Interim*, and *Detected* with respect to the identification of the waterline. In the beginning all nodes start as *Rejected*. If one measures a nearby waterline, it sets itself to be an *Interim* and checks its local neighborhood. If the conditions for an acceptance are fulfilled, it changes to the state *Detected*.

Neighborhood Matching As mentioned above, a node decides only locally whether to accept the waterline or not, and analyzes therefore the own state as well as the ones of its neighborhood. The decision process must handle two potential types of errors. First, false positives, that are nodes identifying the waterline although they should not, must be detected and avoided. Secondly, nodes below the waterline must recognize this albeit their sensor has not detected such a situation.

Figure 4.3 shows an example of a node below the waterline. Due to the already passed water level, the node itself and at least 50% of its neighbors should have identified the waterline, which again can be used in the decision process.



(a) Cross Section (b) Front View

Figure 4.3: Cross Section and Front View of a node which measures a nearby waterline. *The above Figures show a typical situation of an arbitrary node that should identify and accept the nearby waterline. Due to the fact that the waterline has reached the appropriate node, there are obviously more than 50% of the neighbors below the waterline, too.*

At first, the situation of nodes below the waterline is described. There are two different possibilities to be handled. If the node has identified the waterline itself and has set its status to *Interim*, there must be a minimal amount of neighbors which have done so, too. As shown in Figure 4.3, in theory there are at least 50-60% positives in the local neighborhood. Depending on the assumed fraction of faulty sensors, the threshold for an acceptance of the waterline must be reduced. Thus, the minimal fraction of *Interim* for an appropriate node is set to 30%.

The other situation for nodes below the waterline is that one has not identified this event itself, but should do so. This case is handled by a relatively high threshold of neighboring nodes which have already changed their status to *Interims*. The requirement for the high threshold is based on the situation, that a node above the waterline changes its status due measurement failures in its neighborhood, or a nearby location to the actual water level. Especially the latter case can be seen in Figure 4.1 by swapping the lost nodes with *Interim* ones. Thus, to avoid such a situation, the appropriate threshold should be set to at least 60% of the neighborhood. Here, a fraction of 75% is required for an acceptance of the waterline without having measured an appropriate occurrence.

Secondly, there are the opposite cases of nodes above the waterline, which should not identify it in the consequence of a concentrated occurrence of measurement errors. Such a situation is covered by the above given thresholds, but is repeated and discussed shortly from the other point of view. Hence, if the node is located above the waterline, but has a positive sensor return value, it needs at least 30% of the neighboring nodes which have identified the waterline, too. Otherwise, if the local sensor has not given a wrong measurement, a node would need 75% of neighboring nodes with an appropriate state.

Both cases, especially the former one, can lead to false positives primarily in the near of the waterline. Such a situation can be ignored, because an appropriate false positive would be at least in communication range to the accurate identifying ones. Otherwise, such wrong detections are accordingly improbable due to the need of a deep sensor malfunction density, and should be considered by the constitutive algorithms.

At last, the idea is summarized in Table 4.1 by showing only the relevant attributes and thresholds.

Sensor State	Minimal fraction of positive neighbors
INTERIM	30% of the neighborhood
REJECTED	75% of the neighborhood

Table 4.1: Decision thresholds for changing the state to DETECTED.

Example Application: Wetness Measurement The previous description has assumed an appropriate sensor for an identification of the waterline. That can be, for example, a wetness measuring which returns the actual relative moisture content of the surrounding sand.

Characteristics The decision of a nearby waterline is again taken strict locally, but unlike the previous approach there is only the 1-hop neighborhood needed, which in turn causes a smaller amount of messages that need to be sent for coordination issues. Moreover, methods using sensoring hardware should be safer. One drawback is the decreasing communication ability by an increasing moisture. Another drawback is the error-proneness in the sense of local failures (1-hop neighborhood).

4.3 Rough Location Awareness

For building a communication backbone and a consequential routing method, the nodes must dispose of a rough localization awareness. In consequence of the mentioned lack of the knowledge of either a relative or absolute global coordinate system, the localization procedure must be done distributed. As a result of the potential application area, the primary requirements on such algorithms are robustness and simplicity, whereas precision can almost be neglected. This can be postulated, because the later discussed backbone and routing algorithms should be as reliable as possible, and thus do not require an exact coordination system. Moreover, energy efficiency reasons by means of preferably a small amount of messages, and the occurrence of hardware failures, especially by measuring distances, come to the front.

However, the localization awareness is divided into two parts, that are described below. At first, the adjustment at the long side of the dike is presented, followed by the adjustment in the cross section.

4.3.1 Adjustment at the Long Side of the Dike

By routing messages through the dike or along the waterline inside the communication backbone, nodes must be able to forward an incoming message to proper neighbors.

For this purpose, there are so called *Relative Reference Points* built, to which the nodes are able to orientate themselves. A reference point in turn preferably is a base station, but can also be an arbitrary node in the network. The orientation is done on the basis of the minimal hop count to such references by considering the fact that there are neighbors with a lower hop distance on the one side, and ones with a greater distance on the other.

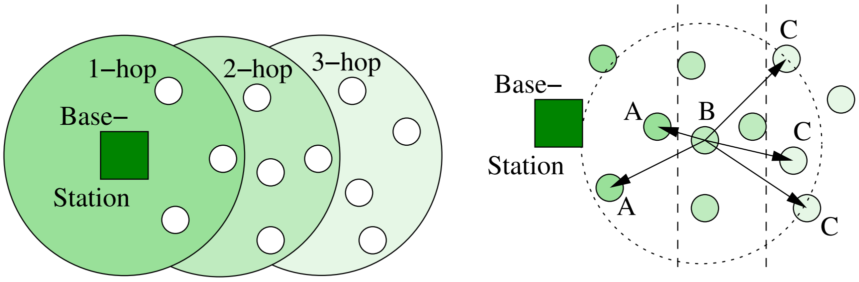
Initial Coordination Phase

In the initial phase each base station sends a COORDINATION message which contains the unique ID of the station, the information that the flooding has been started at a base station, and a hop count of 0. A node that receives such a message increments the hop count by one, and forward it, only if there is no earlier hop count known for this base station, or the received one is smaller than the existing one. Additionally, in case of forwarding the message, it stores the incremented hop count as the distance to the reference point.

Moreover, each received message is used for updating the minimal hop distances of the local neighborhood. Thus, the hop count of the appropriate neighbor to the reference point is set, if it is either not already known or smaller than the existing one. This case must be mentioned separately, because in case of a forwarding of the message there are nodes whose minimal hop count is greater than the own one, but can not be ignored. Figure 4.4 shows an example of the initial coordination phase.

In general, there should be only a few base stations existent in the network. Especially short after the construction phase of the dike it is even possible that there is no base station available. In the consequence of the simple procedure of the coordination process it is also supposable to elect an arbitrary node in network to act as a reference point. For this purpose an additional threshold is introduced. Each node is required to know at least MIN REF POINTS reference points. If not reached, a node can start the flooding of a COORDINATION message by its own.

Such a case requires the use of two additional thresholds. First, the probability of doing so must be defined, followed by setting the interval for checking the threshold



(a) Flooding started at a base station

(b) Rough direction awareness of nodes

Figure 4.4: Relative Reference Points. *The left Figure shows an example of a base station that starts the flooding of a coordination message. Each receiving node stores the minimal hop count, and forwards the message if needed. On the right Figure the nodes are already configured. If node B receives a message from one of the As, it can easily forward the message to one of the Cs. Nevertheless, there are also nodes which cannot be allocated.*

MIN REF POINTS. The former one is chosen by keeping in mind that the network consists of at least tens of thousands of nodes, and that the average connectivity should be between 10 and 20. Hence, it is set to $\frac{1}{10000}$ in the beginning, but is doubled after each step of the check to avoid a hardly ever selected reference point in possibly smaller areas. The other threshold that defines the interval of the check, is set relative to the time, a potential COORDINATION message would take to reach nodes in a distance of 30 hops.

Selection of Reference Points

If a side decision is taken, one of the collected reference points must be selected. There are two criterions that affect the result. On the one hand, the number of neighbors knowing a reference point has an influence, because the number of eligible alternatives is greater the more nodes take part in the decision process. On the other hand, the hop distance to the reference point is essential as shown in Figure 4.5.

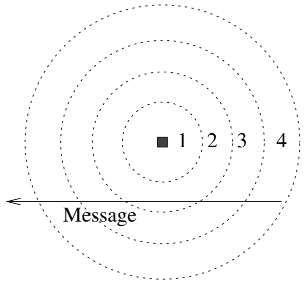


Figure 4.5: Direction Decision in the near of Reference Points. *The figure illustrates the problem of direction decision in the near of a Reference Point. Assuming that a potential message should be routed along the Reference Point, a decision that involves only the hop count is problematic, because the routing process can not be done.*

Depending on the height of the dike and the location of the reference point, an adequate threshold *min_ref_point_distance* for the minimal distance to a potential reference point that is taken into account may vary. Here, it is set to 10 hops. However, all reference points that have a smaller hop distance than *min_ref_point_distance* are dropped from the decision process. The remaining ones are ordered by the number of neighbors that know the appropriate reference point, and the one with the highest number of neighbors is selected. In the probable case that there are identical results, the hop distance comes to the front again by selecting the reference point with the greatest hop count. At last, if there are still identicals, the higher ID wins.

Characteristics

The rough localization by reference points is a very simple, but also robust and energy efficient method, and thus fulfills the given requirements. It does neither need any global localization information nor a base station, because the nodes are able to coordinate themselves. Furthermore, there are only the coordination messages needed which are flooded over the network once per reference point. Additionally, the message overhead is limited by the given flood limit.

4.3.2 Adjustment in the Cross Section

In contrast to the previously discussed adjustment at the long side of the dike, the nodes should also be able to coordinate themselves in the cross section of the dike. Again, robustness, simplicity, and energy efficiency are much more important than precision.

Taking the Waterline into Account

In the consequence of the requirement for an strict local adjustment that works without any localization hardware, it is useful to take an external reference point into account, so that the nodes do not need to perform a boundary detection that would even have the demand for operating in the three-dimensional space. Thus, a simple but sufficient idea is to perform the adjustment with the aid of the waterline, and is indicated in Figure 4.6.

In general, each node determines the minimal hop count to the waterline, but there are two additional restrictions that must be considered. On the one hand, a potential false positive in the means of the waterline identification can occur, and thus could influence the adjustment. On the other hand, a possibly fallen waterline would lead to a need for incrementing the minimal hop count.

However, nodes that have identified the waterline start with a hop count of 0, and send a WL HOP COUNT message by setting the appropriate value. A neighbor that receives the message stores the included hop count as well as the ID of the sender. Analog to the description in the *Neighbor Loss* method in Section 4.2.1, the lowest hop count that has been sent by at least 30% of the neighborhood is accepted, followed by the broadcasting of a new WL HOP COUNT message.

Moreover, in case of a fallen waterline, nodes that had already identified the waterline and an appropriate hop count of 0, can not identify it anymore. Thus, those nodes will update their hop count to 1 in due time, and again broadcast a WL HOP COUNT

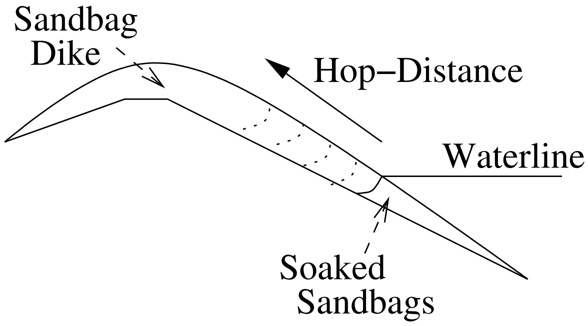


Figure 4.6: Adjustment in the Cross Section. *The figure shows the cross section of a potential sandbag dike and the idea of the adjustment. By taking the waterline into account, the nodes are able to determine the minimal hop count, and thus build the appropriate relationship to each other.*

message. The receivers only update their neighborhood, and the overall minimal hop count is incremented by and by.

Refinement of Hop Count

So far, the adjustment has only be done by means of the hop distance to the waterline of the 1-hop neighborhood. Additionally, by taking the information of the 2-hop neighborhood into account, the adjustment can be stated more precisely. Figure 4.7 illustrates such a situation.

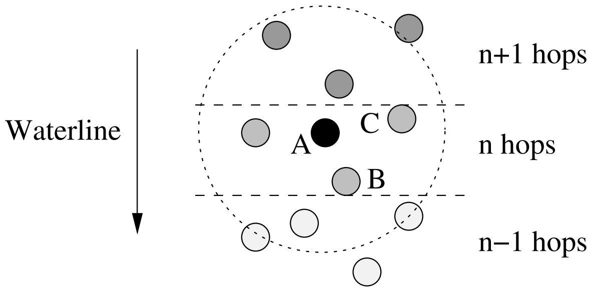


Figure 4.7: Closeness decision by 2-hop neighborhood. *The figure shows a potential situation of a closeness decision by taking the information of the 2-hop neighborhood into account. Node A is able to differentiate between the nodes B and C, that have both a hop distance to the waterline of n. Hence, node C has a greater number of neighbors with a hop count of $n+1$ than node B. The other way around, node B is located nearer to the waterline than node C in the consequence of a greater number of neighbors with a hop count of $n-1$.*

Characteristics

The adjustment in the cross section by taking only the waterline into account takes advantage of its robustness as well as simplicity. In addition, the message overhead can be significantly reduced, because the only sent information is the hop count, and thus can also be added to messages sent anyway.

4.4 Communication Backbone

This section presents algorithms for building a communication backbone near to the waterline, which should be done as reliable as possible on the one hand, and in a mostly energy efficient manner on the other hand. It is assumed that the waterline has been successfully identified, and a rough location awareness with respect to the direction decision is available.

4.4.1 Stripe next to the Waterline

The idea of the stripe next to the waterline is to build a rough communication backbone by a very simple election process, and improve and maintain the existing one in the ongoing iterations.

Building the Backbone

Each node decides strict locally, and thus without any communication overhead, whether it belongs to the backbone or not. The decision is taken only on the basis of the distance to the waterline measured in hops. So, given a minimal as well as a maximal hop distance to the waterline by MIN WATERLINE HOPS and MAX WATERLINE HOPS, each node between these thresholds sets itself as a part of the communication backbone, and broadcasts an appropriate notification to its neighbors. Furthermore, all nodes doing so set their internal state to PURE BACKBONE STATE, which is used later in the description of *Dynamics*. The thresholds define the closeness to the waterline as well as the thickness of the backbone. Figure 4.8 indicates a situation as a part of the network in the near of the waterline.

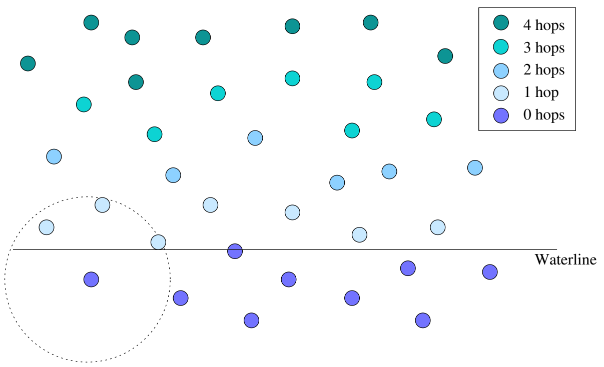


Figure 4.8: Stripe: Hop distance to waterline. *The figure shows a cutout of the network in the near of the waterline. Each node has received the minimal hop count to the waterline, and thus is able to decide whether it is part of the backbone or not.*

As shown, nodes in 1-hop range to the waterline are in the very near, and run the risk of a lose of the communication ability accordingly. Hence, MIN WATERLINE HOPS is set to 2 hops to offer a sufficient closeness to the waterline as well as enough reliability due to malfunctions in the routing process. Furthermore, MAX WATERLINE HOPS is set to 3 hops, resulting in a tolerable small backbone.

In the consequence of potential irregularities in the topology, the resulting backbone could contain holes, and thus may not be continuously connected.

Closing Holes

Either the strict definition of the backbone affiliation by the hop distance to the waterline, or sparse connected parts of the topology linked to close-by thresholds MIN WATERLINE HOPS and MAX WATERLINE HOPS can lead to so called holes. That is, there are multiple parts of the backbone that are not connected to each other. Thus, additional rules must be defined to avoid such situations.

For this purpose, each node needs a given minimum BACKBONE CONNECTIVITY on successors and predecessors which are neighbors of that node, and also part of the backbone. If a node do not fulfill the above requirement, it must start a request in the appropriate direction. Choosing a too great threshold for BACKBONE CONNECTIVITY would lead to plenty of request messages, because the nodes hardly satisfy the wanted connectivity. Hence, BACKBONE CONNECTIVITY is set to 2, so that each node should have at least two successors as well as two predecessors.

There are two supposable situations. On the one hand, a node in the near of MAX WATERLINE HOPS may know nodes in the local neighborhood that are not already part of the backbone. On the other hand, in case of unavailability of such a neighbor, a potential connection must be requested. Both cases are described separately.

Local Request At first, it is assumed that there are neighbors available which are not already part of the backbone. Each node checks in given periods of time BACKBONE CONNECTION CHECK, which is chosen in subject to the average lifetime of the sensor node, the amount of successors and predecessors. If one is missing (in case of an absence of more than one, the procedure is repeated automatically until the given amount is reached), the appropriate node starts a BACKBONE JOIN REQUEST to the wanted direction. The request is sent to exactly one neighbor that does not already belong to the backbone, has a greater hop-distance to the waterline than MIN WATERLINE HOPS, and is located as near to the waterline as possible. Of course, the value could also be set to MAX WATERLINE HOPS, because nodes between these thresholds are part of the backbone. Furthermore, the selection of nodes that are preferably near to waterline is done in two steps. At first, all nodes with the minimal hop-distance are elected. Second, if there is more than one in the resulting set, the appropriate nodes run through an additional election process. Hereby is the one selected that has the most neighbors with a minimal hop-distance. Thus, the neighbor that is the closest to the waterline is chosen. The node that receives a BACKBONE JOIN REQUEST sets itself as a part of the backbone, and broadcasts a notification of its joining.

Furthermore, all nodes doing so set their internal state to REQUESTED BACKBONE STATE. In case that the actual hop-distance to the waterline changes, and is between the thresholds MIN WATERLINE HOPS and MAX WATERLINE HOPS, it changes to the state to PURE BACKBONE STATE. As already mentioned above, the sense of doing so comes to the front later.

Global Request In case that there is no eligible neighbor available to extend the backbone structure as needed, a request must be started which is able to cover multiple hops.

An appropriate node therefore sends a BACKBONE EXTENSION REQUEST message that contains the ID of the requesting node, a time to live, a list of node IDs that contains only the one of the initiator in the beginning, and the actual LOCATION INFORMATION. The latter contains the location of the initiator as well as the wanted direction. Table 4.2 gives an explicit overview of the content.

BACKBONE EXTENSION REQUEST	
Initiator	Node ID
Location	LOCATION INFORMATION
Path	List<Node IDs>
TTL	Integer

Table 4.2: Contents of BACKBONE EXTENSION REQUEST.

The request is flooded over a local area, and the first node which fulfills the given LOCATION INFORMATION answers an acceptance.

If a node receives a BACKBONE EXTENSION REQUEST, it adds its ID to the path list in the message, and broadcasts it again. In addition, it stores the unique ID of the message, and drops all further receiving ones. As already described in Section 4.1, the message can be identified, for example, by the ID of the initiator and a consecutively numbered value. Moreover, to avoid an undesirable extension below the MIN WATERLINE HOPS threshold, an appropriately located node drops each received request.

In case that a node fulfills the given LOCATION INFORMATION of the message, it answers a BACKBONE EXTENSION FOUND message. This message is routed backwards to the initiator along the path that is contained in the request. Each node, including the answering one, sets itself as part of backbone. When necessary, the state is changed to REQUESTED BACKBONE STATE. Hence, a potential connection from the initiator to the answering one is assured.

Dynamics

In case that the hop-distance to the waterline changes, the backbone in turn must change its structure by joining and disjoining of nodes.

At first, the disjoining of nodes is discussed. If a node is already part of the backbone, and the threshold MIN WATERLINE HOPS gets greater than the actual hop-distance to the waterline ACTUAL WATERLINE HOPS, the node must prepare its leaving from the backbone. Therefore it broadcasts a BACKBONE LEAVE REQUEST, and stores the point in time TIME LEAVE REQ STARTED of that request. Each neighbor that receives such a request, and knows at least one alternative node that is located on the same side as the leaving one, answers a BACKBONE LEAVE ACCEPTED. In case, that one does not belong to the backbone, the request is ignored and not responded. Hence, if the disjoining request was accepted by each neighbor that is part of the backbone, the appropriate node leaves the backbone and notifies its neighbors by a BACKBONE DISJOINED message, which in turn update their neighborhood. Alternatively, if not enough BACKBONE LEAVE ACCEPTED messages arrive, the node disjoins anyway after a given time period TIME FORCE BACKBONE LEAVE which is verified against the above mentioned TIME LEAVE REQ STARTED. In the consequence of the relative short amount of time of a potential new BACKBONE JOIN REQUEST at a neighbor, the threshold TIME FORCE BACKBONE LEAVE can be chosen comparatively short with respect to the duration of sending messages. Thus, by keeping in mind that a neighbor may start a new join request, the threshold TIME FORCE BACKBONE LEAVE is set to $10 \times t\_msg\_send$.

If a node recognizes an affecting change of the hop-distance to the waterline at the upper side of the backbone that requires a disjoining, in other words ACTUAL WATERLINE HOPS gets greater than MAX WATERLINE HOPS, and the internal state is equal to PURE BACKBONE STATE, it prepares for disjoining the backbone. In this case the difference between PURE BACKBONE STATE and REQUESTED BACKBONE STATE appears, because the *pure* ones form the primarily backbone structure, whereas the *requested* ones connect parts of the core structure, and thus have a different priority of leaving the backbone. However, in case of the disjoining of a *pure* node, the process works identical to the above mentioned method. Again, the appropriate node is broadcasting BACKBONE LEAVE REQUEST, waits for the replies, and disjoins by sending a BACKBONE DISJOINED, if either TIME FORCE BACKBONE LEAVE is elapsed or all relevant neighbors have answered. Contrary, a node in the state BACKBONE REQUESTED is part of an explicitly requested connection between two groups of the primarily backbone structure, and thus must ensure the maintenance. It would be problematic, if all nodes of such a bridge try to take the structure nearer to the waterline concurrently. Hence, the process is done iteratively. If such a node recognizes a decreasing waterline by a smaller minimal hop-distance than the actual one, it sets the flag BACKBONE TRY LOWER which in turn is checked periodically. In case that the flag is set, and the node has the highest minimal hop-distance to the waterline of all neighbors that are in the state BACKBONE REQUESTED, it starts the disjoining phase as already described above by sending a BACKBONE LEAVE REQUEST, and leaves the backbone in any case on a run off of TIME FORCE BACKBONE LEAVE. If two nodes have an equal high hop-distance n, the one that has more neighbors with a hop-distance of $n + 1$ wins the choice. At last, if also this comparison is equal, the one with the higher ID starts the disjoining. All other nodes that dropped out of one of the comparisons, increment the number of drop outs DROP OUT CNT, and start the same process at the next check of the flag BACKBONE TRY LOWER. Finally, in case that a maximal number of trials MAX LOWER TRIALS is reached, the flag is reset.

The above possibilities have discussed the disjoining of nodes. However, it is also possible that nodes are going to join the backbone due to an appropriate change in the hop-distance. This happens if the thresholds get to MAX WATERLINE HOPS $\geq$ ACTUAL WATERLINE HOPS and MIN WATERLINE HOPS $\leq$ ACTUAL WATERLINE HOPS. In that case the procedure is very simple, because the concerning node only sets its state to be part of the backbone, and broadcasts a notification to the immediate neighbors. After that, the conventional periodical check for the number of eligible successors and predecessors starts.

Routing inside the Backbone

The routing process is a very dynamic and on-demand one. For this purpose a node decides as recently as a message arrives, to which neighbors the message should be forwarded. Each forwarding takes the message closer to the sink by selecting an eligible neighbor, and thus makes use of the direction awareness. The election process is done repeatedly due to the kind of criterion, so that there are dynamically different nodes elected. A more detailed description follows below.

General Behavior As already mentioned above, the path of a message is built completely dynamic and on-demand, respectively. Thus, each time a message must be forwarded, a node takes into account a list of eligible neighbors, and elects a receiver by a specified rule. The election process is given in detail later.

Furthermore, to enhance the reliability of the routing algorithm, messages can be sent on multiple paths. Therefore the initial sender sends the message to a given number of neighbors, whereas each of these nodes act as a starting point of an individual path. As a consequence of the multiple paths and the election process, it is supposable that the same message is sent to one node several times. In such a case the appropriate node tries to forward the message to different receivers.

The Election Process The election process is responsible for the selection of a receiver from the list of eligible neighbors given by the direction algorithm. As a result of afore sent messages, each node is being aware of relevant information about its neighbors, such as the energy level, the distance to the waterline, or whether a neighbor belongs to the communication backbone or not. On the basis of this information it is possible to develop different election rules, each with a special priority, depending on the wanted kind of routing process. If, for example, sensor nodes malfunction on water contact, each node could elect the nodes which have the smallest distance to the waterline, because these are the ones which will get lost. Otherwise, if the nodes are water resistant, each node could elect the nodes with the highest energy level to enable a long-living and dense backbone, which should be still available after a falling waterline. Table 4.3 presents some simple election rules.

As mentioned above, the NBW can be used, if the nodes are not water resistant and the backbone should be as close to the waterline as possible to exhaust the nodes with the highest probability of dying. To apply this rule, the waterline detection must provide distance measurement.

Instead, the FFW routes messages to nodes that are as far away from the waterline as possible, but still part of the communication backbone. This can be used if the nodes are water resistant and takes advantage of the circumstance that the more soaked a sandbag is the more unreliable the appropriate communication behavior is.

As a similar rule there is the LM which elects the nodes of the communication backbone with the lowest measured moisture. Again, this is done to avoid wetness based message loss.

Otherwise, if there is a certain amount of reliability in message transmission guaranteed, one can use the GH to elect the nodes with the greatest measured humidity.

To increase the average lifetime of all nodes belonging to the backbone, in case of the possibility of detecting a falling waterline, one can use the HEL. For this purpose

Election Rule	Acronym	Short Description
Near by the waterline	NBW	Select nodes as near by the waterline as possible
Far from the waterline	FFW	Select nodes as far from the waterline as possible
Lowest Moisture	LM	Select nodes which have measured the lowest level of moisture
Greatest Moisture	GM	Select nodes which have measured the greatest level of moisture
Highest Energy Level	HEL	Select nodes having the highest energy level
Lowest Energy Level	LEL	Select nodes having the lowest energy level
Longest Distance	LD	Select neighbors with longest distance
Most Neighbors	MN	Select nodes with the most neighbors
Random	RND	Select nodes randomly

Table 4.3: Possible Election Rules

the election process chooses the nodes with the highest energy level in each pass, so that the nodes should keep a similar energy level.

The opposite rule of the HEL is the LEL, which chooses the nodes with the lowest energy level. Applying only this rule at a pass takes no advantage to the mentioned rules so far, but it is predestinated as a secondary one in case of applying multiple rules per pass.

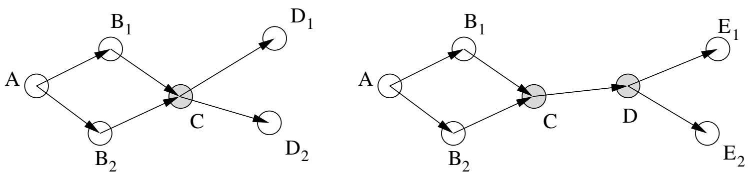
Another approach is to forward messages as fast as possible to the sink, and is described due to the LD. By choosing the nodes, e.g. neighbors, with the longest distance, messages arriving at the source should have taken the smallest number of hops.

To exhaust the most unimportant nodes, there is the chance to elect the nodes with the most neighbors (MN), because in case of a loss of those nodes due to too much energy consumption, there should be enough remaining alternatives.

At last, the random rule (RND) should be mentioned. If one do not want to use a specific rule, or there is not enough information about the neighbors available, the appropriate nodes are elected randomly.

As indicated in the description of the LEL, there is no necessity of applying only one of these rules. On the one hand, multiple rules can be used one after the other, for example at first a fixed number of NBWs, and on a given part of these ones the LEL. On the other hand there are weighted as well as random approaches. The latter means, that a certain probability is assigned to each rule. So, for example, the NBW is taken with a probability of 60% and the LD with a probability of 40%, whereas the decision is repeated in every election process. Instead, the weighted approach assigns a quantifier to each used rule. At first the eligible neighbors are passed through simple ranking methods, followed by a multiplication of each resulting value with the assigned weighting. At last, the best rated neighbor is elected.

Maintenance of Multiple Paths As already indicated in the description of the *General Behavior*, it is supposable that multiple paths are built over one single node, which would lead to overlapping paths, and thus results in a bottleneck of the routing process. An example for a supposable situation is shown in Figure 4.9.



(a) Single Overlapping Path (b) Multiple Overlapping Path

Figure 4.9: Overlapping Paths in Routing Process. *This figure shows the problems of overlapping paths in the routing process. In both subfigures the nodes B_1 and B_2 send the same message to node C which in turn tries to fork the multiple paths again. In Subfigure (a) on the left this is done directly by forwarding the message to nodes D_1 and D_2, whereas Subfigure (b) shows the pass of responsibility to the next node, here D.*

To break up such bottlenecks, each node provides a message history that includes the message ID, the node ID of the sender, and the timestamp of the reception. Furthermore, the neighbor to that the message is routed is stored. Thus, on receiving a message that must be forwarded, nodes are able search the history for an already sent message, and exclude the appropriate node from the election process. Only if there are no eligible alternatives, the message is sent to this neighbor (as had been shown in Figure 4.9 (b).

In the consequence of the establishment of a message history, it must be defined how long those messages are stored. By taking into account that the paths are started almost concurrently, and thus duplicated messages are received in an relatively short time span, messages in the history can be cleaned quite shortly. Hence, the threshold MAX HISTORY AGE is set to the time a message would take to cover 5 hops.

Dead End In the consequence of the way of closing holes in the backbone structure a node must not have immediate successors, but provides a path to an ongoing location as described in *Global Request*.

In such a case the message is equipped with the received path, and is routed along the containing nodes. In addition, if one of those nodes is no longer available, a new request can be started that works analogous to the above described one.

Characteristics

The building of a communication backbone as a stripe next to the waterline was designed for providing a preferably simple coordination phase that needs as few as possible messages to be sent. Thus, the definition of a backbone affiliation depending on the hop distance to the waterline has been set, by adding simple methods of maintenance. In the consequence of the lack of coordination inside the backbone, the routing process has been set up for a load balance of the forwarded messages.

4.4.2 Short Path Mix Up

The idea of the *Short Path Mix Up* is to build arbitrary short paths in the near of the waterline. Each path is thereby connected to at least one other, and thus build the basis for the routing process, because messages can be sent along the paths. Figure 4.10 indicates the idea of this approach.

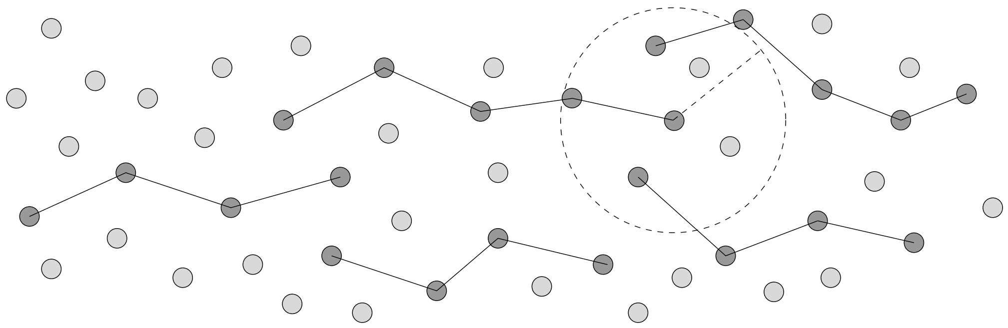


Figure 4.10: Example of Short Path Mix Up. *The figure shows a potential situation of already built short paths in a part of the topology. Each end of a path is connected to at least one other. Thus, the result is one single, continuous connected path.*

At first, the basic structure of the paths must be built, followed by a refinement phase to close potential holes, and the consideration of dynamics by means of an alteration of the waterline. At last, the routing process must be described. The details are presented below.

Building the Backbone

In the beginning the paths must be built from scratch. That means, no node in the network already belongs to a path in the initial phase. In addition, the coordination between the nodes must be done strict locally. However, the idea is to build a rudimentary infrastructure in the near of the waterline. Randomly selected nodes start to build paths that are restricted by a maximal path length MAX PATH LENGTH. The refinement of the result as well as the closing of holes is described later in the according subsection.

Election of Initiators At first, the initiators of the built paths must be elected by appropriate rules. It must be assured that the resulting paths are located relatively near to the waterline. Furthermore, only a small subset of nodes should initiate a path building request to reduce the density of created paths, and thus avoid an unnecessary message overhead in the consequence of the coordination between multiple initiations.

However, each node that is not already part of a path checks in given intervals TIME CHECK INITIATION, if it fulfills the conditions for the initiation of a new path. That is, nodes must be located in a given hop distance to the waterline, which is bounded by the thresholds MIN WL HOPS and MAX WL HOPS. Furthermore, nodes are not allowed to have a neighbor that is already part of a path. This is postulated to reduce the number of initial path requests, and thus to provide only a rough structured backbone in the beginning. A refinement as well as the improvement to a continuous connected path is done later on.

If the conditions are fulfilled, a node starts a PATH BUILD REQUEST with a given probability that depends on MAX PATH LENGTH as well as the neighborhood density of each node, and is given by $p = \frac{1}{\text{MAX PATH LENGTH} * density}$. If so, the request is sent to one specified neighbor that is elected by a simple process as follows. At first, all neighbors that are located between the thresholds MIN WL HOPS and MAX WL HOPS, and for which a definitely direction decision can be made[1], are taken. Then, one of those neighbors is selected randomly.

The PATH BUILD REQUEST message contains the unique ID of the initiator, a list of IDs to which each receiver of the message adds itself in case of joining the path, the maximal length of the path, and the actual hop count the message has been taken. The unique ID is used to identify the created path by setting the BB PATH ID to the one of the initiator. In addition, a LOCATION INFORMATION is given. That can be, for example, the reference points as described in Section 4.3.1, and the designated direction the path should take. Table 4.4 illustrates the contents of such a message.

PATH BUILD REQUEST	
Initiator	Node ID
Location	LOCATION INFORMATION
Path	List<Node IDs>
Max Length	Integer
Actual Hop Count	Integer

Table 4.4: Contents of PATH BUILD REQUEST.

At last, after sending the PATH BUILD REQUEST to the appropriate neighbor, the initiator sets its state to BB TMP PATH that describes a temporary, but not finally accepted path. The state is changed to the accepted one BB ACC PATH, if the initiator receives a PATH BUILD ACCEPTED. In addition, the node stores the timestamp of the path request. Thus, to avoid that it stays in that temporary state in case of a malfunction on another node, the initiator resets the state, if the given time TIME WAIT REQ ACC has passed without receiving an acceptance. The threshold is set proportional to MAX PATH LENGTH, because the potential answer takes at most $2 \times$ MAX PATH LENGTH hops. Finally, the initiator sets its role in the path to BB ROLE ENDPOINT which is set in contrast to BB ROLE LINK on the nodes that connect two endpoints.

Processing the Initial Request If a node receives a PATH BUILD REQUEST there are several cases that must be considered. So, for example, depending on the actual state such requests are either accepted or rejected. Furthermore, an acceptance must be specified. The cases are described in detail below.

- *A node that receives a* PATH BUILD REQUEST *is not already part of a path, and consequently its state is neither* BB TMP PATH *nor* BB ACC PATH. In the consequence of the fact that the node is not already bound to a path, it accepts

[1] In case of using the *Relative Reference Points* from Section 4.3.1, all neighbors with a different hop count to one reference point than the own, are elected.

the request and changes to the state BB TMP PATH, and sets the path identifier BB PATH ID to the ID of the initiator that is contained in the PATH BUILD REQUEST message.

If the maximal path length has not yet been reached, and there is no endpoint of a path in the neighborhood, the actual path is extended. Therefore the node adds its ID to the path in the PATH BUILD REQUEST, increments the actual hop count, and forwards the message to an eligible neighbor. In addition, the state is set to BB TMP PATH and the role to BB ROLE LINK, respectively.

Otherwise, if the maximal path length is reached, or the appropriate node has a neighbor that is an endpoint of a path, the actual path request is finalized. That means, the node sets its state to BB ACC PATH and the role to BB ROLE ENDPOINT, respectively. Moreover, the node sends a PATH BUILD ACCEPTED message back along the temporary path by using the appropriate information that can be taken from the request message. In addition, the message also contains the above described LOCATION INFORMATION. Each path member that receives the acceptance changes the state to BB ACC PATH (the role has already been set to BB ROLE LINK), and forwards the message. Finally, the acceptance message arrives at the initiator which also changes the state and already has the role of an endpoint. Hence, the path is created.

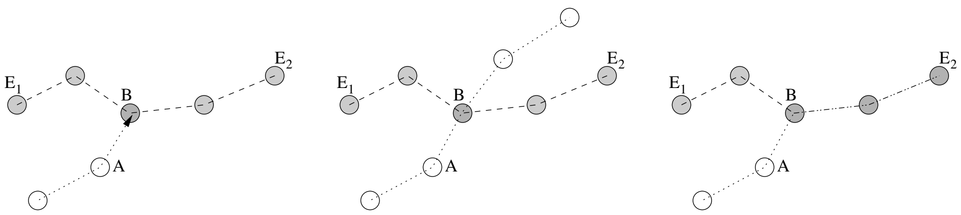
- *A node that receives a* PATH BUILD REQUEST *is already part of a path, and consequently its state is either* BB TMP PATH *or* BB ACC PATH. *Moreover, the role of the receiver is* BB ROLE LINK. Under these circumstances the path request is rejected by sending a PATH BUILD REJECTED back to the sender. Thus, the node that receives the rejection is able to try to find another path. Nevertheless, the building of the path can be forced, but such a situation is described later in an extra subpoint.

- *A node that receives a* PATH BUILD REQUEST *is already part of a path, and consequently its state is either* BB TMP PATH *or* BB ACC PATH. *Moreover, the role of the receiver is* BB ROLE ENDPOINT. In this case, the behavior depends on the length of both paths. If the actual path length plus the length of the requested path is greater than MAX PATH LENGTH, and the request contains at least two nodes, the requesting one is told to finalize its path by sending a PATH FINALIZE message back. Thus, the result are two endpoints that are already in communication range.

  Otherwise, if the maximal path length is not exceeded, or the request contains only one node, both paths are joined. Of course, the latter condition could result in a path length longer than allowed, but is accepted due to simplicity matters. However, an endpoint that receives such a request sends a PATH JOINING message back by adding the knowledge of the existing path. Furthermore, it sends a PATH UPDATE along the path it already belongs to, and thus informs the other endpoint about an update of the path.
  Furthermore, the node is going to change its role from being an endpoint to a link. If there is no other endpoint in the neighborhood, the change of role is done without further circumstances. Otherwise, if there already is at least one endpoint connected, the node is a link as well as an endpoint depending on the

point of view, and fulfills the role of a BB ROLE HYBRID. This is an unfortunately solution. Thus, a node that fulfills both roles, tries to cancel the existing connections by sending a CANCEL CONNECTION REQUEST to the appropriate endpoints. Those in turn try to fulfill the request by either simply cancelling the connection (if permitted by the threshold MIN CONNECTIONS that is described later in the subsection about *Closing Holes*) or starting a NEW PATH REQUEST (that is also described in *Closing Holes*). If successful, an endpoint answers a CANCEL CONNECTION ACCEPTED and cancels the connection. In case that the initiating node has received the acceptance by all relevant endpoints, all connections are canceled, and the role is changed to be only a link.

- *A node receives* PATH BUILD REJECTED. Here, a node has sent a path build request to a neighbor that is already part of a path. At first, it marks the neighbor as a path member. Next, it tries to select an alternative neighbor to send the build request to. If the latter is not successful in the consequence of a lack of alternatives, or all potential nodes are already path members, the request is forced. Therefore the node elects one of the marked neighbors, and send a PATH BUILD FORCE message. The node that receives such an enforcement, can react in two different manners depending on the actual situation, as shown in Figure 4.11.



(a) Force Path Build (b) Cross-Over Paths (c) Overlapping Paths

Figure 4.11: Force a Path Building. *The Figure shows the enforcement of a path building. In Subfigure (a), node A sends an enforcement to node B which is already part of a path. On the one hand, if there is an eligible node in B's neighborhood, B proceeds the path building request. The result are two paths with B as an cross-over point as shown in Subfigure (b). On the other hand there is the situation in Subfigure (c). Here, node B joins the two paths, and acts as a branching or merging point depending on the point view. Hence, all nodes from B to the endpoint E_2 are part of two paths.*

If the enforced node has at least one eligible neighbor for proceeding the path building request, it forwards the request to the appropriate neighbor. This is done by ignoring the maximal path length contained in the message, and thus can result in a path that is at one greater than the limit. However, the node is part of both paths and acts as a cross-over point. To distinguish one path from the other, it stores the appropriate information such as the unique path ID, the role it takes, and so on, separately.

Otherwise, if the request can not be forwarded in the consequence of a lack of potential neighbors, the new path is joined to the existing one. Thus, the node acts as a branching point, or the other way around, as a merging point of the two paths. Again, the information about both paths is stored separately. To finalize the path request, a PATH BUILD ACCEPTED that contains the information about the already existing path is sent back to the requester. In addition, the appropriate part of the existing path is updated by sending a PATH NEW OVERLAP message that contains the information of the requested path. The message is routed along the path up to the endpoint, and each receiving member creates a separated path information for the new one.

General Behavior of Forwarding Path Requests In general, there are preferably neighbors chosen that are located between the thresholds MIN WL HOPS and MAX WL HOPS with respect to the hop distance to the waterline. If such a election is not possible, and there are alternatives with a hop distance greater than MAX WL HOPS, the nearest one (with respect to the waterline) is chosen. Otherwise, if the path request occurs in the near of the waterline, and there are only alternatives with a hop distance less than MIN WL HOPS, none of them is chosen due to reliability matters.

Arranging the Endpoints In case that endpoints of different paths are in communication range, each endpoint must know the direction of the connected paths. Figure 4.12 gives an overview of the situation.

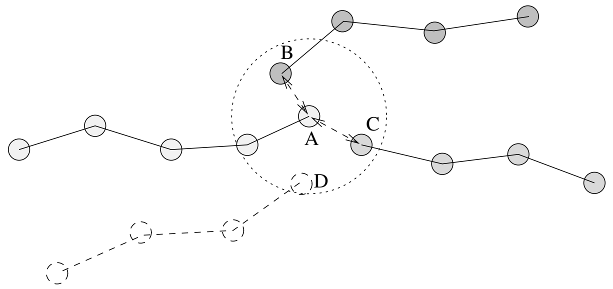


Figure 4.12: Arrangement of Endpoints. *The figure shows several paths that are connected over their endpoints. Due to the fact that the other ends of the paths are located in different directions, it must be decided which connected paths are going forwards, and which ones are directed backwards. Thus, by exchanging the information about the respective ends of the paths, node A is able to decide that nodes B and C are endpoints of proceeding paths, whereas node D is a backward one.*

As described in the section about the processing of the initial request, each endpoint knows the respective location information of the other endpoint of the path. Thus, if a new endpoint has been recognized, they exchange a PATH TARGET message that contains the appropriate location information.

As-Is State after Initial Phase After the initial phase has been finished, there are several paths available, and each node in the given hop distance to the waterline is in range of at least one path. The paths in turn may not already been connected to each other, so it is possible that there are endpoints which do not know any succeeding paths.

This happens, because the initial phase does not provide a continuous connection, and thus is discussed in the section about *Closing Holes*.

Moreover, all paths can be identified by a unique path ID that is equal to the one of the initiator. If the initiator gets lost, or the path is divided in the consequence of a malfunction of a member, the path must be restructured. Amongst others, such cases are discussed in the section about *Dynamics*.

Closing Holes

As already mentioned above, the short paths may not be continuously connected. That can be, for example, because the initial phase only provides a rough structuring to reduce the number of needed paths. Moreover, the path building starts only in a given hop distance to the waterline, and in the consequence of potentially missing parts in the dike structure, there is an appropriate hole in the backbone.

General Process The basic idea is that each endpoint needs at least one other endpoint in its communication range to build an ongoing connection as already described above and shown in Figure 4.12.

If the required number of connections MIN CONNECTIONS is only set to one, the overall result could be a single path subdivided in short ones. On the other hand, if the minimum is set to two or greater, there would result too much requested paths. Thus, each endpoint selects the appropriate threshold randomly. To provide an adoptable mixture the probability for one and two, respectively, required connections is set to $p = \frac{1}{2}$ each. More than two required connections are not used.

If an endpoint recognizes that the threshold MIN CONNECTIONS has not been reached, it elects one neighbor in the appropriate direction, and sends a NEW PATH REQUEST. The neighbor in turn is a new initiator of a path, and thus the building works as already described above. The NEW PATH REQUEST forces a node to initiate a new path. Hence, the restriction of the given distance to the waterline is not used, and it is possible to create paths outside the defined area.

Special Case: Path Finding The *General Process* assumes that there are neighbors in the direction of a path request. If not, an appropriate node can not extend the backbone although there may be a possible path. Figure 4.13 shows a potential situation.

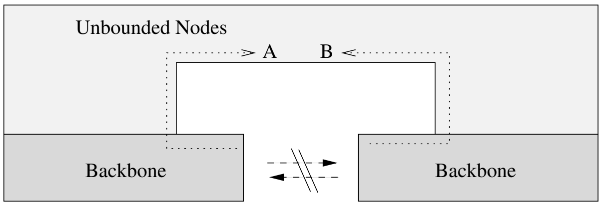


Figure 4.13: Dead End of Backbone. *The figure shows a supposable situation for the creation of a communication backbone. If the backbone is built strict locally, that means it is extended one by one, the structure may be caught in a dead end, although there would be an alternative path as illustrated by the points A and B.*

Such a situation occurred already in the *Stripe next to the Waterline*, but is repeated here by reason of completeness. Thus, such a case must be solved by allowing parts of a path to run in the opposite than the wanted direction. The idea is that an appropriate node without an eligible successor floods a request over the network. If there is any node that receives the request and is located in the wanted direction, it answers an acceptance. The initiator in turn can start a new path build request after receiving such an acceptance. A more detailed description follows.

If an endpoint wants to extend the backbone due to a lack of connections, it starts a PATH SEARCH REQUEST by adding its ID and the actual LOCATION INFORMATION. Moreover, a TIME TO LIVE (TTL) is added as well as the capacity for storing the path the message has taken. The structure of the message has already been shown in Table 4.2, but is repeated in Table 4.5.

PATH SEARCH REQUEST	
Initiator	Node ID
Location	LOCATION INFORMATION
Path	List<Node IDs>
TTL	Integer

Table 4.5: Contents of PATH SEARCH REQUEST.

The PATH SEARCH REQUEST is flooded over the network, and each node that receives the message adds itself to the containing path, before forwarding it. In addition, such a node stores the ID of the message, and discards every further receiving one. Assuming that such messages are not sent very often, the TTL and the time TIME HOLD SEARCH REQUESTS of storing the IDs can be chosen relatively high. Thus, the TTL is set to 100 hops (but depending on the supposed size of the topology it can be chosen greater, of course), and TIME HOLD SEARCH REQUESTS proportional to the TTL.

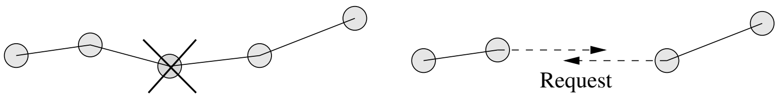
In the consequence of the added LOCATION INFORMATION a receiving node is able to decide whether it is located in the wanted direction with respect to the given location or not. If so, the backbone can be extended and the appropriate node sends a PATH FOUND message back to the initiator by adding the so far covered path on which the message is routed back, and the own LOCATION INFORMATION. After sending the PATH FOUND, the node discards all other messages with the same ID.

The first PATH FOUND message that is received by the initiator (there can be more than one, of course) is used for building an appropriate path, whereas later received ones are ignored. However, the initiator knows a potential path for extending the backbone, and thus sends a PREDEFINED PATH BUILD message that contains the received path of the PATH FOUND message, and is routed over these nodes. Each node that receives the PREDEFINED PATH BUILD adds itself to the path, and sets the state to BB ACC PATH.

Dynamics

The case of dynamic behavior in the network can occur in several matters. At first, path members may malfunction, and thus the broken path must be either fixed or splitted in two new ones. Then, in the consequence of a variation in the backbone structure, endpoints of paths that are connected to each other should check if it is possible to join both paths. And lastly, the waterline may vary, and either new nodes must join the backbone or existing nodes leave.

Malfunction of Path Members At first, the malfunction of path members is discussed. The basic idea is that a node which recognizes a lost neighbor in the path, starts a new PATH BUILD REQUEST in the appropriate direction. Such a situation is shown in Figure 4.14.



(a) Path that loses a member (b) New request due to a lost member

Figure 4.14: Malfunction of Path Members. *If a path member gets lost as shown in Subfigure (a), both nodes in the path which recognize the malfunction start a new* PATH BUILD REQUEST *in the appropriate direction as illustrated in Subfigure (b).*

In the consequence of the way creating the path as already described above in Subsection *Building the Backbone*, each path member knows the nodes as well as the order of those, and thus send a PATH BUILD REQUEST by adding the nodes of the still existing path to satisfy the MAX PATH LENGTH. If the malfunction occurs to both sides of a path member, the PATH BUILD REQUEST is sent in both directions.

In addition, if a endpoint gets lost, the connected ones recognize a missing path, and thus start a NEW PATH REQUEST.

Path Joining It is not required that a path is composed of exactly MAX PATH LENGTH members. Hence, there may be several very short paths that should be joined if possible.

Each endpoint knows the length of the connected paths. If the added lengths are not greater than MAX PATH LENGTH, the one with the higher ID sends a PATH JOIN REQUEST message to the connected endpoint. The condition of the ID is postulated to avoid a concurrent sending of join requests. The node that receives the message, sends a PATH JOIN ACCEPT back, changes its role to be a link, and starts a PATH UPDATE on the former existing path. On receiving the acceptance, the appropriate endpoint also performs the change of role as well as the PATH UPDATE.

In case that a endpoint of the joining process was connected to at least one other path, it fulfills both roles BB ROLE ENDPOINT and BB ROLE LINK as already described above with respect to the joining of paths after receiving a PATH BUILD REQUEST. Again, a CANCEL CONNECTION REQUEST is sent, by waiting for a CANCEL CONNECTION ACCEPTED. If the request succeeds, the role changes to be only a link. Otherwise, the node still fulfills both roles and is called a BB ROLE HYBRID.

In addition, the initiation of a join request must consider two special cases. On the one hand, an endpoint may have different alternatives of joining the paths. In such a case, the prospective longer one is chosen, followed by the decision for the higher ID on equality. On the other hand, an endpoint may receive more than one request.

Then, simply the first request is accepted, by rejecting the later ones with a CANCEL CONNECTION REJECTED message.

Variation of the Water Level The last discussed case of dynamics is the variation of the water level. In such a case, the minimal hop distance to the waterline may change, and thus either existing paths must be canceled or new ones created.

If a node's minimal hop distance to the waterline was greater than MAX WL HOPS, but changes to be less or equal, it is a potential new path member for the rough backbone structure. Thus, if there is no path in the neighborhood of such a node, it starts the initial phase as already described above. Otherwise, if there is at least on member in the neighborhood, the node waits for an incoming PATH BUILD REQUEST.

On the other hand, if the minimal hop distance of a path member changes to be smaller than MIN WL HOPS, it tries to leave the path and consequently the backbone structure. In the consequence of the assumption that an increasing waterline is noticed by many path members on the lower side of the backbone, it is omitted to let single nodes leave a path. Instead, the whole path of such an occurrence is canceled.

However, if a node is below the MIN WL HOPS range, it sends a PATH CANCEL REQUEST along the path to both endpoints to inform them about the increasing waterline, and thus the unreliability of the actual path. Of course, if an endpoint recognizes such a variation, the message is sent to the other one. The endpoints in turn try to cancel the whole path by sending a FORCE CANCEL CONNECTION to the connected paths, so that those ones are also informed about the potential unreliability. In case that there is an alternative path, an endpoint of the designated canceled path receives a CANCEL CONNECTION ACCEPTED by the appropriate connection, and forwards a PATH CANCELED message through the path. A member that has received such a message by both endpoints, leaves the path.

Routing inside the Backbone

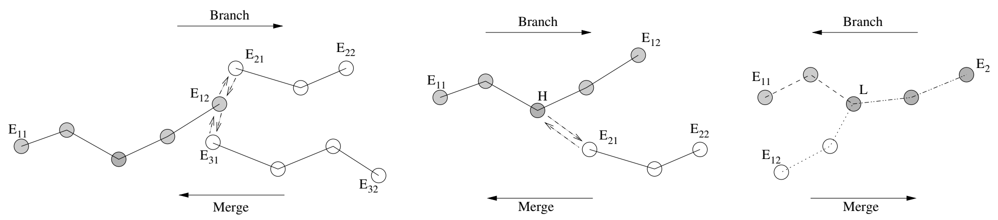
In the consequence of the structure of the backbone, the routing process can be simplified, because a continuous connection is provided by the short paths. Hence, routing is done by forwarding a message to the next path member, and from an endpoint to the appropriate connections.

Nevertheless, in consequence of the path building process there are some special cases that must be considered. An endpoint is generally supposed to be connected to multiple paths, and thus messages are branched and merged on those points. Moreover, each path member has specified successors and predecessors, so that the reliability of delivery of messages can be improved. Both cases are described in detail below.

Branching and Merging There are several cases in which a path member is connected to multiple nodes, and thus a node can either receive more than one of the same message, or must send a message to more than one neighbor. Figure 4.15 illustrates different examples.

The branching of messages is obvious, because an incoming message is sent to all connected nodes that are directed in the appropriate direction.

On the other hand, the merging of messages is also a simple and obvious method, but requires additional resources. Hence, a routing cache is used that stores the ID of



(a) Multiple Connections (b) Hybrid Path Member (c) Overlapping Paths

Figure 4.15: Branching and Merging in Routing Process. *Subfigure (a) shows a standard case of an endpoint E_{12} connected to two paths over the nodes E_{21} and E_{31}. Thus, if a message is sent from the left to right, E_{12} sends the message to both nodes E_{21} and E_{31}. The other way around, by sending a message from the right to the left, the same one is received two times on node E_{12}. Hence, by enabling a routing cache of message IDs, the second one can be discarded. In Subfigure (b) there is an analogous case, but in contrast to the former mentioned, the appropriate node H is a link as well as an endpoint. At last, in Subfigure (c) the focused node L is only a link, but also connects two paths.*

a forwarded message as well as the timestamp of the event. If the ID of a received message is found in the cache, the message is discarded.

Although the contents of the routing cache are very small, because it contains only the ID and a timestamp, old ones must be cleared after a given time. By taking into account the maximal path length MAX PATH LENGTH, each content with a timestamp older than the average time a message is transmitted over $2 \times$ MAX PATH LENGTH hops is deleted.

Improving Reliability of Message Delivery For improving the reliability of message delivery, it is postulated that there are periodic neighbor detection messages sent. If such a message is no longer received by a neighbor, the appropriate node is assumed to be dead. Furthermore, the interval is given by NEIGHBOR DETECTION INTERVAL.

To enable a message cache that contains the forwarded message and the neighbor to which it has been sent to. If there is another message received by that neighbor, it is supposed to be still alive, and thus the messages which are associated with the neighbor are deleted from the cache. Otherwise, if the neighbor is supposed to be dead after the NEIGHBOR DETECTION INTERVAL, there is a new path built as described above in the section about *Dynamics* and *Malfunction of Path Members*. After that has been done, the cache is cleared and the appropriate messages are sent along the new path.

Characteristics

The *Short Path Mix Up* has been designed for reliability reasons on the one hand, and dynamics on the other. The latter is provided by the shortness of the paths, so that the maintenance of each path is hold tolerably local to only a few nodes. The reliability on the other hand is assured due to the general path structure. That means, each node that belongs to the backbone has got at least one successor and one predecessor which allows for an improvement of the reliability of message delivery. Moreover, messages are sent over multiple paths to provide additional robustness.

5 Implementation Details

This chapter presents the used simulation environment as well as implementation details. At first, the used discrete event simulator `Shawn` is presented by a short overview. Then, the additionally implemented visualization framework is introduced, followed by details about the implementation and environment of the designed algorithms.

5.1 Simulation Environment

The algorithms were implemented and simulated using the discrete event simulator `Shawn`, which has been designed for simulating large wireless sensor networks on an algorithmic point of view. For portability reasons it is written in Standard C++, which in turn allows the usage on different platforms, but at least UN*X and Windows systems providing a recent compiler version. Moreover, the object-oriented design structure leads to an easy extensibility, and thus a comfortable possibility of implementing the designed algorithms.

However, as already mentioned above, the central point of `Shawn` is the simulation of very large sensor networks containing up to hundreds of thousands of nodes, as well as the principal functionality of conceptualized algorithms. This means that there is less focus on representing the physical layers of sensor nodes, or the support of real time applications. Instead, an abstraction of the design and functionality of algorithms comes to the front. The simulation process is divided into multiple iteration steps, each allowing a sensor node to do its work, send messages, and receiving messages which have been sent in previous iterations.

As a rough description of the representation of the simulated world in `Shawn`, the topology consist of a given number of nodes. Each node in turn is able to contain multiple processors, which are able to execute distributed algorithms on the network. In addition, it is possible to implement so called *simulation tasks* that allow a central point of view onto the network, and can be executed either once or repeatedly on given points in time.

When the simulation is started, the boot method of each node is called, which in turn let the associated processors start their initialization phase. The same process is done in each iteration step by executing the working phase of the nodes, and thus the appropriate procedure of the processors. In addition, it is possible to send messages that are broadcasted, and received by each neighbor which is located in the given communication range. The decision whether a neighbor is in range can be done by several communication models which are available in the core library. Message delivery in turn is simulated by different transmission models. The available models vary from a reliable one that delivers each sent message, over a random drop model that simulates a random packet loss, up to several MAC layer models.

In contrast to the simulation of the nodes in each iteration, it is also possible to execute simulation tasks. They are started either before or after one step, and can additionally be added to the repeatedly executed *pre-* or *poststep-tasks*. Such a task has access to the whole simulation environment, and can be used, for example, to iterate through all nodes and collect wanted information or elect ones for special purposes.

The simulation environment provides additional features that can be used for the assistance of added extensions. With respect to the implementation that has been done for this thesis, a short description of the most important ones follows.

In the consequence of the general and basic interface that is provided by the classes of the core library, a *tagging* concept has been realized. This concept allows each class derived from the `shawn::TagContainer` to store additional arbitrary data. One can therefore add a *Tag* to such a class. The *Tag* is identified by its name, and contains an integer, a string, a reference to a `shawn::Node`, and further information. Each other class is now able to read the stored data, without knowing who has written the content.

Next, there are the *Readings*. A reading returns a defined value such as an integer or a double value for a given position in the simulated world. That can be, for example, the current temperature at the appropriate position, or with respect to this thesis the moisture penetration in a dike. In addition, a reading can also be used for computing the height at a given 2D-coordinate.

The logging ability is the feature presented at last. Selected classes such as a `shawn::SimulationTask`, `shawn::Processor`, and so on, are able to use additional logging libraries, and are associated with an own logger object. This leads to the possibility of customizing each logger separately. For example, a logger of an implemented processor can be completely disabled, or set to a wanted logging level.

The structure of the source code can be divided into two different parts. On the one hand, there is the core library which provides access to the whole simulation and is responsible for the fundamental functionality such as message transfer, execution of the iteration steps, and so on. On the other hand, there are different applications, or, as better said, modules, which can contain implementations of algorithms, separate extensions of the core library, or the like. By building the binary, each module must be enabled on purpose to be compiled. Hence, especially extensions that use external libraries should be implemented as modules.

5.2 Visualization Framework

For debugging purposes on the one hand, and the possibility of an appealing demonstration of the used algorithms on the other, the discrete event simulator `Shawn` needed to be extended by an visualization framework that allows a three-dimensional view on the simulations as well as running in real-time with respect to the current simulation instance. Thus, the issue was divided into three different tasks. At first, an adequate library for the visualization itself had to be selected and integrated into the simulator. Second, `Shawn` had to be extended to be able to allow multi-threaded applications. At last, a general and mostly simulation independent possibility of displaying different scenarios in `Shawn` had to be realized. A rough design overview is given in Figure 5.1.

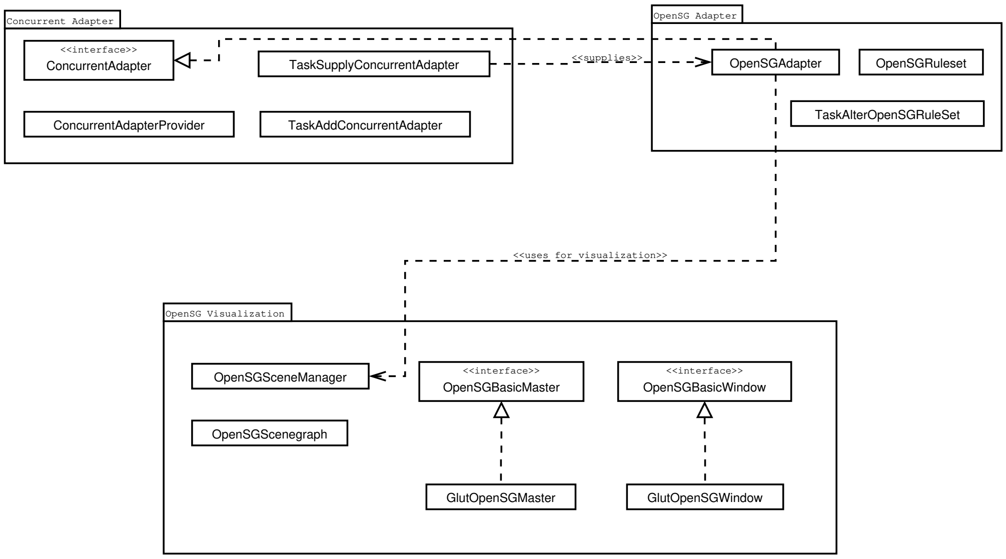


Figure 5.1: Visualization Overview. *The visualization framework consists of three different parts, each performing a separate function. The* `Concurrent Adapter`*, which is responsible for a multi-threaded, parallel execution of applications, is shown on the top left. The* `OpenSG Visualization` *at the bottom acts as an interface to OpenSG by implementing an own scene manager. The* `OpenSG Adapter` *enables the network topology of* `Shawn` *to be visualized by OpenSG, and is shown on the top right.*

The design goal of this solution was an almost independent realization of the several parts. Hence, the only purpose of the `Concurrent Adapter` is to provide an easy to use interface for enabling an arbitrary application to be executed in parallel to the simulation. Furthermore, the `OpenSG Adapter` connected with the `OpenSG Visualization` has been designed to allow to be easily replaced or appended by other graphic libraries.

A more detailed description of the different parts follows below.

5.2.1 Concurrent Adapter

As already mentioned above, the responsibility of the `Concurrent Adapter` is a multi-threaded execution of tasks. For a realization of this objective, there are generally two different basic concepts which have both its advantages and disadvantages. On the one hand, the core library of `Shawn` can be extended to handle multiple threads, that have all access to all the available data in every point in time. Such an approach would result in a very powerful, flexible, and almost unrestricted solution, but would also require a deep intrusion into `Shawn`'s system libraries. On the other hand, the multi-threading ability can be implemented as an additional module, which would be the more restrictive solution, but would also not need any change in the core library.

The decision has been made for the latter concept of implementing concurrency as a module, and is based on the following reasons. The most important argument is the requirement of an optionally use of the multi-threading functionality due to the need of an inclusion of an additionally external library for multi-threading support in C++ that also provides portability, by a coexistent lack of importance of concurrency in most simulations. Furthermore, as already mentioned in Section 5.1, the simulations run in multiple steps, which would require a synchronization of all threads at the end of each iteration by use of the concept of an integration in the core library. Additionally, it should be defined what happens on an alteration of the same data of multiple threads during one iteration. However, the module concept is the better one for this purpose, and is designed as follows.

At first, the multi-threading ability itself must be provided by a portable solution. For this purpose, there is the chance to implement this functionality either from scratch or to use an external library. Due to stability reasons, as well as not to reinvent the wheel, the latter has been done. Hence, by selecting a portable library for C++ that also provides object-oriented support, the decision has been made for ZThread[1],that is, for example, well documented in [EA03]. The most important class is `Runnable` which provides an interface to enable inherited classes to be executed as a thread. Those classes can be either run separately by creating a new `Thread` object, or handled by so called executors. Executors are responsible for the management of created threads, and thus reduce the coding overhead as well as acting as a supervisor. Furthermore, the ZThread library offers different types of mutexes and conditions for controlling the access of multiple threads on the same resource. Especially the usage of mutual exclusions is provided by an additional feature called `Guard`, which allows a safer as well as simpler handling of mutexes. The library offers much more possibilities of implementing multi-threaded applications such as *Thread Local Storage*, several types of executors, and so on, but the most important features are indicated above.

However, using the ZThread library, a multi-threaded extension for `Shawn` had to be written, and designed to be an optional module. As mentioned above, a simulation in `Shawn` is composed of multiple iteration steps, each allowing the nodes in the network to work, as well as sending and receiving messages. Hence, the idea is to create a `shawn::SimulationTask` that can be added to the simulation as a so called *pre-* and *poststep-task*, respectively, which is run before, or after, each iteration. This task in turn contains a provider that is maintaining the several applications. Figure 5.2 shows a class diagram of the implementation.

[1] http://zthread.sourceforge.net/

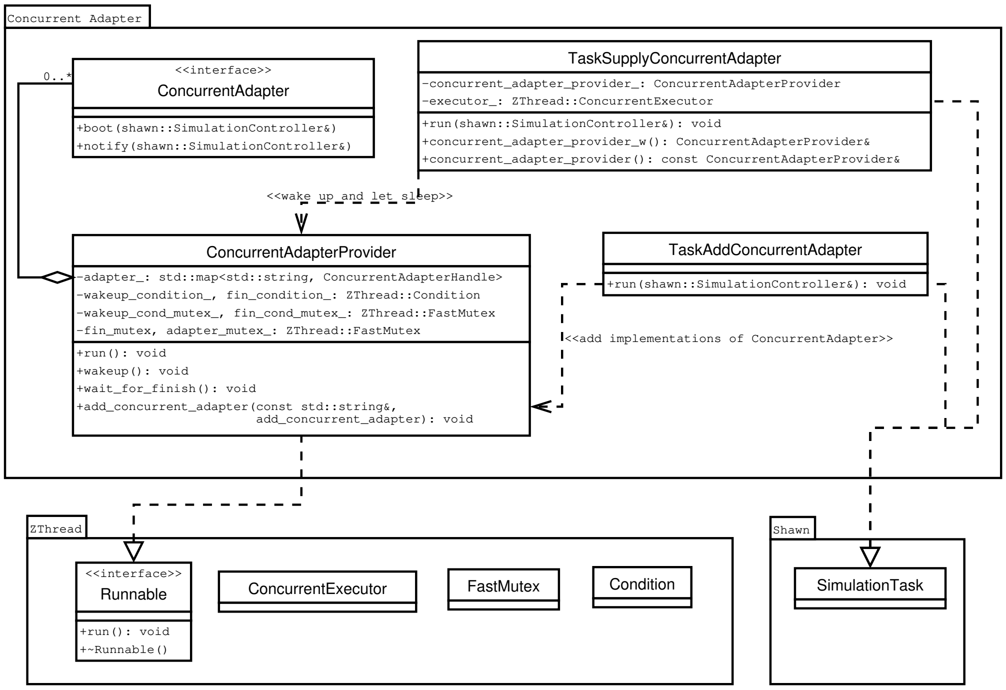


Figure 5.2: Class diagram of the Concurrent Adapter. *This class diagram demonstrates the functionality of the Concurrent Adapter. The* `ConcurrentAdapterProvider` *maintains implementations of the* `ConcurrentAdapter`*, which acts as an interface for potential applications. Furthermore, it is supplied by the simulation task* `TaskSupplyConcurrentAdapter`*. That means, each time this task is started, it calls the* `ConcurrentAdapterProvider` *to wake up, which in turn notifies the appropriate applications. At last, the simulation task* `TaskAddConcurrentAdapter` *is supposed to add those applications to the provider.*

Considering the above description, a potential application must implement the `ConcurrentAdapter`, and add the result to the provider with the aid of the `TaskAddConcurrentAdapter`. Moreover, the `TaskSupplyConcurrentAdapter` should be added as a pre- or poststep-task.

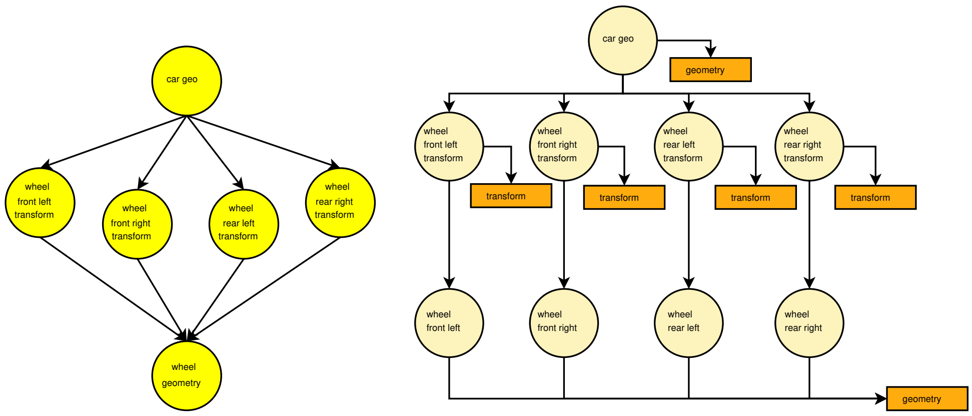
5.2.2 Visualization Library

By selecting a capable library for the visualization process, the decision had to be made out of several libraries. One of the most significant rating values was the demand on portability, so that the library can be used at least on Windows as well as Linux systems. Further important requirements were an Open Source license like the GPL, a relatively easy to use but potent interface, preferably a stable release, and a well documentation.

OpenSG

The choice had been made for OpenSG[2], a portable and Open Source scenegraph system using OpenGL, that provides a fully object-oriented C++ interface, and was designed, in contrast to other scenegraph systems, for multi-threading support. The advantage of scenegraph systems compared to other graphic libraries is the internal organization of the drawable components in a graph of connected nodes building a tree structure, which in turn allows a rise of performance in displaying scenes due to the efficient possibility in optimizing the data passed to the underlying OpenGL layer, as well as an easy handling of the data that should be shown by adding, removing, or modifying appropriate nodes in the graph. Especially the latter characteristic can be directly used for visualizing a simulation in `Shawn`, and is described in detail in Section 5.2.3.

However, OpenSG, in always the same manner like other scenegraph systems, hold the whole scene in a graph structure that allows multiple parents for nodes. An example is shown in Figure 5.3.



(a) Graph Structure with Multiple Parents

(b) Graph Structure of OpenSG

Figure 5.3: Graph Structures of Scenegraphs. *Subfigure (a) shows a typical graph structure of scenegraphs with multiple parents per node. For example, by representing a car that is built of a body and four wheels, the geometry of the wheels can be stored only once. By keeping the multiple graph structure in mind, OpenSG uses a more complex structure that is shown in Subfigure (b) on the right. Nodes are used only for a representation of the hierarchy of the graph, and contain potential children as well as a core in any case. The core in turn is the most important structure, and can contain the geometry, transformations, and so on.*

The scene in turn is maintained by a scene manager that holds the root node of the graph structure, and thus all geometries, transformations, and so on. Furthermore, lights and cameras, a navigation for an interaction with the user, and a window with an assigned viewport should be provided. In OpenSG there is the `Simple Scene Manager` that is designed for a simple beginning, and manages all the above men-

[2] http://www.opensg.org

tioned things. Nevertheless, in the consequence of the simplified design there are only the basic features provided, e.g. one single viewport per window.

For a more flexible usage of the available features, it is of course also possible to implement an own scene manager to avoid the above mentioned restrictions. For this purpose, one must create an own window and add one or more viewports, create lights and a camera, integrate a navigator, and enable the interaction between those.

However, the used window, either in the `Simple Scene Manager` or a self created scene manager, is generally based on an external library such as GLUT[3] or QT[4]. The former one is the simple alternative, but widely available and portable. It only manages a window in which the scene is drawn, and can handle user input via mouse or keyboard. The latter, QT, is a common and platform independent GUI toolkit that can be used for developing an interactive user interface.

Integration in `Shawn`

The integration of OpenSG in the discrete event simulator `Shawn` has been done by implementing an own scene manager, and additional classes for representing the scene itself as well as different windows that can be used for the visualization process. A class diagram of the implementation is shown in Figure 5.4.

The center of the integration process is the `OpenSGSceneManager` that holds all the important information for displaying a scene. Thus, it creates a `osg::WindowPtr` and connects a viewport to it, provides a camera and lights, and has access to the graph of the scene that contains the drawable objects like geometries, lines, and so on.

The `OpenSGScenegraph` in turn provides access to the scene, and thus can be used for adding objects that are supposed to be shown in the visualization. The appropriate methods use the structure of the underlying scene graph, and hold references to the important nodes in the graph. For example, if one want to add a node to the scene, there is a group as well as a geometry needed. The group represents the parent in the scene graph, and the geometry is the object that is drawn. Both are identified by their name and are looked up in the appropriate data structures. Hence, before adding a node, the required information must be added. In general, the whole access to the important information is done by the identification by names, and thus can be easy handled.

Moreover, the `osg::WindowPtr` that is provided by the `OpenSGSceneManager` must be shown on the screen. This is done, as already above mentioned, by an additional library. For this purpose, GLUT and QT are supported. The basic classes are `OpenSGBasicWindow` and `OpenSGBasicMaster`, and both are implemented for one of those libraries separately.

The window acts as an interface between the scene manager and the used library. The main purpose, beside the display on the screen, is the handling of user input. On the one hand, the scene manager must be notified if the user is moving the mouse, and thus changes the view of the scene. On the other hand, keyboard input can be forwarded and processed somewhere else. In this case, input is transformed in a `TransferMessage`, and added to the message queue of the connected master which in turn processes the messages in given points in time.

[3]http://freeglut.sourceforge.net

[4]www.trolltech.com

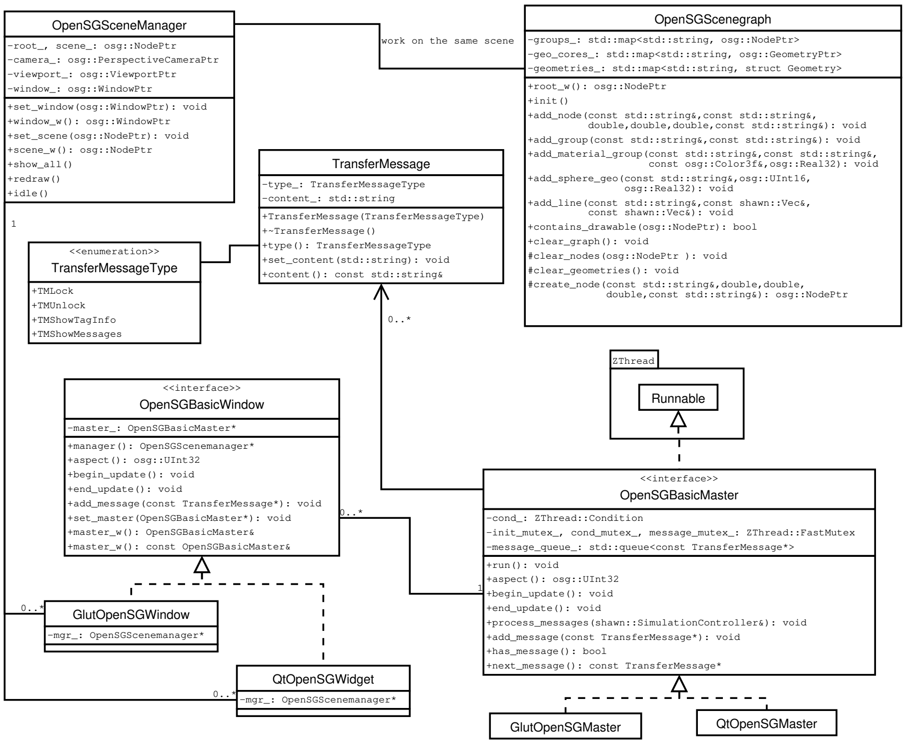


Figure 5.4: Classes of OpenSG Visualization. *The main classes of the integration of OpenSG in* `Shawn` *are the* `OpenSGSceneManager` *and the* `OpenSGScenegraph`*. The former one manages the whole visualization process by proving a window that displays the scene, creating a camera and lights, connecting the window to a viewport, and holding the scene that represents the simulated data. The scene in turn can be handled by the already above mentioned* `OpenSGScenegraph` *that provides methods to add nodes, geometries, lines, and so on. Moreover, different windows that are associated with an appropriate master can be created, either using GLUT or QT. In the consequence of the functionality of GLUT and QT, the master runs in an own thread, and provides access for a message queue that is responsible for the handling of user input.*

The master is responsible for mainly two kinds of functionality. At first, it is able to process the current message queue and the corresponding `TransferMessages`. This is done on calling the method

```
process_messages(shawn::SimulationController&)
```

which provides access to the actual running simulation in `Shawn`. Consequently, it has been realized that moving the mouse pointer on a node in the visualization associated with an appropriate keyboard input, shows either the actual state, or the received and sent messages of the node. Moreover, the current simulation can be paused for taking screenshots of the shown scene.

The second responsibility of the master is the management of the associated windows as well as initializing and running the main application of the used library. Both alternatives, either the `glutMainLoop()` in GLUT or `QApplication->exec()`

in QT do not return after calling. Consequently, the master is also a thread by deriving it from `ZThread::Runnable`.

5.2.3 Visualization of Simulations

The visualization of simulations uses both above described approaches, and provides a customizable control for the user. An overview of the classes and associations is given in Figure 5.5

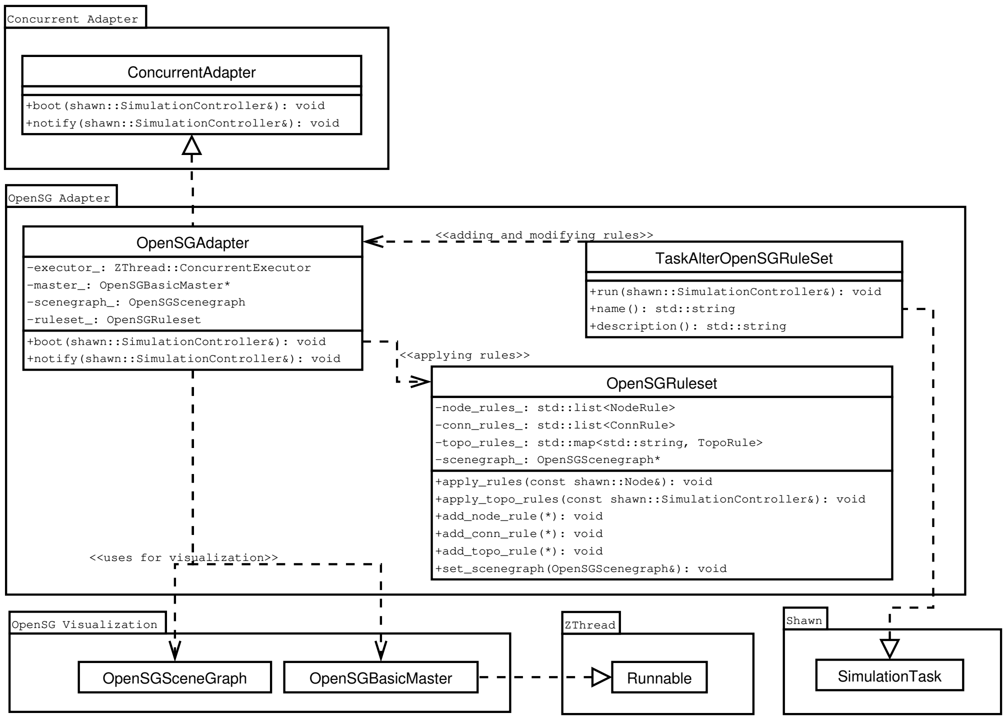


Figure 5.5: Classes of OpenSG Adaption. *The class diagram shows the idea of a customizable control of the integrated OpenSG. The* `OpenSGAdapter` *implements the* `ConcurrentAdapter`*, and thus is notified in each iteration step as already described above. In addition, it is connected to the implemented* `OpenSGScenegraph`*, and is able to add drawable objects to the displayed scene. The configuration of the visualization is done by rules which are provided by the* `OpenSGRuleset`*. A rule in turn defines, if and how parts of the simulation are drawn.*

At first, the `OpenSGAdapter` implements the `ConcurrentAdapter` for enabling the multi-threading support, and is added to the provider that in turn is supplied by the appropriate task as already described above in Section 5.2.1.

In addition, the adapter uses the integration of OpenSG as illustrated in Section 5.2.2. The important associations are the `OpenSGScenegraph` and the appropriate implementation of the `OpenSGBasicMaster`. The latter manages the display on the screen, and is told on each iteration to process its message queue. The scene graph in turn holds the data that is supposed to be shown, and provides methods to add objects to the graph.

Basic Concept

The objective is to display a running simulation in `Shawn` with the aid of the integrated OpenSG library, and hence enable a three-dimensional view on the simulated network. The main design goal was the realization of a solution that is completely independent of the used simulation scenario, and can be used by each implementation without modifying the source code for the visualization process. For this purpose, two conditions must be fulfilled. At first, the visualization of special properties of nodes is only allowed to use information that is provided by the core library of `Shawn`. Second, the customization must be done by a configuration file that is loaded when a simulation is started.

The idea is to manage the visualization by so called *Rules* which are used to decide whether an object is drawn or not. Furthermore, there are three different kinds of *Rules*: First, there are *Node Rules* that are applied on each `shawn::Node` which in turn represents a sensor node in the simulator. The second one, *Connection Rules*, are used to describe relationships between two nodes, or a node and its neighborhood. At last, *Topology Rules* represent structures in the network, and work with a `shawn::Reading`. Each type of *Rule* is described in detail later in this section.

The basic container is the `OpenSGRuleset` in which all used rules are stored. Moreover, the class got a reference to the `OpenSGScenegraph` for adding appropriate objects to the displayed scene, which in turn is done if one of the *Rules* can be applied.

For enabling the possibility of a flexible configuration, the contents of the `OpenSG Ruleset` are controlled by the simulation task `TaskAlterOpenSGRuleset` that allows the conception and modification of *Rules*. Thus, a customized visualization can be completely controlled by an appropriate configuration file.

However, to provide an insight into the functionality of the rule concept, and a rough overview of how they are used, a more detailed description of each kind of rule as well as the idea of integration follows.

Building a Graph

As already indicated above, the *Rules* are used to decide, whether an object is drawn on the screen or not. But before those *Rules* can be applied, it must be defined what kind of object is supposed to be drawn on success. Again, the idea is to allow a maximal amount of flexibility to the user by a concurrently increasing complexity.

All available data is hold in a graph structure that corresponds to the one that is used by OpenSG. Hence, the user is able to create different types of nodes that are identified by their name, and are supposed to have an existing parent in the graph. A typical node can be, for example, a group node or a material. Group nodes are only used to enable the possibility of creating a hierarchy in the graph, but are not needed in the visualization process. In addition, the user can also create material nodes. These ones define the used material of a drawable object, for example the color and level of transparency. At last, there is also an existing root node that must be used for creating the whole graph structure.

In addition, it must also be possible to draw different kinds of objects. Remember, OpenSG distinguishes between *nodes* and *cores* in its internal structure. A *core* can be added to a *node*, and may contain the geometry of the drawn object. Again, this structure is adopted to the customizable visualization framework in `Shawn`. The user is able to create several kinds of geometries, e.g. a sphere or box, that are again identified by their name.

At last, after creating the graph structure and different geometries, the rule concept is used for displaying objects on the screen. If a rule can be applied, the associated object is drawn by using the associated material. The whole idea is indicated in Figure 5.6.

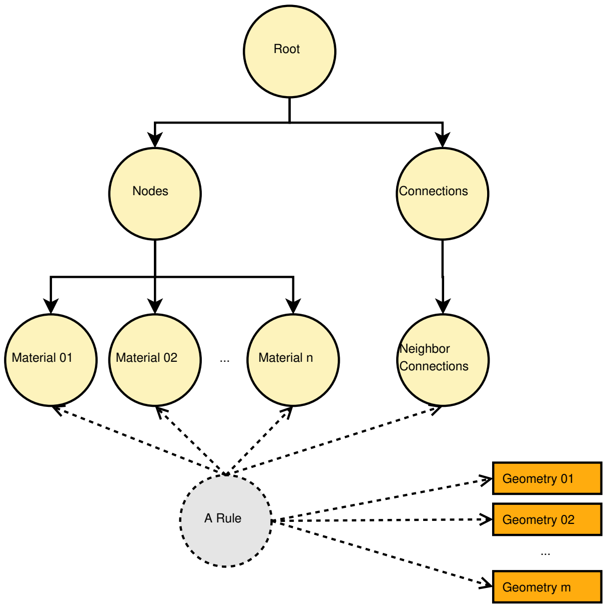


Figure 5.6: Graph Structure and Rule Concept. *The figure shows an overview of using the visualization framework in* `Shawn`. *At first, a graph structure must be created, and added to the already existing root node. For example, such a node may be a Material that describes the color and kind of drawing of a potential object. In addition, there can be so called Geometries created that define the structure of an object, e.g. a sphere or a box. At last, the Rules are used to combine this approach to the visualization process. If a Rule can be applied, the connected Geometry is drawn by using the appropriate Material.*

However, the main design goal was a maximal amount of flexibility. Thus, all of the mentioned tasks are done by the user. For this purpose, the simulation task `TaskAlterOpenSGRuleset` has been implemented that allows the creation of a graph, defining geometries, and the set up of different rules. The latter is described in detail below.

Rule Description

The general concept has already been indicated above. In general, a *Rule* defines whether an object is drawn or not. An object is thereby an abstract description and can be, for example, a geometrical figure like a sphere or a box, a simple line, or, in later implementations, even a complete model. There are three different kinds of rules that can be used: *Node Rules*, *Connection Rules*, and *Topology Rules*.

Node Rules These *Rules* are applied on each `shawn::Node` in the network, and allow the display of a geometrical figure at the appropriate position. The available conditions of the *Rule* are shown in Table 5.1.

NODE RULE	
Rule Name	Unique name of a Node Rule
Parent Name	Name of the parent in the graph
Geometry Name	Name of the used geometry
Tag Comparison	Provide the use of tags to decide whether an object is drawn or not
State	Access to the internal state of a `shawn::Node`, e.g. the active ones
Position	Either the real position or the estimated one
Offset	Allows the movement of objects by the given offset
Priority	The Rules are ordered due to their priority
Close	If `true`, no further Rules are applied on success

Table 5.1: Basic Contents of a NODE RULE.

At first, the unique name of the *Rule* must be set, and can also be seen as an identifier. Then, a parent in the graph structure is needed. Mostly this should be the potential material of the drawn object that in turn is given by the name of the geometry.

In addition, it is possible to define conditions that specify whether a *Rule* is applied successful or not. On the one hand, there is the *Tag Comparison* provided. As already described in the introduction of `Shawn`, the so called *tags* can be added to a `shawn::Node` and may contain a string, a bool value, an integer, or a double value, and can be read by each other class that has access to the appropriate node. The implementation of the *Node Rule* allows the comparison of a tag with a given value, e.g. if a bool content is true, or an integer one is greater than the given one. On the other hand, the state of the node can be checked. That is, provided by the simulator, an active state, an inactive one, or sleeping.

Moreover, also the position of the drawn object is customizable. As the variable one, the position is either the real or the estimated one of each node. Furthermore, it is possible to define a fixed offset by that each object is moved if the *Rule* is applied.

At last, there is the chance to assign priorities to the *Rules* to define the order they are applied. The standard value is 0, and a smaller one represents a higher priority, whereas a greater one let the *Rule* be applied later. In addition, on processing multiple *Rules* in the given order, a successfully applied one is checked for being a final one. That means, no further *Rules* are applied on the appropriate node.

Connection Rules Again, these *Rules* are applied on each `shawn::Node` in the network, but in contrast to the former introduced *Node Rules* there are only relationships shown. That means, a successfully applied *Rule* draws a line between two nodes. Table 5.2 shows the customizable data.

At first, a unique name by that the *Rule* can be identified, and the name of a geometry object to that the drawn lines are added must be given.

CONNECTION RULE	
Rule Name	Unique name of a Connection Rule
Line Name	Name of the used geometry object where the line is added
To	Either `src` or `neighbors`
Tag Comparison	Allow the comparison of tags
Source Position	Either the real position or the estimated one of the source
Source Offset	Allows the movement of the source by the given offset
Source Rule	Node Rule that is applied on the source
Destination Position	Either the real position or the estimated one of the destination
Destination Offset	Allows the movement of the destination by the given offset
Destination Rule	Node Rule that is applied on the destination
Priority	The Rules are ordered due to their priority
Close	If `true`, no further Rules are applied on success

Table 5.2: Basic Contents of a CONNECTION RULE.

Next, the kind of destination is chosen. The standard case is the neighborhood in which potential connections are drawn between a node and its neighbors, but it is also possible select a destination relative to the handled node. The latter can be, for example, drawing a line from the real source position to the estimated one, or otherwise to a given offset.

By using the relationships to the neighborhood, the tags of the nodes can again be compared. In this case, the only implemented possibility is the comparison of integer lists. A node must therefore write an appropriate value, or several ones separated by commas, in a tag. If one value is part of both lists of the compared nodes, the line is drawn.

In addition, the source and the destination can be specified. On the one hand, both can be drawn on the real or estimated position, moved by a optional offset. Moreover, both ones need an assigned *Node Rule* that is applied on the appropriate node.

At last, the possibility of assigning priorities to the *Rules* as well as the check for being a final one are provided again, as already described above.

Topology Rules The third kind of *Rules* are the *Topology Rules* that are used to display structures, terrains, and so on. In contrast to the former presented ones, they are not applied on nodes. Instead, the `shawn::Reading` concept is used. The contents are shown in Table 5.3

TOPOLOGY RULE	
Rule Name	Unique name of a Node Rule
Topology Name	Name of the used geometry where the drawn object is added
Height Reading	Name of the height reading
Bool Reading	Name of the bool reading
Step	Iterate through the `shawn::World` by the given step

Table 5.3: Basic Contents of a TOPOLOGY RULE.

The first parameters are already known from the *Connection Rules*, and describe the name as an unique identifier and the geometry for adding the drawable objects. For now, it is only possible to add lines to the geometry, and thus create a wire frame of the topology.

The procedure of drawing is very simple and operates as follows. It is iterated through the x and y coordinates of the world by using the defined step counter. On each point the given height reading that is identified by its name returns the appropriate z value to enable a three-dimensional view on the topology. Additionally, the bool reading is used to decide whether the point should be drawn or not.

5.3 Algorithms and Dike Representation

The simulation of the designed algorithms in a sandbag dike scenario has been done by using several options of `Shawn`. Figure 5.7 shows a rough overview of the most important classes.

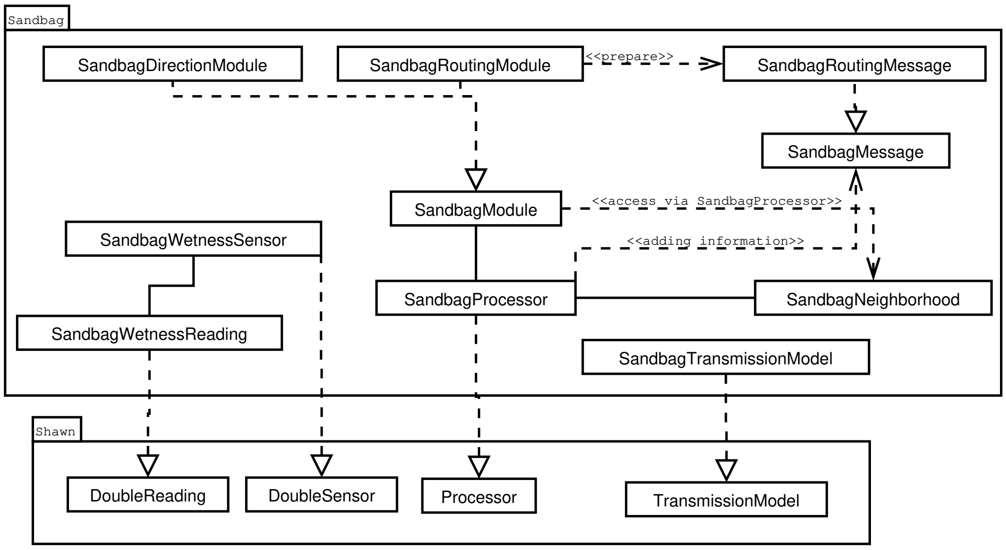


Figure 5.7: Algorithms and Dike Representation Overview. *The class diagram shows the most important classes for the simulation of the designed algorithms in a sandbag dike scenario. The central unit is the* `SandbagProcessor` *that provides the possibility of implementing the algorithms in so called Modules that are divided in arbitrary ones, direction ones, and routing modules. The latter are used for allowing the routing of appropriate messages from a source to a given sink. In addition, it is possible to simulate the behavior of soaked sandbags. On the one hand, the* `SandbagTransmissionModel` *can be enabled to drop messages sent through wet sand. On the other hand, the* `SandbagWetnessReading` *simulates the moisture penetration of the sandbags, and can be used by an implementation over the* `SandbagWetnessSensor` *that allows the simulation of errors in the measurement process.*

At first, the implemented processor is presented by describing the principal tasks to give a general design overview. Then, the used module concept is shown in detail. At last, the simulation of an increasing waterline and the consequently moisture penetration of the sandbags is illustrated.

5.3.1 Sandbag Processor

The `SandbagProcessor` is the central unit for a simple and flexible implementation of the designed algorithms. To allow exchangeability of different approaches as well as a potential extensibility, a module concept has been established. That is, the processor can contain multiple modules that are defined at run-time per configuration file, and provides a basic functionality like a working phase in each iteration, sending and receiving of messages, and an one-time boot phase. More precisely, the relationship between the processor and the modules is like the relationship between a `shawn::Node` and a `shawn::Processor`, but allow a more intensive connection. For example, all modules have access to the neighborhood of a node provided by the `SandbagProcessor`. In addition, a very simple energy model is realized on the processor. At last, each sent and received message is processed, which in turn is used to manage the routing of messages. Figure 5.8 shows a class diagram of the implementation of the processor.

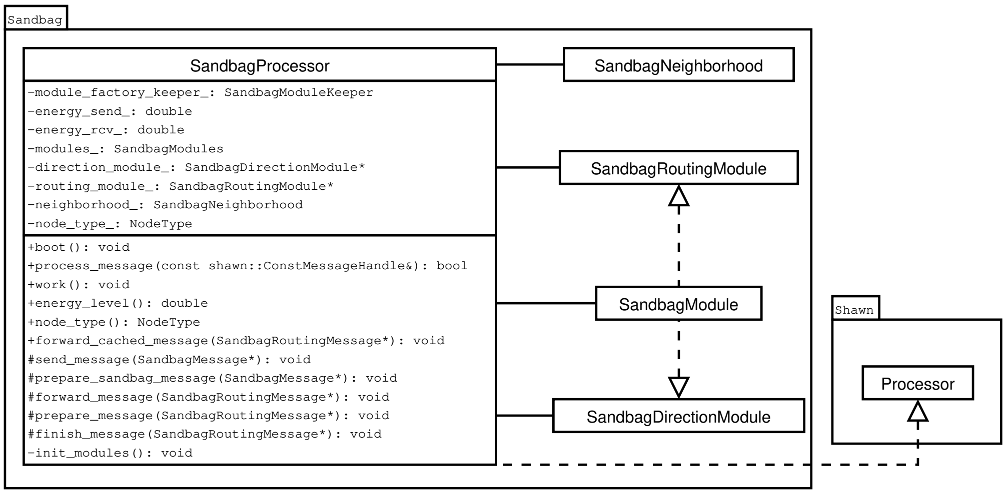


Figure 5.8: Class `SandbagProcessor`. *The class diagram shows a basic overview of the implementation of the* `SandbagProcessor`*. The main purpose is the management of* `SandbagModules` *that can fulfill different tasks. In addition, there is exactly one direction as well as one routing module associated. The latter is used for the routing of messages that can be prepared, forwarded, and received. Moreover, there is access to the* `SandbagNeighborhood` *and the type of the node provided. The latter can be either a normal one or a base station. At last, the remaining of energy is hold.*

The several aspects of the `SandbagProcessor` are presented in detail below.

Modules

The main design goal for the module concept was exchangeability as well as extensibility. For this purpose the factory pattern also used in `Shawn` has been taken. The processor got therefore a module keeper object that contains factories which in turn are able to create instances of appropriate modules. The advantage of this procedure is that an implemented module can be load at run-time by adding the name to an appropriate parameter. On booting, the processor reads the content of the parameter and loads all wanted modules. Figure 5.9 gives an overview of the basic `SandbagModule` and the appropriate factory.

In general, the module provides access to the most important contents of the processor such as the neighborhood, the associated `shawn::Node`, access to the direction and routing module, and so on. Moreover, sent messages are passed to the processor. In addition, the module is derived from `shawn::Logger`, and thus is in possession

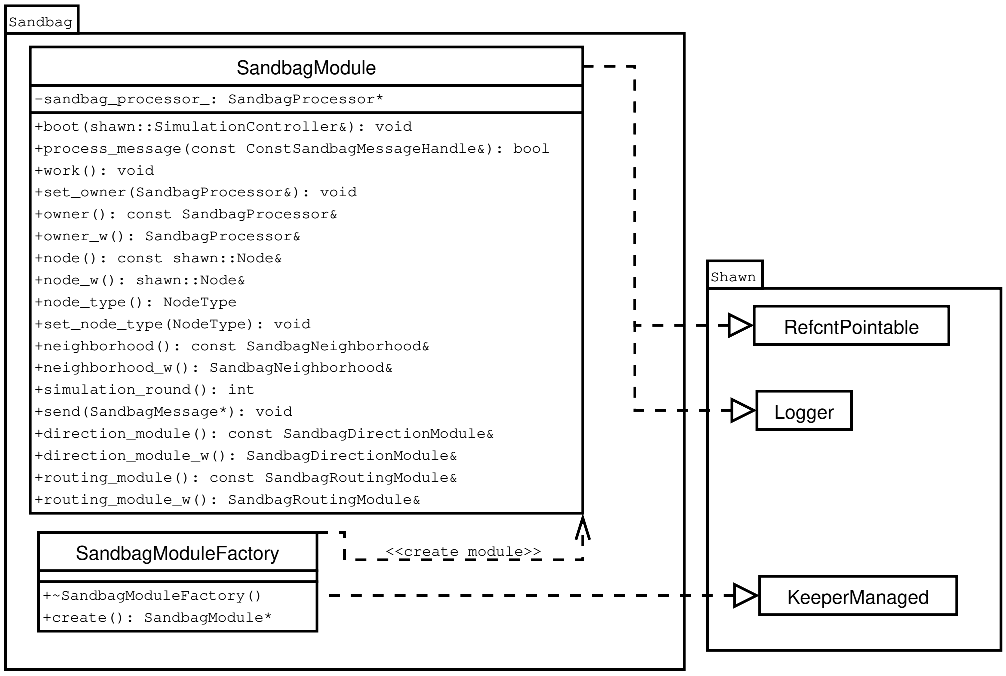


Figure 5.9: Class `SandbagModule`. *The sandbag module concept has been designed by keeping the factory pattern in mind. Each module has an associated factory that is able to create an instance of the module. The module in turn provides access to the basic functionality of the processor.*

of an own logger object to customize the logging output of different implementations of modules.

A more detailed description as well as the special purposes of the derived modules is done in Section 5.3.3.

Neighborhood

In general, the implemented algorithms need an awareness of the local neighborhood. That can be, for example, the point in time of the last activity to recognize potential dead neighbors, the remaining energy level, or in case that it is needed, the 2-hop neighborhood. The latter means that a node is also informed of the neighborhood of its neighbors.

For this purpose, the processor provides information of the local neighborhood that can be accessed by each module. In the consequence for the need of different information in the algorithms, the processor do not update the contents of the neighborhood. Hence, an appropriate module has been implemented, and is described in Section 5.3.3.

However, the implementation allows the building of the n-hop neighborhood in theory, because it contains information about the neighbors that in turn use the neighborhood class. An overview is given in Figure 5.10.

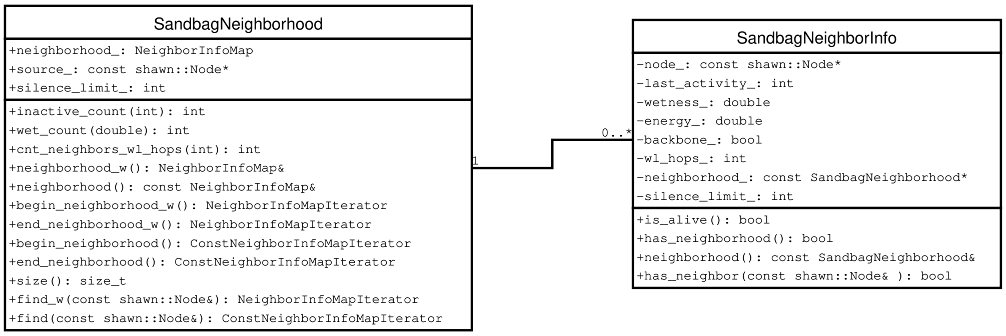


Figure 5.10: Class `SandbagNeighborhood`. *The `SandbagNeighborhood` contains multiple instances of the `SandbagNeighborInfo` which in turn describes the available information of an appropriate neighbor. The neighbor info is again associated with a neighborhood to allow the building of a n-hop neighborhood. The class diagram shows the basic information that is provided by the classes, but leaves all the get and set methods out.*

Energy Model

The main focus of this thesis is the maintenance of a communication backbone near to the waterline, which is responsible for forwarding and routing messages. Although this is done primarily for energy efficiency reasons, the backbone can also be rated by the number of messages routed by nodes belonging to the backbone in contrast to nodes which do not. Thus, there is only a very simple energy model which only considers sent and received messages. Each node got a standard energy level of 100% whereas sending and receiving decrease this value by 0.05 and 0.01, respectively. A node is dead, if its energy level is ≤ 0.

Message Sending and Routing

The handling of messages is divided in two parts. On the one hand, there are the `SandbagMessages` that can be used for the standard case of broadcasting. On the other hand, there are `SandbagRoutingMessages` that are used for the routing process to a given sink in the network. Figure 5.11 shows a class diagram of both classes.

The basis for the message sending process in the implementation is the `SandbagMessage` which is used for each local broadcasted message. The appropriate method in the module passes a sent message to the processor which in turn forwards it. In addition, the processor adds basic information about the node to the message, such as the actual energy level and the hop distance to the waterline. Consequently, the neighbors can update their knowledge of the sender by each received message.

For enabling the routing of messages from an arbitrary source to a given sink, there is the `SandbagRoutingMessage` that is derived from the `SandbagMessage`. The general routing process as the sending and receiving of such messages is done in the processor, whereas the preparation is passed to a routing module, and is described later in the *Module Concept*. The routing message provides suitable information for the appropriate module. Hence, there is a list of wanted receivers on the one hand, and

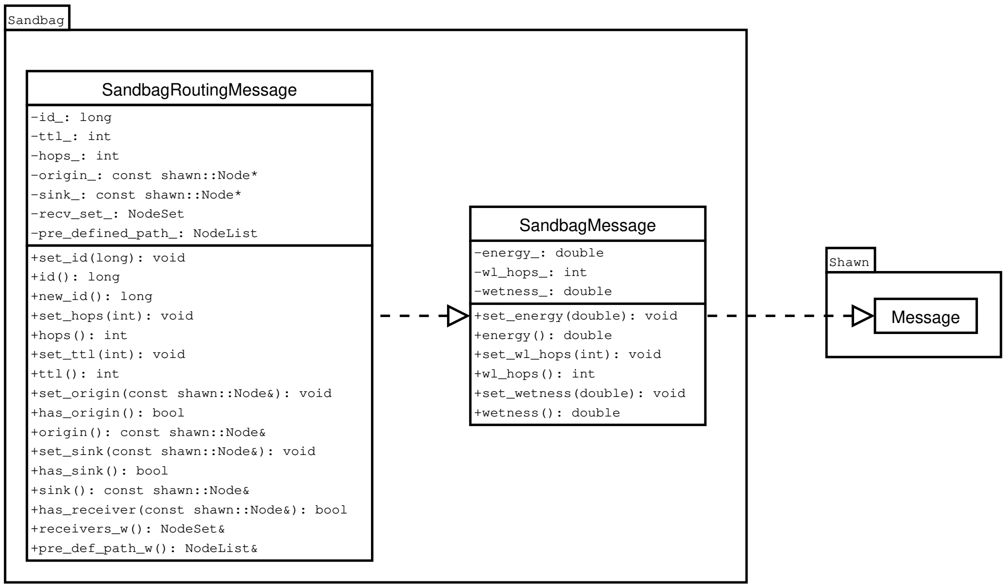


Figure 5.11: Class `SandbagMessage`. *The `SandbagMessage` is the basis for message sending in the implementation of the dike scenario of this thesis. Each sent message is derived from this one, and thus general information of the node such as the energy level and the current hop distance to the waterline are stored. The data is set by the processor, so that a potential module does not need to do this on each sent message. In addition, there is also a special message that is used for the routing process. It contains the source and the destination of the message, as well as the time to live. Moreover, a set of receivers in the local neighborhood can be set just as well a predefined routing path.*

the possibility to add a predefined routing path on the other. Moreover, each created routing message get a unique ID by which it can be identified. In addition, the origin and the sink are stored. At last, the message contains the time to live, and the actual hop count.

5.3.2 Module Concept

The module concept has already been indicated in Section 5.3.1, and shown in Figure 5.9. There are three different kinds of modules: First, the standard ones that have only access to the data provided by the `SandbagModule`. Second, a direction module. A potential implementation must provide a rough location awareness that enables at least a *left-right* feeling. The third one is the routing module. If a message is routed, the processor passes it to the routing module for preparation. The objective is either adding nodes to the potential receivers, or build a predefined routing path.

Miscellaneous Modules

The *Miscellaneous Modules* have been implemented to fulfill basic tasks, and can be auxiliary for the implementation of backbone or direction algorithms. Figure 5.12 gives an overview of the available modules.

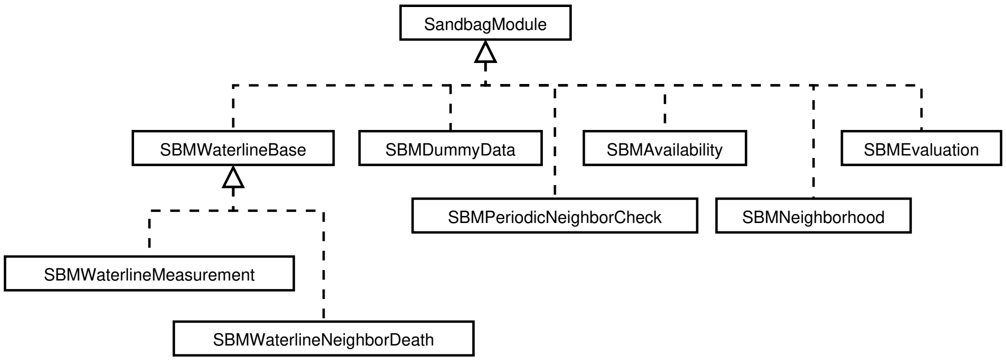


Figure 5.12: Miscellaneous Modules. *The figure shows the implemented additional modules. The waterline identification has been implemented by using a base class that fulfills tasks of both methods. The* `SBMDummyData` *is used for sending a routing message to a particular node. The availability one checks whether a node malfunctions, and the evaluation one collects data in each iteration. Moreover, the neighborhood module updates the local neighborhood, and the last one sends messages in periodic intervals to inform the neighbors of being alive.*

Waterline Identification The waterline identification has been implemented as a standard `SandbagModule`. There is also a base class that fulfills common tasks. For instance, the update of the minimal hop distance to the waterline is done here.

Sending Routing Messages The `SBMDummyData` sends routing messages from a given node to a particular sink. The destination is chosen by using the *Direction Module*, and selecting the base station with the greatest hop distance.

The number of sent messages is stored in a tag, and thus can be read for evaluation issues. The other way around, the receiving node also stores the number of received routing messages in a tag.

Availability The availability module checks in each iteration whether a node is still alive or not. It reads the current wetness of the surrounding sand, and deactivates the node on demand.

Evaluation The evaluation module has been implemented for evaluation reasons. On the one hand, the important data of the current simulation is written to the associated logger object. On the other, the result can be written to a file that can be analyzed.

Periodic Neighbor Check This module sends messages in periodic intervals to inform the neighbors of being still alive. If there has no message been sent in a given interval, a small dummy message is broadcasted. Thus, if other modules are sending messages anyway, this one recognizes the sending, and does not send a message itself.

Neighborhood The neighborhood module is responsible for the update of the local neighborhood. It receives each message that is sent to the associated processor, and reads the contained data if it is a `SandbagMessage`. In addition, it sends periodic messages to enable an ongoing 2-hop neighborhood.

Direction Module

The direction module is responsible for a rough location awareness, and provides abstract methods that must be implemented by an derived class. Figure 5.13 shows the class diagram of the module.

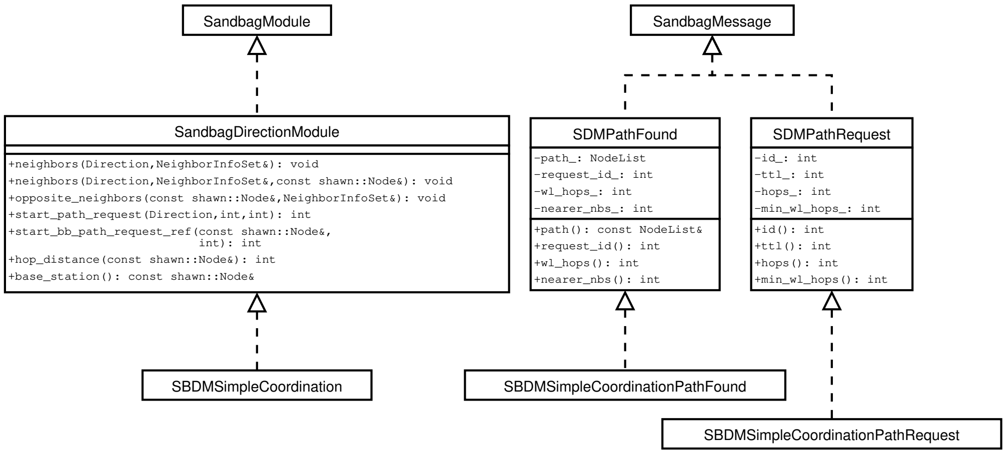


Figure 5.13: Direction Module. *The class diagram shows the responsibility of the direction module. At first, a potential implementation provides a direction decision for selected neighbors. Second, it is possible to start path requests that are either sent over all neighbors or only to backbone ones. By starting the path request, an unique ID is returned. This ID can be used by processing `SDMPathFound` messages. If such one is received, the ID can be compared to the requested one. At last, a direction module must be aware of base stations in the near, and store the minimal hop distance. There has also an implementation of the direction module been done, namely `SDMSimpleCoordination`.*

The first responsibility for a direction module is the classification of the neighborhood in directions. Thus, it must be possible to select nodes from a given set that are located in the given direction. Moreover, for a given node such as a base station the ones which are located towards or contrary can be elected.

Next, an implementation of the direction module must provide the processing of path requests. That is, each module can start the request in the given direction by calling the appropriate method. In addition, the time to live and the minimal hop distance to the waterline can be given. The latter ensures that the request is sent only to nodes with a greater value. However, the request returns a unique ID that must be stored by the initiator. If the path has been found, the message is routed backwards, and contains the requested path. Then, the initiator processes `SDMPathFound` messages, and compares the stored request ID to the one in the message. Thus, it is assured that the correct path is received. Moreover, it must also be possible to send requests along the backbone, and only the backbone. For this purpose, the second shown request method is responsible.

At last, the direction module provides information about base stations. On the one hand, such a node can be requested. On the other hand, the hop distance to a given one can be asked for.

The idea of the *Relative Reference Points* has been implemented as a direction module, namely `SDMSimpleCoordination`, and provides all required possibilities.

Routing Module

The routing module is responsible for the routing of messages. If a `SandbagRoutingMessage` is going to be routed, the processor calls the `prepare_routing` method to add wanted receivers to the message. The module provides more helpful methods for potential implementations as shown in Figure 5.14.

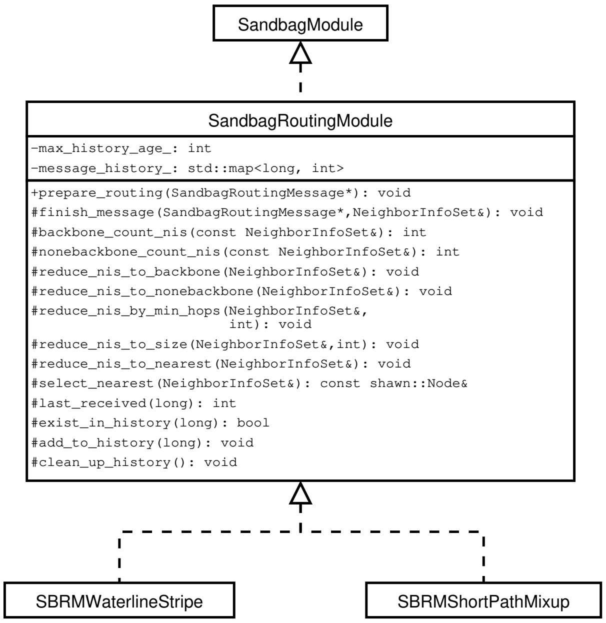


Figure 5.14: Routing Module. *The figure shows a class diagram of the* `SandbagRoutingModule`*. The most important method is* `prepare_routing` *that is called by the processor to prepare the routing message and add the wanted receivers. In addition, there is some helpful functionality for potential implementations provided. On the one hand, there are methods to work on a given set of neighbors. On the other hand, a message history is handled, and contains the last received messages.*

The processor owns exactly one instance of a routing module. If a routing message is sent, it calls the module to prepare the message. That is, either adding nodes to the receiver set, or defining a predefined routing path. The latter can be used to cover short distances, for example, by adding the nodes of a path request. After the preparation of the message, it is finished by the processor and then broadcasted. Again, the receiving processor decides whether the message should be dropped, or forwarded and thus prepared by the routing module.

The routing module also provides auxiliary methods which can be used by potential implementations of the module. For instance, a set of neighbors can be reduced only to backbone ones, or the neighbor that is nearest to the waterline can be selected.

Moreover, a message history is provided. The ID of each received routing message is put into the history by additionally storing the simulation round of the reception. In each iteration, message IDs that are older than a given value are cleared.

In the consequence of the close relationship between the routing process and the backbone algorithms, the appropriate implementation has completely been done in a routing module. Hence, there are the classes `SBRMWaterlineStripe` and `SBRM ShortPathMixUp`.

5.3.3 Dike Representation

In addition to the implementation of the designed algorithms, the behavior of a sensor node in a sandbag must be simulated. For this purpose a waterline is needed to let the nodes decide whether they are below the actual level. Furthermore, the moisture penetration of the dike must be simulated, and a sensor should be able to measure the current wetness of its environment. At last, if a sandbag is getting soaked, the transmission of messages is possibly reduced by a given factor. The implemented approaches are described in detail below.

Readings and Sensors

As already mentioned in Section 5.1, a `shawn::Reading` returns a computed value for a requested position. This is used for the representation of the waterline and the moisture penetration of the dike as shown in Figure 5.15.

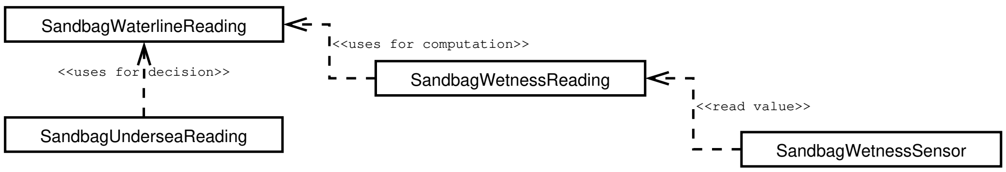


Figure 5.15: Sandbag Reading Overview. *The figure shows the realization of the simulation of the dike representation. The waterline reading represents the actual height of the waterline, and is used by the undersea reading as well as the wetness one. The former decides for a given position whether it is located below the waterline. The latter takes the information into account for a computation of the moisture penetration. At last, the sensor is used to read the actual amount of wetness by allowing the integration of measurement errors.*

At first, a waterline reading has been implemented. The only purpose is to manage the actual height of the waterline by returning the same value for each position in the topology. That means, the reading is not aware of a potential dike in the topology which would block the water, and represents a completely flooded scenario. It possible to customize the waterline, and define increasing, decreasing, and stalled periods.

For the purpose of a consideration of the dike, the undersea reading can be used. It returns, whether a given position in the topology is located below the waterline. For example, a TOPOLOGY RULE as described in Section 5.2.3 can be created by assigning the waterline reading as the height one, and the undersea reading to decide whether a point is used.

Moreover, also the wetness reading uses the waterline one. It represents the moisture penetration of the dike. That is, for a given position the current amount of wetness is computed. The implementation considers also an approximation of the seepage of the dike as described in Chapter 3 by customizing the amount of leakage, the draft, and so on.

At last, there is a wetness sensor that can be used by a module to simulate the measurement of the surrounding moisture. In addition, the sensor supports also errors in the measurement process. For example, a sensor can be completely malfunctioned, or returns diverged values.

Message Transmission

For the simulation of message transmission through wetted sand, a new `shawn::TransmissionModel` has been implemented. The `SandbagTransmissionModel` drops messages that are either sent or received by nodes in a wetted region. A message is dropped with a given probability that depends on the actual amount of wetness and a given threshold, and works as follows: $ActualWetnessLevel \cdot Threshold \leq Random[0, 1]$.

5.3.4 Additional Tasks

By implementing the above described classes there have also two additional tasks been created.

The `SimulationTaskSandbagEvaluation` collects important information of the current simulation. On the hand, it shows the results of the routing process, the remaining energy of backbone and none-backbone nodes, the number of sent messages, and so on. On the other hand, it uses the data of the evaluation module and collects average, minimal, and maximal values of the simulation process. The information is written to console as well as logged in a given file.

The other implemented task is the `SimulationTaskSandbagSelectNode` is able to select a node and write wanted content in a given tag. The selection process is done by defining a position, and electing the nearest node. The task is used, for example, to elect base stations or the node that starts the routing process.

6 Evaluation

This chapter presents the results of the simulation of the designed algorithms. They are tested in different scenarios, under several conditions like the variation of the communication range, and by modifying essential parameters. At first, the waterline identification methods are evaluated by taking their individual assumptions of the environment into account. Then, by using the waterline identification methods and the approach for a rough location awareness, the results of the backbone algorithms are shown.

6.1 Waterline Identification

At first, the waterline identification is evaluated. This is done by varying the communication range, allowing the detection of false positives, and analyzing the parameters of the algorithms.

The most important rating factor is the amount of false positives and false negatives, respectively. The former are the nodes which have identified the waterline albeit they should not do so, whereas the latter are the ones which do not identify it albeit they are located in the near. The results are given in absolute number of nodes as well as the relative fraction. In addition, the average real distance of false positives is presented.

Both waterline identification methods, *Neighbor Loss* and *Local Measurement*, are rated separately. This is done in the consequence of the different assumptions on which they are based. The *Neighbor Loss* assumes that nodes malfunction on contact with wetness, whereas *Local Measurement* requires on appropriate event to identify the waterline.

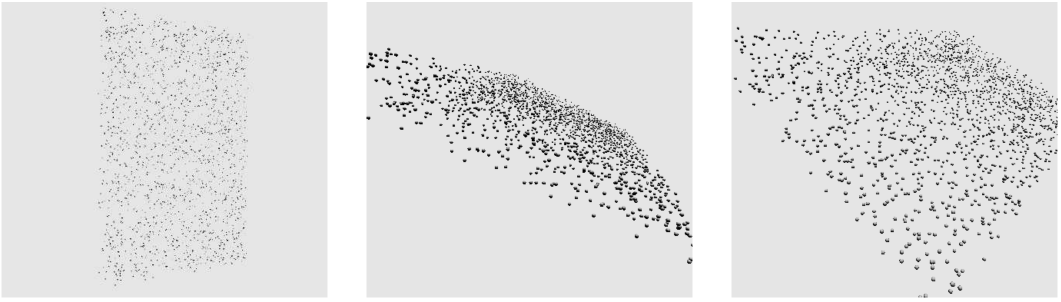
Nevertheless, for the evaluation process both methods are tested in the same scenario which is described in Section 6.1.1. After that, in Section 6.1.2 and 6.1.3 the results of the simulations of the different methods are presented.

6.1.1 Scenario Description

Both waterline identification methods have been simulated in a simple standard scenario as shown in Figure 6.1. The network consists of 2000 nodes that are placed randomly in the given region. The communication range is set to 18 units which in turn results in a connectivity of 15 which defines the average number of neighbors of the nodes.

The message transmission is done in a reliable manner. That means, each sent message is delivered to the receiver without loss or delay. In addition, a diskgraph communication model is used to assure that if a node u can receive messages from node v, node v in turn always receives the broadcasted messages of node u.

The processor provides several implemented modules. The *Availability* one simulates an arbitrary node loss on the one hand, and a wetness based on the other hand. The *Neighborhood* module allows the use of the information of the 2-hop neighborhood,



(a) Top View (b) Side View (c) Diagonal View

Figure 6.1: Standard Scenario for Waterline Identification. *The figure shows the used standard scenario for the waterline identification. It consists of 2000 nodes that are represented by the grey spheres. It is assumed that the shown nodes were inserted into sandbags which in turn are placed on the riverside of a dike.*

and is updated every 5 rounds. In addition, each sent message contains the energy level, the hop distance to the waterline if known, and the potential measured amount of wetness. Such information is updated on each reception. Messages in turn are sent at least every 3 rounds which is assured by the *Periodic Neighbor Check* module that sends dummy messages in case of need.

The simulation runs for 100 iterations after that the results are evaluated. In addition, the waterline is increased in the first 30 iterations by 15 units. After that, it holds it current height up to the end of the simulation. The height is chosen to put about $\frac{1}{3}$ of the network below the waterline.

To allow accurate results, each simulation has been done 100 times, and the below presented results represent the appropriate average values.

In addition, Figure 6.2 shows a situation in the scenario of an increasing waterline and identifying nodes.

6.1.2 Neighbor Loss

The neighbor loss method uses the standard parameters as already described above. In addition, algorithm specific parameters must be set.

To increase the possibility of false detections, an arbitrary node loss has been enabled. From iteration 25 to 35 each node malfunction with a probability of $p = 0.01$. This configuration has been chosen with respect to the increasing waterline that stops at iteration 30.

Nodes are supposed to be dead if there has no message been sent for 5 rounds. In addition, nodes with a last activity of more than 10 rounds ago are ignored, as well as nodes with a lower energy level than 10%.

At last, the relative loss bound is set to 30%.

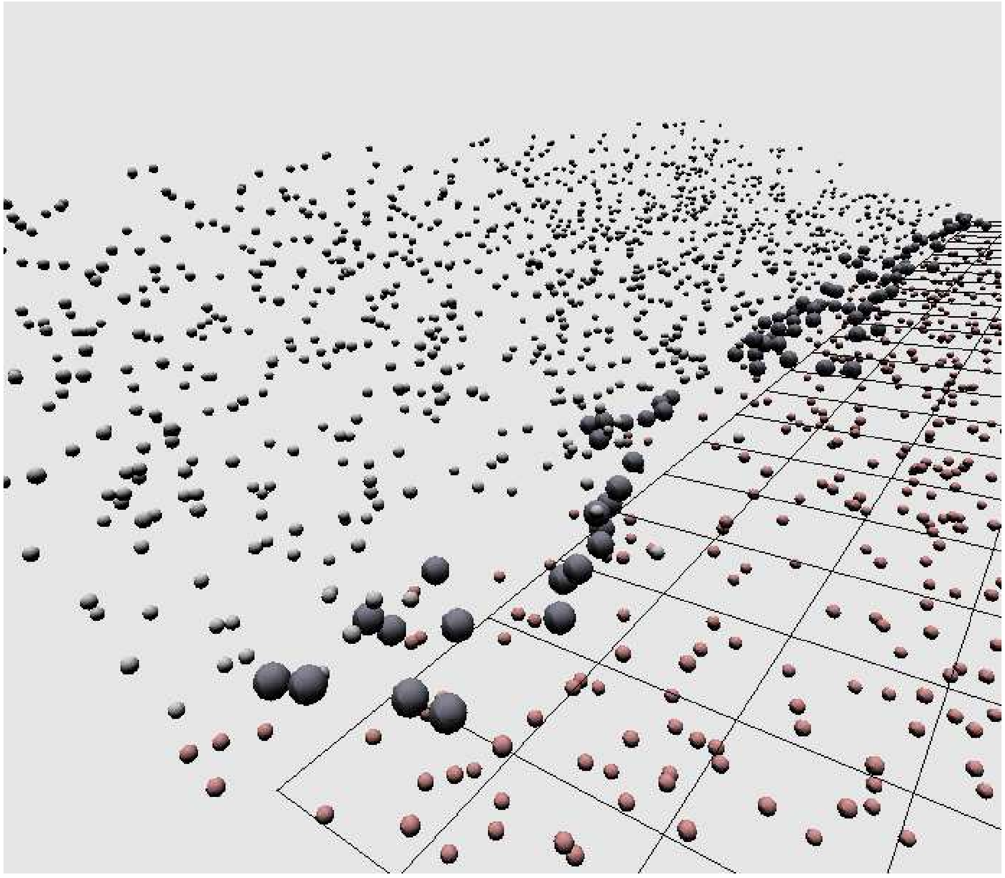


Figure 6.2: Example for Waterline Identification. *The figure shows a situation of nodes identifying the waterline which is represented by the wired grid. The red nodes below the grid are soaked, and thus already dead. The small grey ones at the upper side are still alive and have not identified anything. The bigger dark grey ones have identified the waterline, and enable the other ones to determine their minimal hop distance to the waterline.*

Variation of Communication Range At first, the variation of the communication range is evaluated. The parameter varies from 14 to 24, which leads to connectivities from 8 to 30. Figure 6.3 shows a more detailed overview of the used ranges.

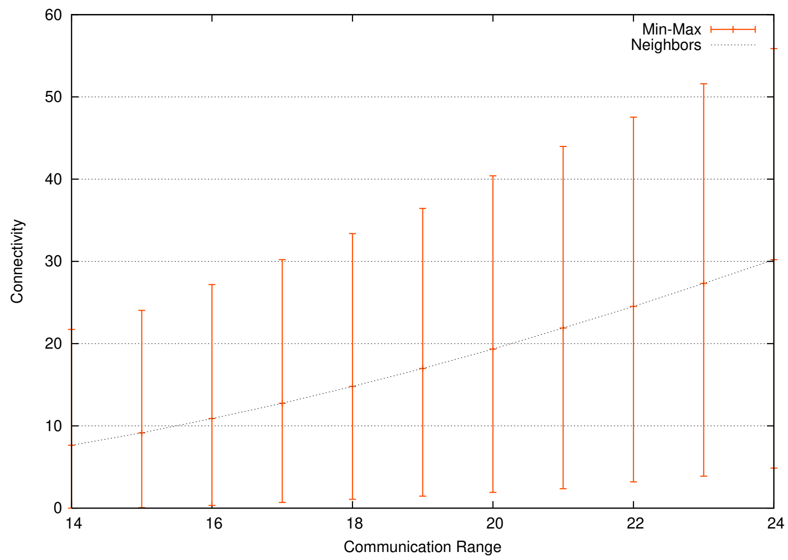


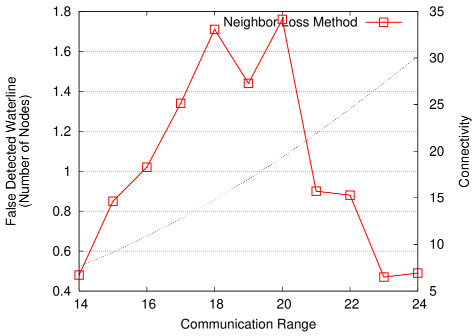
Figure 6.3: Connectivity by the Variation of Communication Range. *The figure shows the resulting connectivity by the variation of the communication range in the standard scenario. In addition, the minimal and maximal number of neighbors of a node are shown. For example, a communication range of 18 leads to a connectivity of 15. In addition, there is at least one node that has only got one neighbor, as well as at least one node with 33 neighbors.*

However, Figure 6.4 shows the results of the simulations. There are the false positives, the false negatives, and the distances of the false ones to the waterline evaluated.

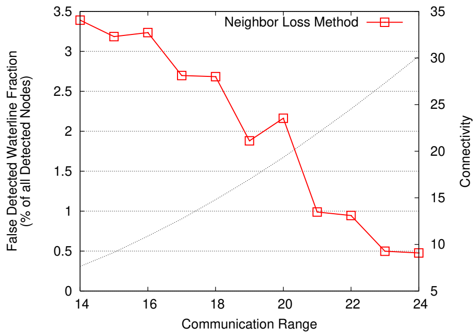
In (a), the absolute number of false positives is shown. The value varies between 0.5 and 1.8 nodes that have detected the waterline albeit they are not in the immediate near. A small communication range as well as a bigger one leads to better results than a middle range one between 17 and 20. Nevertheless, the absolute number is quite small with about one false positive per simulation. In almost the same manner the relative fraction of false positives is proportionally small as shown in (b). It varies from 3.5% to 0.5%, and is constantly getting smaller with an increasing communication range.

The number of false negatives in (c) varies from 98 to 135, and is much greater than the absolute values of (a). With an increasing communication range, the value is continuous rising. Considering the relative fraction in (d), it is continuous increasing, too, but still relative small with a span from 9.8% to 14.7%.

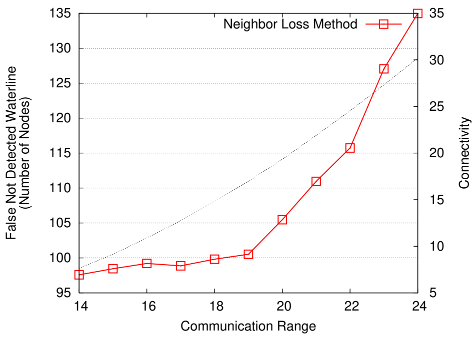
In (e), the average real distance to the waterline is shown. By varying from 38 to 70, there are nodes that have detected the waterline but are multiple times the communication range away. The other way around, the false negatives in (f) are mostly more than the half of the communication range away.



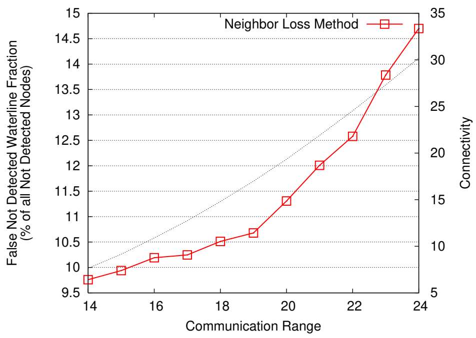
(a) Absolute Node Count of False Positives



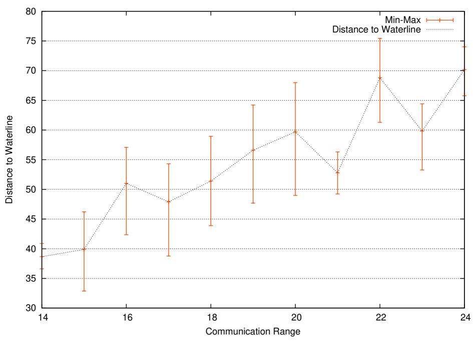
(b) Relative Fraction of False Positives



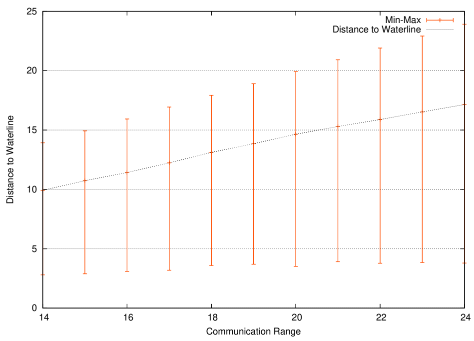
(c) Absolute Node Count of False Negatives



(d) Relative Fraction of False Negatives



(e) Average Distance to Waterline of False Positives



(f) Average Distance to Waterline of False Negatives

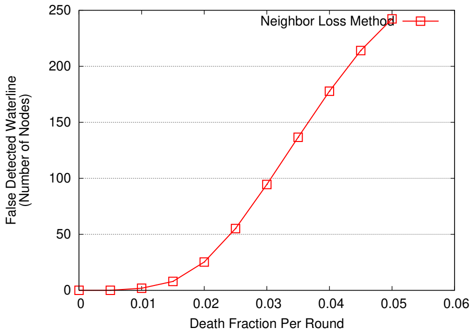
Figure 6.4: Neighbor Loss: Variation of Communication Range.

Variation of Random Node Malfunction The variation of node malfunctions is done in multiple iterations as described in the *Standard Scenario*. From round 25 to 35 a node dies with the given probability. The evaluated values vary from 0 to 0.05. The results are shown in Figure 6.5.

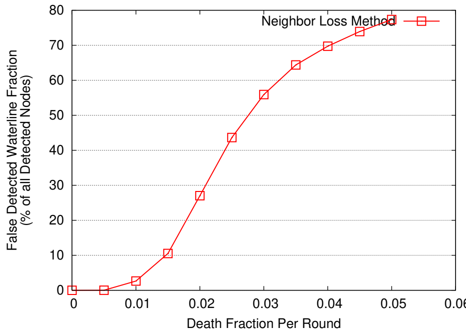
Up to a node loss of 0.015 the absolute number false positives is acceptable small as shown in (a). After that, the false detections increase precipitously to number of nearly 250 nodes by a node loss of 0.05 per round. A similar observation can be done in (b) that shows the relative fraction of false positives. In the beginning, only a small fraction identifies the waterline albeit it is incorrect. Later, by a probable loss of 0.05, the fraction reaches 80%.

On the contrary, the number of false negatives decreases continuously with a greater probability of node loss. If there is no node loss, an average number of 125 nodes do not identify the waterline albeit they are in the immediate near. If the probability is set to 0.05 per round, only 20 nodes do so, too. The relative fraction falls from about 12% to 6%, and is shown in (d).

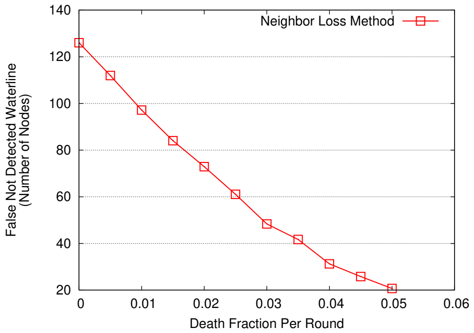
The average real distance to the waterline is multiple times the communication range, if the probability of a node loss is greater or equal than 0.01, and is averaged to 60 units. Otherwise, if the probability is smaller than 0.01, the mean distance is acceptable inside the communication range, or even 0 if there is no node loss. The distance of the false negatives is constantly 13 units by a communication range of 18.



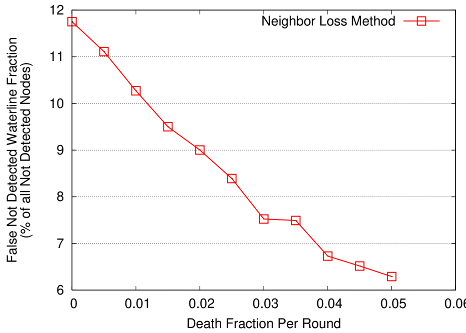
(a) Absolute Node Count of False Positives



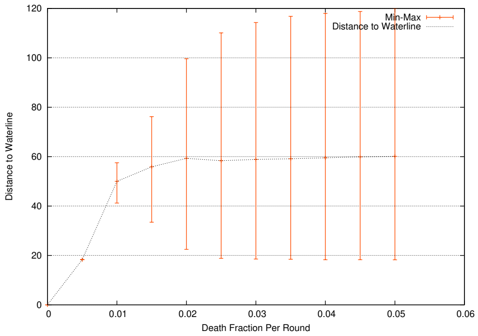
(b) Relative Fraction of False Positives



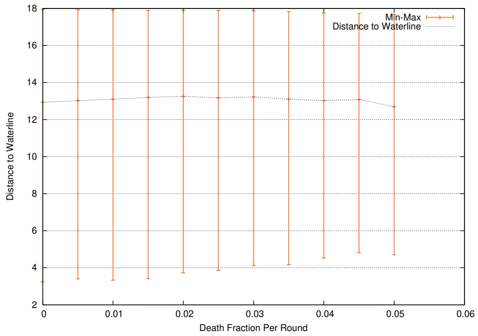
(c) Relative Absolute Node Count of False Negatives



(d) Fraction of False Negatives



(e) Average Distance to Waterline of False Positives



(f) Average Distance to Waterline of False Negatives

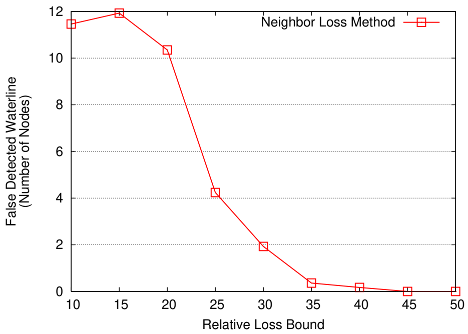
Figure 6.5: Neighbor Loss: Variation of Random Neighbor Malfunction.

Variation of Parameter *Relative Loss Bound* The variation of the *Relative Loss Bound* parameter defines the relative fraction of nodes that are assumed to be dead to identify the waterline. In addition, the additional methods given in 4.2.1 to allow for a more reliable decision are also applied, but not varied. However, the *Relative Loss Bound* has been tested from 10% to 50%, and the results are shown in Figure 6.6.

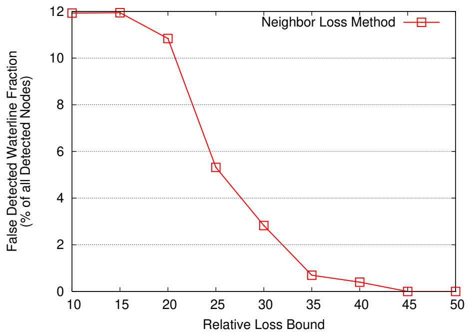
If the bound is set to at most 20%, there are between 11 and 12 nodes that identify the waterline albeit they should not. By a minimal loss of 25% or 30%, the number drops to only 4 and 2 nodes, respectively. After that, on the setting the parameter to 35% or greater, there are nearly no false positives. Proportional to the absolute number, the relative fraction is decreased from 12% to 0% by having the same mentionable points.

In contrast to the false positives, the number of false negatives increases with a higher *Relative Loss Bound*. From 10% to 20% the number is constant by about 80 nodes. After that, it rises linear to over 140 nodes. Again, the process of the relative fraction is proportionally, and varies from about 8% to over 14%.

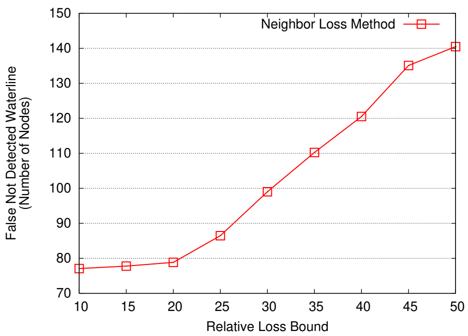
The distance of the false positives is approximately 50 units by a communication range of 18, if the *Relative Loss Bound* is less or equal than 40%. In the consequence of a lack of such ones at 45% or 50%, the distance is 0 at the according positions. False negatives must be again in communication range to the waterline, and are approximated 14 units away. With an increasing parameter, the nodes draw near the waterline to 12 units.



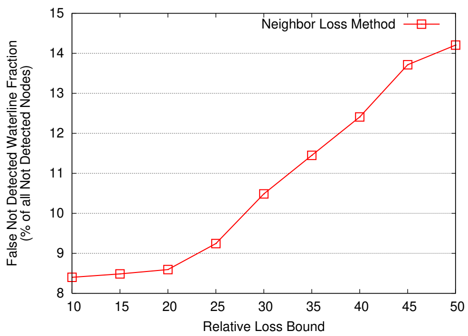
(a) Absolute Node Count of False Positives



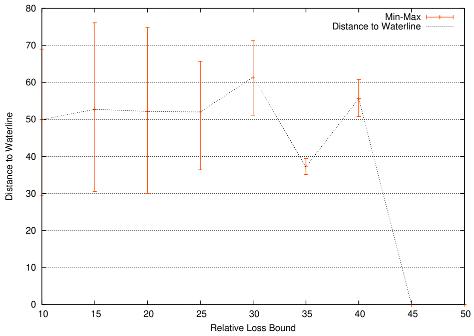
(b) Relative Fraction of False Positives



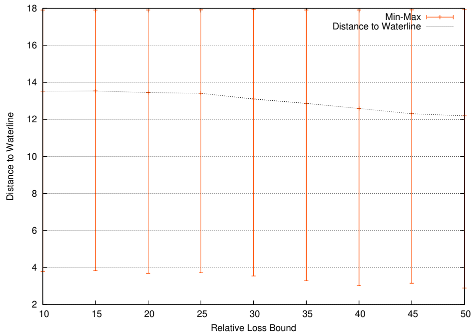
(c) Absolute Node Count of False Negatives



(d) Relative Fraction of False Negatives



(e) Average Distance to Waterline of False Positives



(f) Average Distance to Waterline of False Negatives

Figure 6.6: Neighbor Loss: Variation of Parameter *Relative Loss Bound*.

6.1.3 Local Measurement

The *Local Measurement* has been tested by using the standard parameters in almost the manner as the *Neighbor Loss* method. Again, the communication range, the error-proneness, and the setting of the important parameters are evaluated.

In general, 10% of the sensors are faulty. Consequently, such sensors do not measure wetness albeit they should, or identify wetness in dry sandbags.

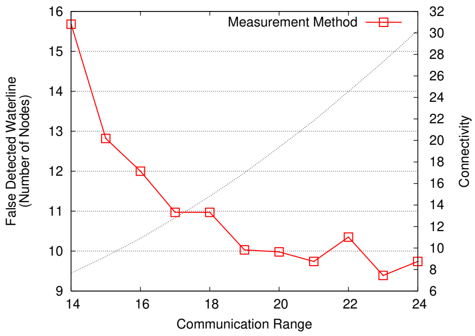
As already described in Section 4.2.2, a positive sensor needs at least 30% of neighbors that have done so, too, whereas a negative one identifies the waterline if 75% of its neighbors have measured wetness.

Variation of Communication Range At first, a variation of the communication range is evaluated. The resulting connectivity is the same as shown in Figure 6.3. The results can be seen in Figure 6.7.

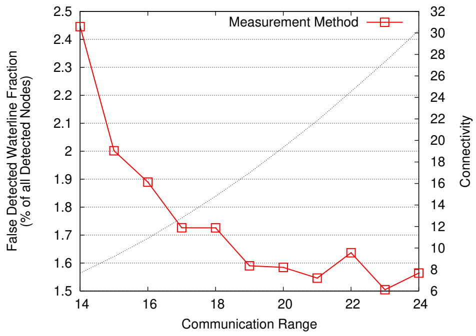
The number of false positives is shown in (a). With an increasing communication range the number of such nodes drops from 16 to 10 whereby the steepest descent occurs from a communication range of 14 to 17. After that, the value is nearly constant. A similar process occurs by considering the relative fraction shown in (b). The fraction varies from 2.5% to 1.5%.

The number of false negatives is relative high in comparison to the *Neighbor Loss* method, but is also hardly affected by the communication range. There is a flat ascent from about 250 nodes to 260. Again, the relative fraction proceeds similar, and varies from 18% to 19%.

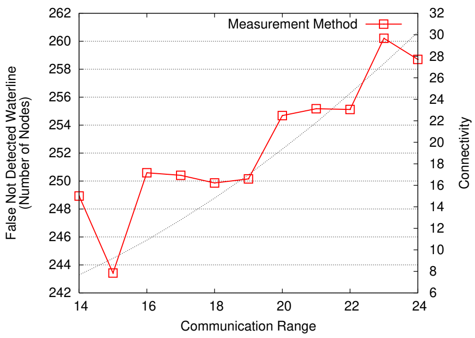
The distance to the waterline of the false positives shows much better results than the ones of the *Neighbor Loss* method. With a communication range of at least 19, the average distance as well as the maximal measured one is not greater than the range. Also the results at 17 and 18 are acceptable with being not greater than two times the communication range. Nevertheless, a smaller range leads to worse results, although the average distance is near to the communication range. The distance of the false negatives is always 0, because such nodes are already below the waterline to be able to measure potential wetness.



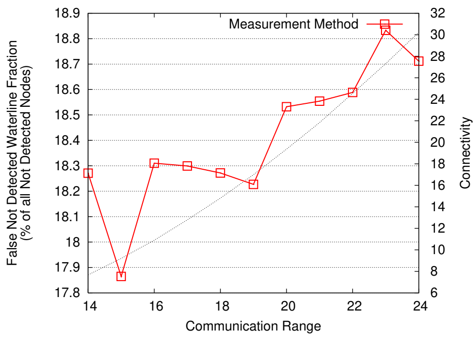
(a) Absolute Node Count of False Positives



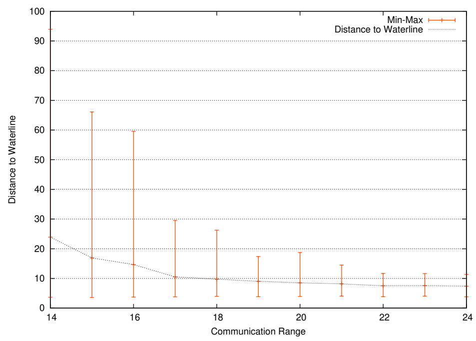
(b) Relative Fraction of False Positives



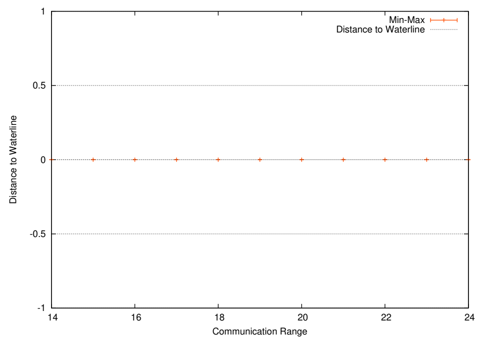
(c) Absolute Node Count of False Negatives



(d) Relative Fraction of False Negatives



(e) Average Distance to Waterline of False Positives



(f) Average Distance to Waterline of False Negatives

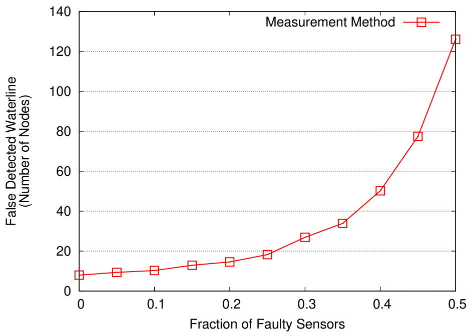
Figure 6.7: Local Measurement: Variation of Communication Range.

Variation of Sensor Malfunction The reliability in the sense of measurement failures is simulated by a variation of the fraction of faulty sensors. As already mentioned above, a faulty sensor measures wetness if there is not any, and detects one if it is not supposed to do so. The fraction varies from 0% to 50% of faulty sensors, and is shown in Figure 6.8.

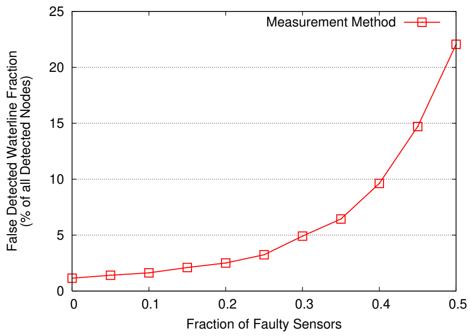
Up to a fraction of 25% faulty sensors there are less than 20 false positives. Even a fraction of 0% results in false detections, but occurs in consequence of the 75% limit on which nodes that have not identified the waterline are allowed to do so. Hence, such a situation occurres only in the near of the waterline and is acceptable. However, from a faulty fraction of 30% on the number of false positives proceeds a steep ascent that results in over 120 nodes. The same process is shown in (b). A value of 50% leads to more than 20% false detections.

The number of false negatives proceeds a linear increasing from 200 nodes to nearly 450. In the consequence of the fraction of failed sensors the nodes are no longer able to provide an accurate identification of the waterline.

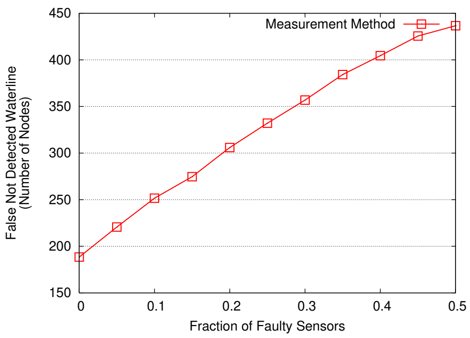
In (e) it is shown that the false positives with 0% faulty sensors occur in the very near of the waterline. The average as well as the maximal distance of such nodes is smaller than the communication range and can thus be tolerated. In general, the average value is acceptable up to a fraction of 30%, and overwhelms the communication range by a mentionable amount only with greater fractions.



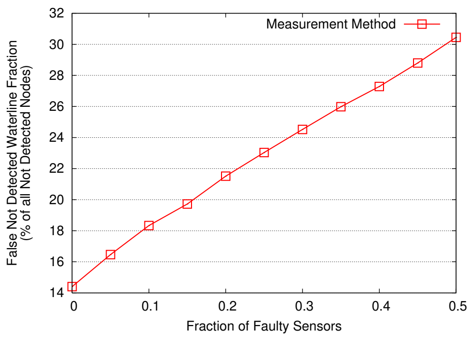
(a) Absolute Node Count of False Positives



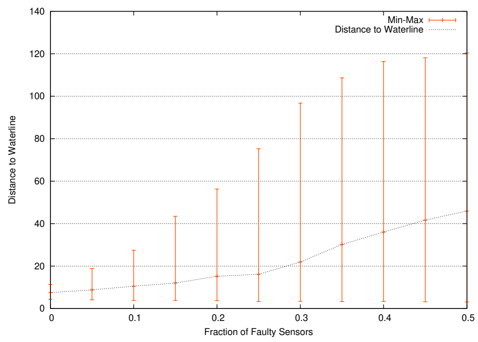
(b) Relative Fraction of False Positives



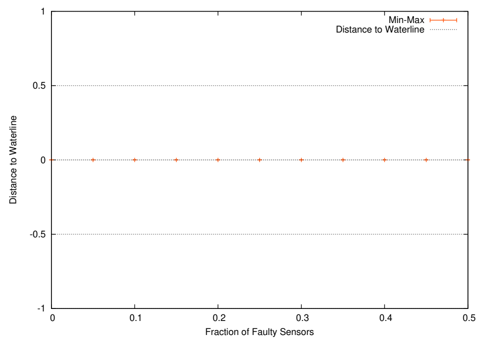
(c) Absolute Node Count of False Negatives



(d) Relative Fraction of False Negatives



(e) Average Distance to Waterline of False Positives



(f) Average Distance to Waterline of False Negatives

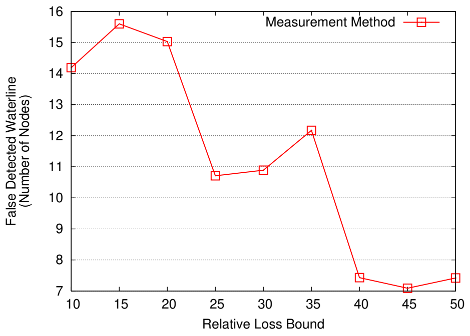
Figure 6.8: Local Measurement: Variation of Random Neighbor Malfunction.

Variation *Relative Bounds* Finally, a variation of the parameters for an identification of the waterline is presented. There are two mentionable parameters. On the one hand, the one that accepts an own detection if at least 30% of its neighbors has done so, too. On the other hand, the one that allows a node to identify the waterline, if itself has not detected such ha occurrence, but 75% of the neighbors have done so indeed. However, both parameters are varied concurrently. If the former one is set to 10%, 15% or 20%, the latter is set to 75%. The same relationship is given between 25%-35% and 80% as well as 40%-50% and 85%. This graduation explains the steps in Figure 6.9 which shows the results of the variation of the parameters.

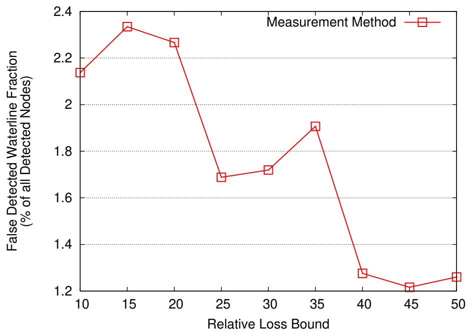
The number of false positives as shown in (a) is decreased from 16 to 7. The first three values are settled down at 15, the next three at approximated 11, and the last ones at 7. The same proceeding occurs in the relative fraction that reaches from 2.4 to 1.2.

An increasing of the parameters leads to a higher number of false negatives. The steps are settled down at 220, 250, and 300 which is proportionally to the relative fraction of 16.5, 18.5, and 21, respectively.

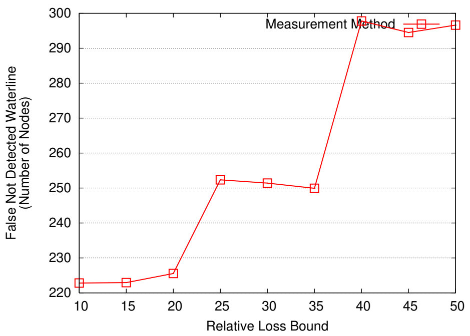
The distance to the waterline of the false positives is hardly affected by a variation of the parameters. The average value is constantly by about 10 units which in turn is approximated half of the communication range. In addition, the maximal distances are also acceptable, because none of them is greater than two times the communication range.



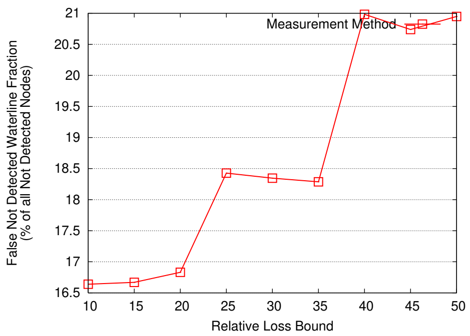
(a) Absolute Node Count of False Positives



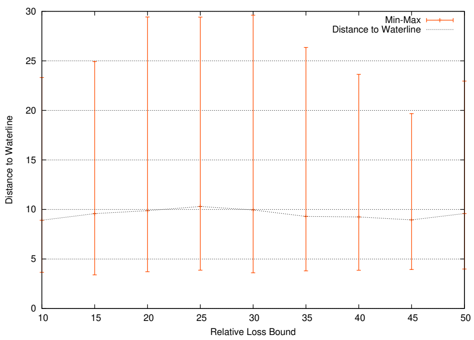
(b) Relative Fraction of False Positives



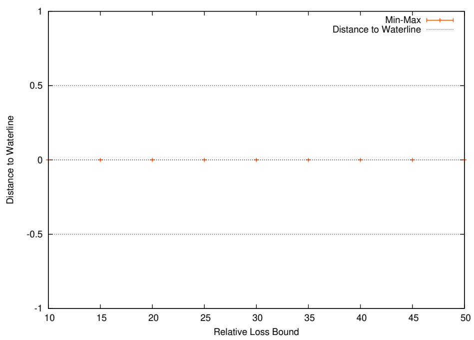
(c) Absolute Node Count of False Negatives



(d) Relative Fraction of False Negatives



(e) Average Distance to Waterline of False Positives



(f) Average Distance to Waterline of False Negatives

Figure 6.9: Local Measurement: Variation of *Relative Bounds*.

6.2 Backbone Algorithms

The backbone algorithms are evaluated in different scenarios by using both waterline identification methods. In addition, the communication range, the velocity of an increasing waterline, and important parameters of the algorithms are varied.

The are several rating factors of interest. Firstly, the remaining energy level of the nodes belonging to the backbone can be compared. Then, the number of messages that are used for coordination differs in the algorithms. Another rating factor is the number of nodes belonging to the backbone as well as the average hop distance to the waterline. Finally, the reliability and velocity of the routing algorithms is of interest. The former is the number of messages that are received by the sink of a routing message, whereas the latter describes the number hops taken by the message compared to the minimal hop distance of source and destination.

At first, the different scenarios are described in Section 6.2.1. Then, examples of both algorithms are presented in Section 6.2.2 and 6.2.3, respectively. Finally, the evaluation is presented in Section 6.2.4.

6.2.1 Scenario Description

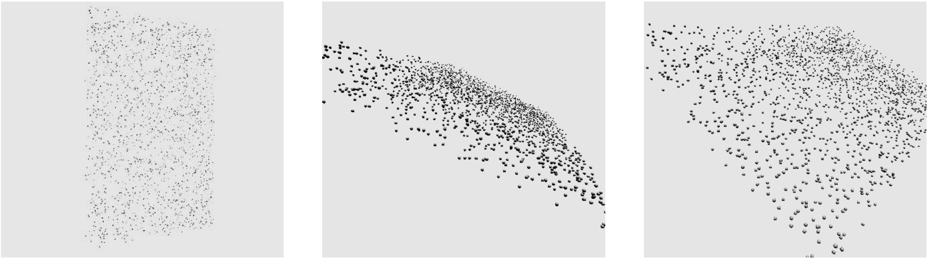
The algorithms have been simulated in three different scenarios: First, the standard scenario that has already been used by evaluating the waterline identification, and is shown again in Figure 6.10. Second, a scenario that contains one missing part in the dike structure as shown in Figure 6.11. Third, a scenario with multiple missing parts which can be seen in Figure 6.12.

The network in each scenario consists of 2000 nodes which are placed randomly in the given region. The communication range is set to 18 units.

In general, the same settings are used as in the scenario for the waterline identification and described in Section 6.1.1. Hence, the transmission of messages is done in a reliable manner. The communication is simulated by a diskgraph model to assure bidirectional connections between neighbored nodes.

The simulation runs for 200 iterations. The water level is static for the first 100 rounds, and increases for the next 80 rounds. The standard alteration height is set to 20 units.

The simulation is repeated 10 times in each scenario which consequently leads to 30 results per parameter variation. Each presented diagram shows the aggregation and average of all scenarios.



(a) Top View (b) Side View (c) Diagonal View

Figure 6.10: Normal Scenario for Backbone Algorithms.

(a) Top View (b) Side View (c) Diagonal View

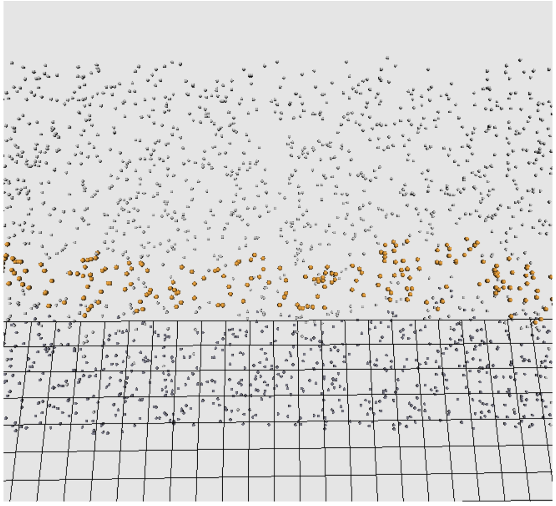
Figure 6.11: Scenario containing one Hole in the Topology.

(a) Top View (b) Side View (c) Diagonal View

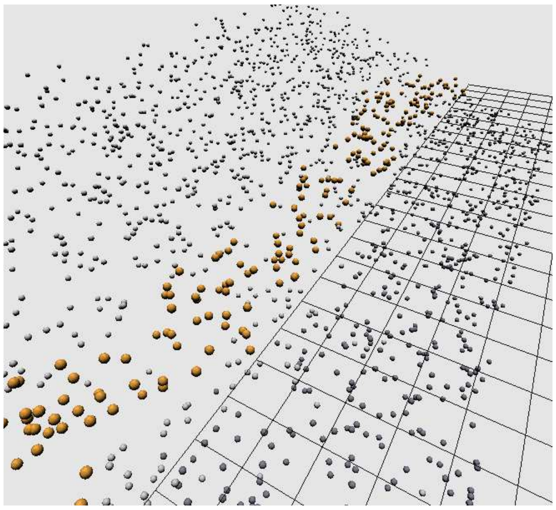
Figure 6.12: Scenario containing multiple Holes in the Topology.

6.2.2 Waterline Stripe

To illustrate the behavior of the *Waterline Stripe*, Figure 6.13 shows an example of a running simulation.



(a) Top View



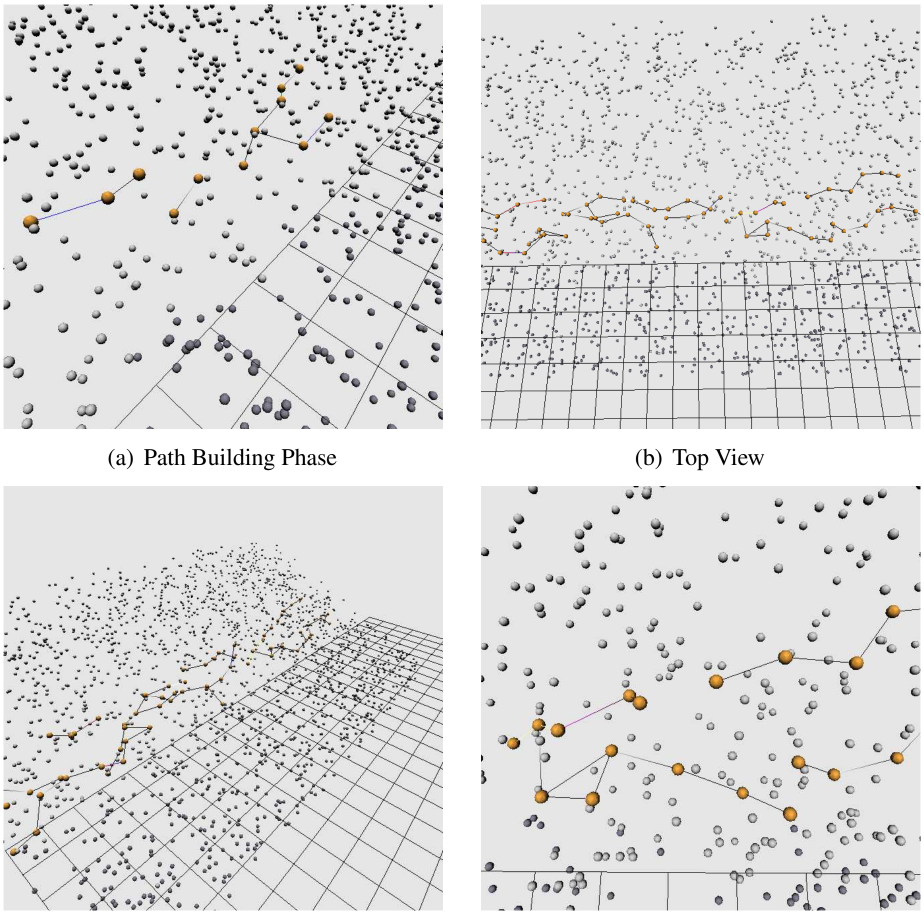
(b) Diagonal View

Figure 6.13: Example of *Waterline Stripe*. *The figures show an example of the Waterline Stripe. The waterline is represented by the wired grid. The orange spheres are the nodes belonging to the backbone structure which is built short above the waterline. The remaining nodes are represented by the grey spheres.*

The waterline is identified by the measurement method. The minimal hop distance is set to 2 hops, and the maximal one to 3 hops. The result is a relative broad structure in which messages can be routed.

6.2.3 Short Path Mix Up

An example of the *Short Path Mix Up* is shown in Figure 6.14. The backbone consists of only a small number of nodes, but builds a continuous connected structure.



(a) Path Building Phase (b) Top View

(c) Diagonal View (d) Cutout

Figure 6.14: Example of *Short Path Mix Up*. *The figure shows an example of the Short Path Mix Up. The wired grid represents the waterline. The orange spheres are the nodes belonging to the communication backbone, whereas the grey ones are the none-backbone ones. In addition, the short paths are illustrated by the lines which connect the appropriate members. In Subfigure (a), the path building phase is shown, whereas (b) and (c) enable the view on a continuous connected backbone. Finally, (d) shows a cutout of the built structure to allow for a more detailed view.*

6.2.4 Comparison of the Algorithms

For the comparison of the both backbone algorithms *Waterline Stripe* and *Short Path Mix Up*, each has been combined with both waterline identification methods *Neighbor Loss* and *Local Measurement*. Hence, there are four different combinations compared.

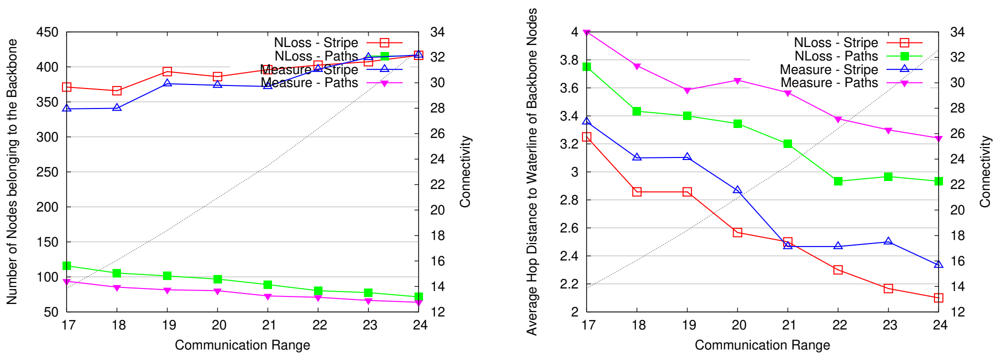
The evaluated parameters are the communication range, the velocity of the increasing waterline, and the minimal and maximal hop distance for the initial backbone building process.

On comparing the number of sent messages, the results include all sent messages that are used for coordination. This includes periodic dummy messages, neighborhood updates, and so on. Such messages do not depend on the used backbone algorithm, and thus the comparison can be done by the difference of additional backbone coordination messages. However, this approach takes advantage of being able to rate also none-backbone nodes.

Variation of the Communication Range

At first, the communication range is varied from 17 to 24. The according connectivity is shown in each diagram.

Figure 6.15 shows the number of nodes belonging to the backbone structure and the average hop distance of the backbone nodes to the waterline.



(a) Number of Backbone Nodes

(b) Average Hop Distance to Waterline

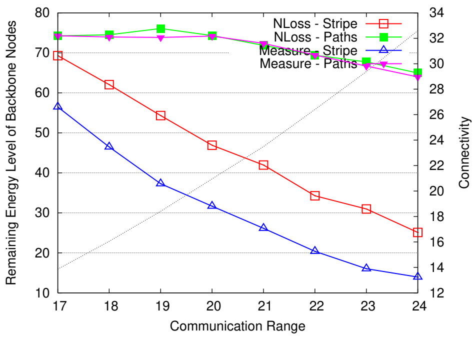
Figure 6.15: Variation of Communication Range: Number of Backbone Nodes and Average Hop Distance to Waterline.

In (a) it can be seen that the *Waterline Stripe* consists of more nodes than the *Short Path Mix Up*. With an increasing communication range it rises from about 350 nodes to 400, nearly independent from the waterline identification method. By using *Neighbor Loss*, there are only a few more nodes belonging to the backbone than in combination with *Local Measurement*. In contrast, the *Short Path Mix Up* consists of about 150 nodes and is decreased 70 with an increasing communication range. Again, the *Neighbor Loss* leads to few more members.

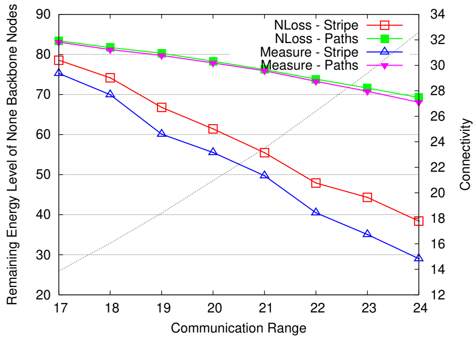
By comparing the average hop distance to the waterline as shown in (b), the *Waterline Stripe* is closer to the waterline than the *Short Path Mix Up*. The hop distance of the stripe is decreased from 3.3 to 2.2, whereas the one of the paths drops from 3.9 to

3.1. In general, the combination with *Neighbor Loss* leads to a smaller distance except for the *Waterline Stripe* by a communication range that is greater or equal 21.

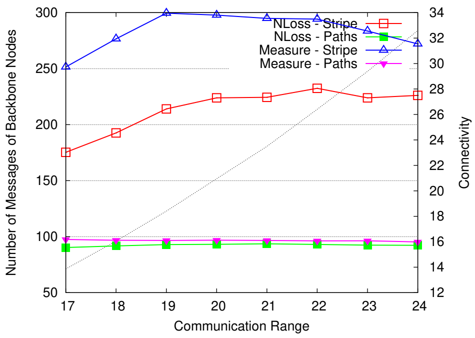
Figure 6.16 shows the remaining energy level and the number of sent messages.



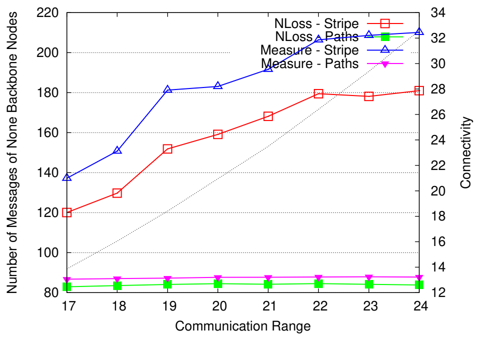
(a) Remaining Energy on Backbone Nodes



(b) Remaining Energy on None-Backbone Nodes



(c) Number of Sent Messages by Backbone Nodes



(d) Number of Sent Messages by None-Backbone Nodes

Figure 6.16: Variation of Communication Range: Energy Level and Number of Sent Messages.

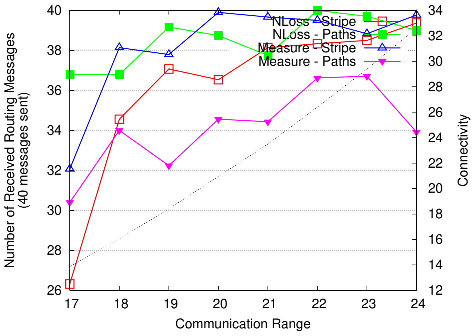
As shown in (a), the usage of *Short Path Mix Up* leads to a higher remaining energy level than the usage of *Waterline Stripe*. The former drops from 75 to 65, whereas the latter is decreases from a mean value of 64 to a mean one of 20. The combination with *Neighbor Loss* leads to a higher level than with *Local Measurement* by using the stripe. In contrast, there is no mentionable difference by using the paths.

The remaining energy level of the none-backbone nodes shown in (b) proceeds similar to the one of the backbone nodes in (a), but each with a greater value of about 10.

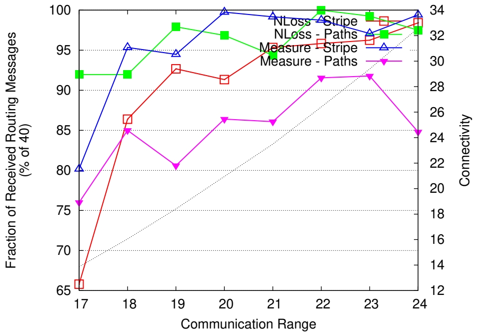
The number of sent messages in (c) makes the *Short Path Mix Up* distinguishable from the *Waterline Stripe*, too. The former proceeding is constantly by about 100 messages in combination with both waterline identification methods. In contrast, the combination of the stripe with *Neighbor Loss* varies from 175 to 220, and in combination with *Local Measurement* from 250 to 300. The latter has a maximum by a communication range of 19, and is then decreased down to 280.

Again, the none-backbone nodes show a similar proceeding, but each with a smaller amount of sent messages. By using the paths the value is constantly by about 80 messages, whereas the usage of the stripe leads to drafts from 120 to 180 and 140 to 210, depending on the used waterline identification method.

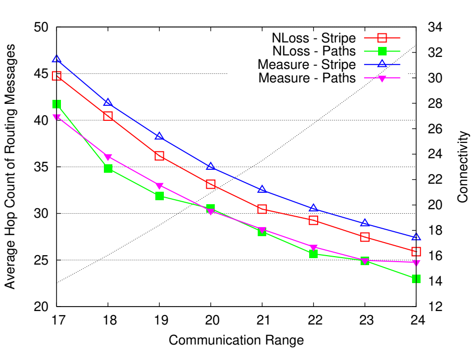
Figure 6.17 shows the results of the routing process.



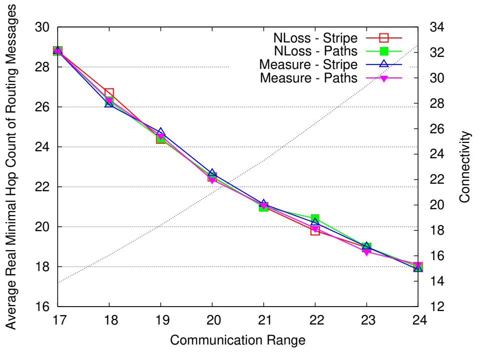
(a) Absolute Number of Received Routing Messages



(b) Relative Fraction of Received Routing Messages



(c) Average Hop Count of Routing Messages



(d) Theoretical Minimal Hop Count of Routing Messages

Figure 6.17: Variation of Communication Range: Routing of Messages.

In (a), the absolute number of received routing messages is shown. Both combinations with the *Waterline Stripe* produce a noticeable reliability with an increasing communication range. The combination with *Neighbor Loss* rises from 26 messages to 40, the combination with *Local Measurement* from 32 to 40. The latter reaches 40 messages by a communication range of 20. The *Short Path Mix Up* in combination with *Local Measurement* varies from 30 to 37, whereas the combination with *Neighbor Loss* varies from 37 to 40. In (b), the results are shown as the relative fraction of received routing messages, and produce the same proceedings.

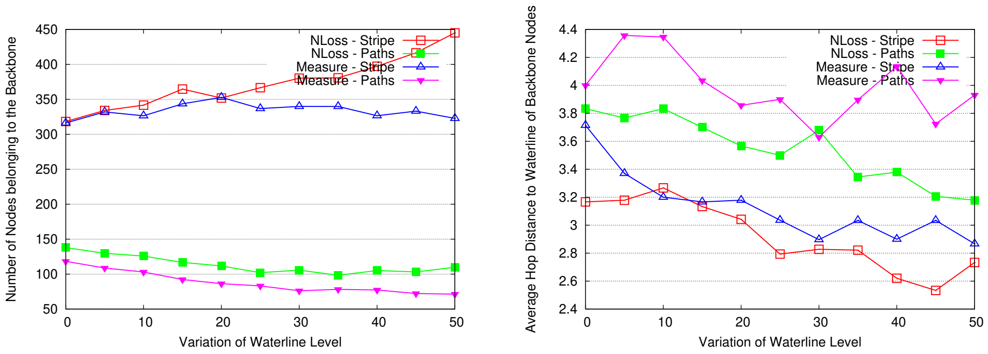
The average hop count of routing messages decreases with an increasing communication range as shown in (c). The hop count of the *Short Path Mix Up* drops from 40 to 24 nearly analog with both waterline identification methods. The *Waterline Stripe* in combination with *Neighbor Loss* drops from 45 hops to 26, whereas the combination with *Local Measurement* falls from 46 to 27 in parallel. The theoretical minimal hop distance between the source and the sink is shown in (d) and drops from 29 to 18.

Variation of Velocity and Height of the Increasing Waterline

The velocity of the increasing waterline has been simulated by varying the overall height of the waterline. In the standard case, the increasing of the waterline starts at iteration 100. Then, the height rises by 20 units for 80 rounds. This results in 0.25 per round.

To simulate different water levels, the parameter of the overall height varies from 0 to 50, each for 80 rounds. The latter results in 0.625 per round.

Figure 6.18 shows the number of nodes belonging to the backbone structure and the average hop distance of the backbone nodes to the waterline.



(a) Number of Backbone Nodes (b) Average Hop Distance to Waterline

Figure 6.18: Variation of Waterline: Number of Backbone Nodes and Average Hop Distance to Waterline.

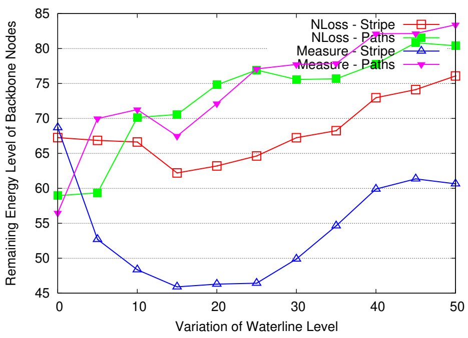
The number of nodes belonging to the backbone structure as shown in (a) points the differences between the *Short Path Mix Up* and the *Waterline Stripe* out again. The former varies from 140 to 100 in combination with *Neighbor Loss*, and from 120 to 70 with *Local Measurement*. In contrast, the stripe varies from 310 to 350 in combination with *Local Measurement*, and from 310 up to 450 with *Neighbor Loss*.

By analyzing the average hop distance to the waterline in (b), the *Waterline Stripe* combinations are closer than the *Short Path Mix Up* ones. Applying together with *Local Measurement*, it drops from 3.7 to 2.9, and from 3.2 to 2.6 with *Neighbor Loss*. The paths drop from 3.8 to 3.2 combined with *Neighbor Loss*, and vary from 3.6 to 4.4 with the measurement method.

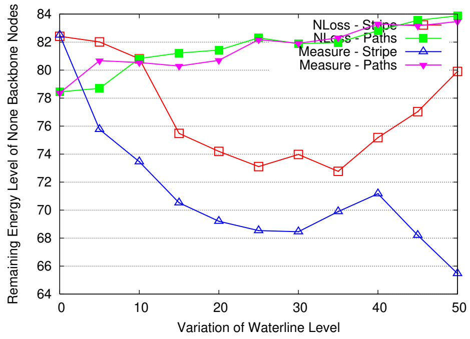
Figure 6.19 shows the remaining energy level and the number of sent messages.

Except for the static waterline with a variation of 0, the *Short Path Mix Up* leads to a greater amount of remaining energy than the *Waterline Stripe*. The former rises from 58 to 81 with similar results for both waterline identification methods. The stripe combined with *Neighbor Loss* varies from 62 to 76, and with *Local Measurement* it drops from 69 to 60 by having a minima of 45 at a waterline alteration level from 15 to 25.

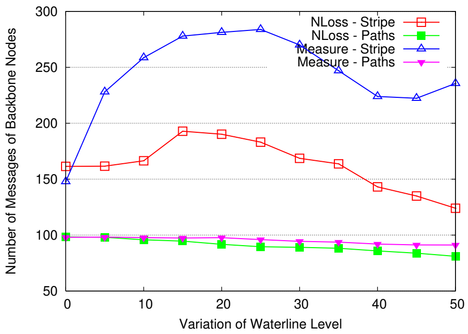
There are similar results for the none-backbone nodes, but having greater remaining amounts of energy. The *Waterline Stripe* applied together with *Local Measurement* drops from 82 down to 66, whereas the combination with *Neighbor Loss* falls from 82 to 80 with minima of 73 by an alteration of 25 to 35. In contrast, the *Short Path Mix*



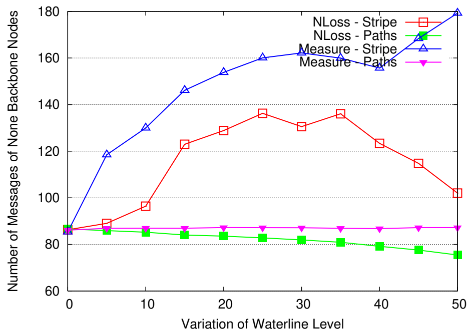
(a) Remaining Energy on Backbone Nodes



(b) Remaining Energy on None-Backbone Nodes



(c) Number of Sent Messages by Backbone Nodes



(d) Number of Sent Messages by None-Backbone Nodes

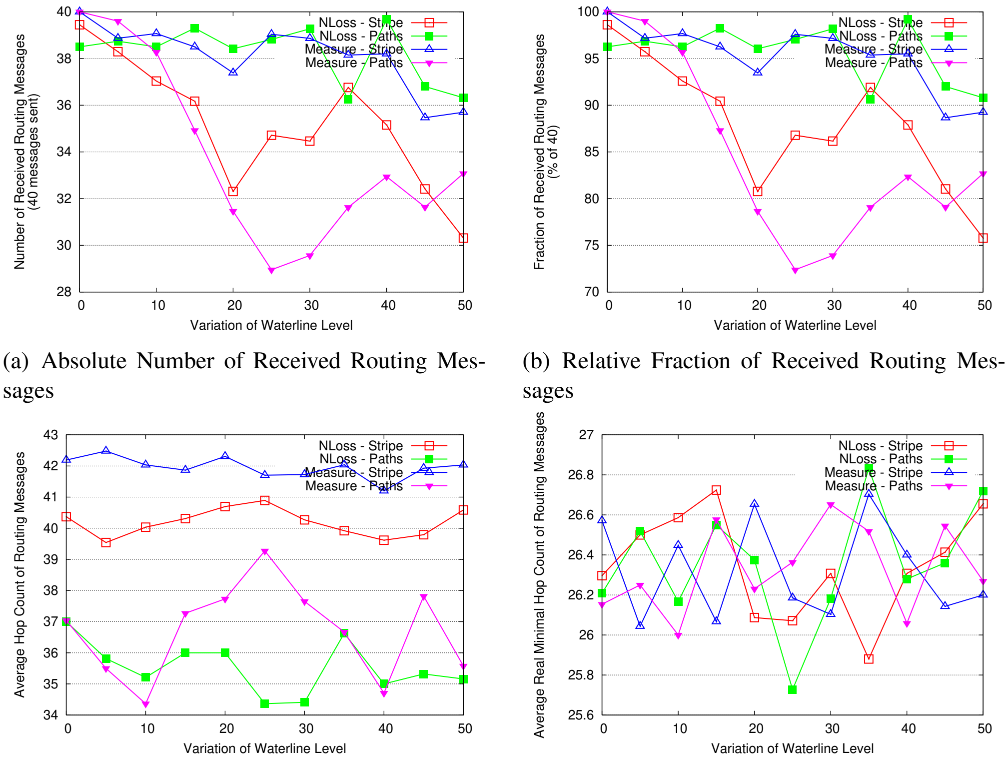
Figure 6.19: Variation of Waterline: Energy Level and Number of Sent Messages.

Up leads to an increasing remaining energy level. Both combinations rise from 78 to 84.

The number of sent messages of backbone nodes is nearly constant for the *Short Path Mix Up* and decreases from 100 to 90 in both combinations. The *Waterline Stripe* applied together with *Neighbor Loss* varies from 200 to 125 by starting at 160. Combined with *Local Measurement* it proceeds from 150 to 240 by having a maximum of 290 at an alteration level of 25.

The *Short Path Mix Up* leads again to nearly constant results for the number of sent messages for none-backbone nodes. Combined with *Local Measurement* is permanently at 85, whereas the combination with *Neighbor Loss* drops from 85 to 78. In contrast, the *Waterline Stripe* applied together with *Local Measurement* increases from 85 up to 180. Combined with *Neighbor Loss* it proceeds from 85 to 100 with maxima of nearly 140 by waterline alterations from 25 to 35.

Figure 6.20 shows the results of the routing process.



(a) Absolute Number of Received Routing Messages

(b) Relative Fraction of Received Routing Messages

(c) Average Hop Count of Routing Messages

(d) Theoretical Minimal Hop Count of Routing Messages

Figure 6.20: Variation of Waterline: Routing of Messages.

The number of received routing messages shows no advantage of a special backbone algorithm. The *Short Path Mix Up* combined with *Neighbor Loss* varies from 36 to 40. A similar result with the same variation is obtained by applying the *Waterline Stripe* together with *Local Measurement*. In contrast, the *Short Path Mix Up* combined with *Local Measurement* proceeds from 40 to 33 with a minimum of 29 at the waterline alterations 25 and 30. The *Waterline Stripe* and *Neighbor Loss* drop from about 40 to 30 with a local maximum of 37 at an alteration level of 35, and a local minimum of 32 at 20. Again, the same results are shown in (b) with the relative fraction.

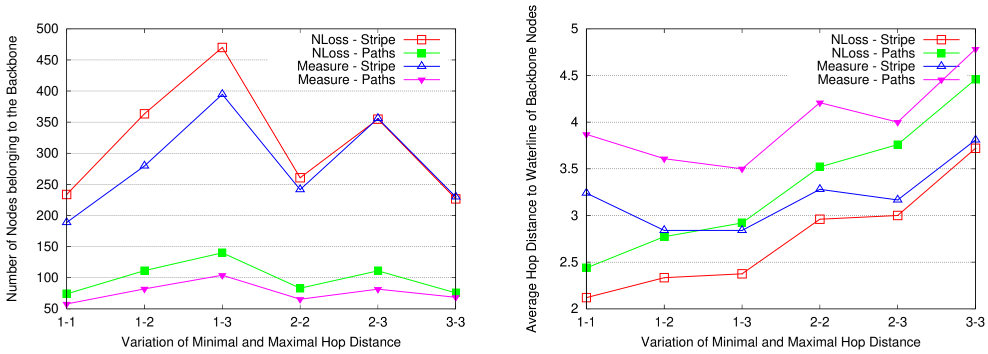
The average hop count of the routing messages is shown in (c). The *Waterline Stripe* combined with *Local Measurement* leads to a nearly constant value of 42. If the *Neighbor Loss* method is applied, the stripe uses a nearly constant hop count of 40. The *Short Path Mix Up* proceeds from 37 to 35 in both combinations, but with different amplitudes.

The theoretical minimal hop distance between source and destination is approximated 26.4 as shown in (d).

Variation of the Hop Distance to the Waterline

Finally, the minimal and maximal hop distance to the waterline for the initial phase of the backbone algorithms is varied. The appropriate values are shown in the diagrams on the x-axis with the minimum as the first value and the maximum as the second one, respectively.

Figure 6.21 shows the number of nodes belonging to the backbone structure and the average hop distance of the backbone nodes to the waterline.



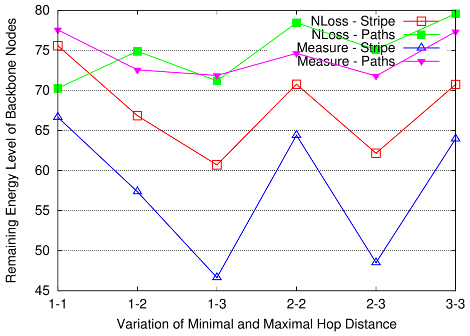
(a) Number of Backbone Nodes (b) Average Hop Distance to Waterline

Figure 6.21: Variation of Distance Parameters: Number of Backbone Nodes and Average Hop Distance to Waterline.

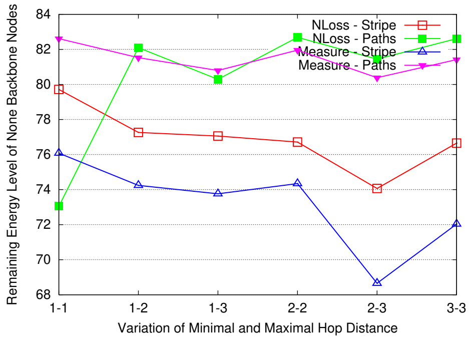
The number of nodes belonging to the backbone as shown in (a) depends on the span of the minimal and maximal hop distance. The *Short Path Mix Up* combined with *Local Measurement* leads to the smallest values. It varies from 50 to 100 nodes, depending on the given span. Applied together with *Neighbor Loss*, the variation is from 75 to 150. The *Waterline Stripe* combined with the measurement method varies from 200 to 400, and with *Neighbor Loss* from 250 to 475.

The average hop distance to the waterline depends particularly on the given minimal parameter. For the *Waterline Stripe* combined with *Neighbor Loss*, it increases from 2 to 3.6, together with *Local Measurement* from about 3 to 3.8. The *Short Path Mix Up* applied together with *Neighbor Loss* rises from 2.5 to 4.5, and with *Local Measurement* from about 3.5 to 4.8.

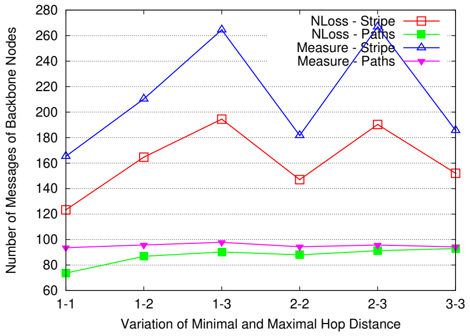
Figure 6.22 shows the remaining energy level and the number of sent messages.



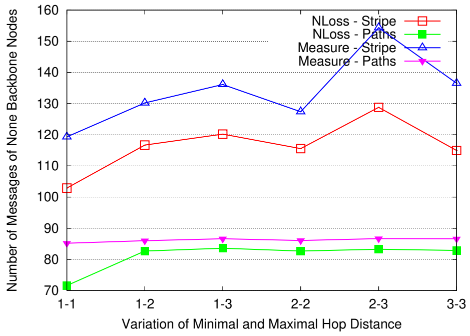
(a) Remaining Energy on Backbone Nodes



(b) Remaining Energy on None-Backbone Nodes



(c) Number of Sent Messages by Backbone Nodes



(d) Number of Sent Messages by None-Backbone Nodes

Figure 6.22: Variation of Distance Parameters: Energy Level and Number of Sent Messages.

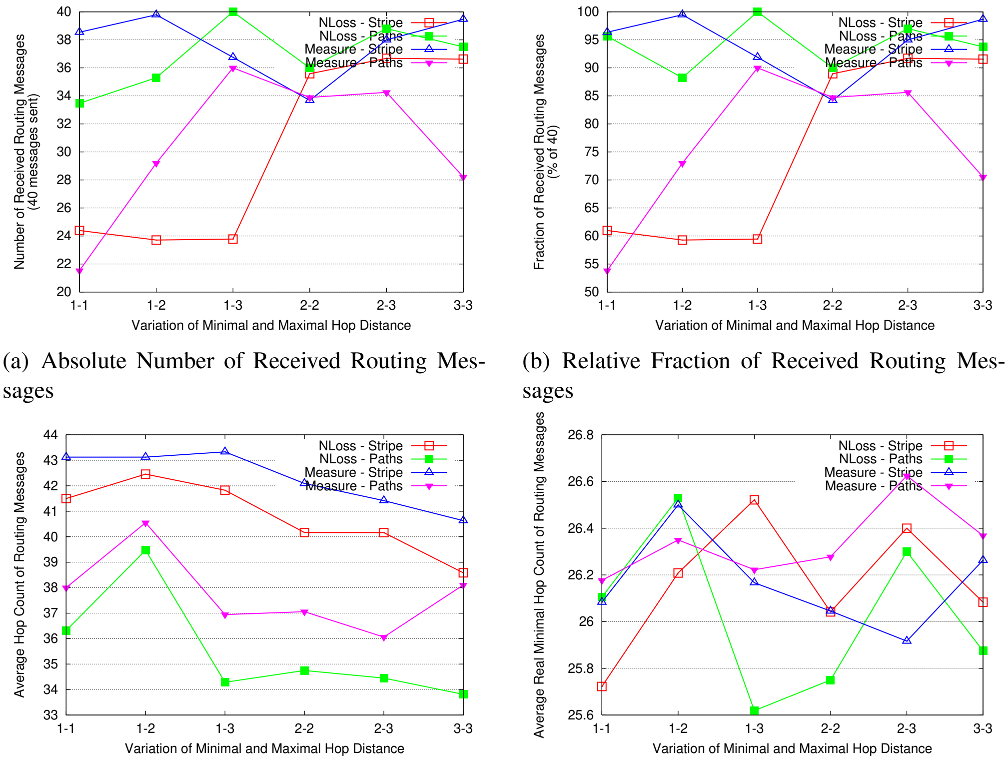
The remaining energy of the backbone nodes as shown in (a) depends on the used parameters for the *Waterline Stripe*, whereas the *Short Path Mix Up* is nearly not affected. The latter varies from 70 to 80 for both combinations, and produces only marginal differences. The *Waterline Stripe* varies from 65 down to 45 for the combination with *Local Measurement*, and from 75 down to 60 with *Neighbor Loss*.

Except for a minimal as well as maximal hop distance of 1, the *Short Path Mix Up* can be averaged to a remaining energy level of 82 for the none-backbone nodes. The *Waterline Stripe* combined with *Neighbor Loss* decreases from 80 to 76 but with a local minimum of 74. The combination with *Local Measurement* proceeds similar, but from 76 to 72 and a local minimum of 68.

The number of sent messages of backbone nodes is nearly constant at 90 for the *Short Path Mix Up* in both combinations. Depending on the parameters, the *Waterline Stripe* varies from 120 to 200 if applied together with *Neighbor Loss*, and from 160 to 260 combined with *Local Measurement*.

The number of sent messages for none-backbone nodes proceed similar. The *Short Path Mix Up* can be averaged at 85, and *Waterline Stripe* depends again the used parameters. Combined with *Neighbor Loss* it varies from 100 to 130, and in combination with *Local Measurement* it varies from 120 to 150.

Figure 6.23 shows the results of the routing process.



(a) Absolute Number of Received Routing Messages

(b) Relative Fraction of Received Routing Messages

(c) Average Hop Count of Routing Messages

(d) Theoretical Minimal Hop Count of Routing Messages

Figure 6.23: Variation of Distance Parameters: Routing of Messages.

In (a) the number of received routing messages is shown. The *Short Path Mix Up* combined with *Neighbor Loss*, and the *Waterline Stripe* applied together with *Local Measurement* show the best results by varying from 34 to 40, each with different advantages in parameter settings. The *Short Path Mix Up* in combination with *Local Measurement* proceeds best on "1-3", "2-2", and "2-3" with a result of 34 to 36. For "1-2" and "3-3" there are 29 and 28, respectively, messages received, and for "1-1" only 22. The *Waterline Stripe* combined with *Neighbor Loss* leads to a value of 24 if the minimal hop distance is set to 1, and 36 otherwise. The relative fraction of received messages is shown in (b), and shows the same proceedings.

In (c), the average hop count of the routing messages is shown. The results of the stripe in combination with *Local Measurement* are decreased from 43 to 41. The combination with *Neighbor Loss* leads to 42 down to 39 hops. The *Short Path Mix Up* combined with *Neighbor Loss* starts with 36 hops at "1-1", rises to 39 at "1-2", and is then constantly at about 34 hops. Combined with *Local Measurement*, it starts at 38, rises to 40, and can be averaged to 37 for the following.

The theoretical minimal hop distance between source and destination is approximated 26 as shown in (d).

7 Conclusion

This thesis examined the potential usage of sensor networks in flood protection. It has been assumed that the nodes are therefore inserted into sandbags which in turn are used to protect an area against flooding. They are at most able to measure the actual wetness of the surrounding sand. In addition, they either lose their communication ability due to physical restrictions, or fail on an increasing moisture penetration if not water resistant. Consequently, a valuable idea is to exhaust particularly nodes close to the water level by routing messages through the network. Such an approach requires the construction of a continuous connected communication backbone, and the definition of a routing algorithm that operates on the structure. Finally, the nodes are not aware of their global position, neither in an absolute nor in a relative coordinate system. Hence, the algorithms must be designed in a strict local manner, and are only allowed to use the information of their immediate neighbors.

This thesis presented different algorithms for the construction of such a communication backbone by separating the several tasks. Firstly, two methods of waterline identification were designed, each with a particular main focus on the behavior of the nodes. The *Neighbor Loss* method takes only the loss of communication to the immediate neighbors into account, whereas the *Local Measurement* requires the availability of appropriate sensors. Both methods were extended by additional conditions to improve the robustness against false detections. Secondly, a coordination method was designed by taking only the minimal hop distance to selected reference points into account. The main design goal hereby was robustness as well as simplicity. Finally, two different backbone algorithms were presented, each with an adapted routing method. The main design goal of the *Waterline Stripe* was a wide backbone structure that provides a preferably simple coordination phase. Consequently, the routing method operates on the comparative high density and provides a load balance when forwarding messages. In contrast, the *Short Paths* method was designed for a more reliable backbone structure. Each node is associated with at least one successor and one predecessor. The paths are maintained as local as possible, but provide a continuous connected backbone. The routing method makes use of the given structure and operates in a simple but reliable manner.

The designed algorithms were implemented and simulated in the discrete event simulator `Shawn`. The evaluation has shown that the waterline identification succeeds in an acceptable area. On greater perturbations, the methods fail in the consequence of the strict local decision process. To avoid such situations, the methods must be extended to operate over multiple hops and gather the useful information. However, the *Local Measurement* is more reliable than the *Neighbor Loss* method, because false detections occur mostly in the near of the waterline, whereas the false ones in the latter method are spread over the network. Furthermore, also the backbone algorithms have shown different results. In general, the *Short Paths* should be preferred to the *Waterline Stripe*. Nevertheless, the reliability of both algorithms can be improved, especially with respect to a rapidly increasing waterline.

Bibliography

[Bra03] Landesumweltamt Brandenburg. *Hochwasserschutz in Brandenburg*. Land Brandenburg - Ministerium für Landwirtschaft, Umweltschutz und Raumordnung, 2003.

[BW99] Peter G. Burnett and John J. Wanner. Reducing leakage through sandbag dikes using a bentonite or other clay mud slurry. Patent, June 1999.

[CDA03] Shruti Chugh, Sagar Dharia, and Dharma P. Agrawal. An energy efficient collaborative framework for event notification in wireless sensor networks. In *LCN*, pages 430–. IEEE Computer Society, 2003.

[CE04] Alberto Cerpa and Deborah Estrin. Ascent: Adaptive self-configuring sensor networks topologies. *IEEE Trans. Mob. Comput.*, 3(3):272–285, 2004.

[CIE00] R. Govindan C. Intanagonwiwat and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 56–67, Boston, MA USA, 2000.

[CJBM01] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *Mobile Computing and Networking*, pages 85–96, 2001.

[Dav64] Rotislaw Davidenkoff. *Deiche und Erdämme - Sickerströmung, Standsicherheit*. Werner-Verlag, Düsseldorf, 1964.

[EA03] Bruce Eckel and Chuck Allison. *Thinking in C++. Volume 2: Practical Programming*. Prentice Hall, 2003.

[Fle02] G. Fleming. *Flood Risk Management*. Thomas Telford Publishing, 2002.

[Gad88] Peter E. Gadd. Sand bag slope protection: Design, construction, and performance. In *Arctic Coastal Processes and Slope Protection Design*, pages 145–165, New York, 1988.

[Gra05] Michael Grabe. *Measurement Uncertainities in Science and Technology*. Springer-Verlag, 2005.

[IPPR03] Prakash Ishwar, Rohit Puri, S. Sandeep Pradhan, and Kannan Ramchandran. On rate-constrained estimation in unreliable sensor networks. In Zhao and Guibas [ZG03], pages 178–192.

[JM96] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Thomasz Imielinski and Hank Korth, editors, *Mobile Computing*, volume 353, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[KI03] Bhaskar Krishnamachari and S. Sitharama Iyengar. Efficient and fault-tolerant feature extraction in wireless sensor networks. In Zhao and Guibas [ZG03], pages 488–501.

[KJ85] Nobuhisa Kobayashi and Brian K. Jacobs. Experimental study on sandbag stability and runup. In *Proceedings of the Fourth Symposium on Coastal and Ocean Management, Coastal Zone '85, ASCE*, volume 2, pages 1612–1626, Baltimore, MD, 1985.

[KSPSV02] Farinaz Koushanfar, Sasha Slijepcevic, Miodrag Potkonjak, and Alberto Sangiovanni-Vincentelli. Error-tolerant multi-modal sensor fusion, 2002.

[Kup97] Klaus Kupfer. *Materialfeuchtemessung - Grundlagen, Messverfahren, Applikationen, Normen*. expert-Verlag, 1997.

[NDH$^+$03] Tim Nieberg, Stefan Dulman, Paul Havinga, Lodewijk van Hoesel, and Jian Wu. Collaborative algorithms for communication in wireless sensor networks. In *Ambient intelligence: impact on embedded system design*, pages 271–294. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[Ohl04] Christoph Ohlig. *Wasserhistorische Forschungen - Schwerpunkte Hochwasserschutz/Elbe*. Schriften der Deutschen Wasserhistorischen Gesellschaft (DWhG)e. V., 2004.

[PP92] Paul Profos and Tilo Pfeifer. *Handbuch der industriellen Messtechnik, 5. Auflage*. R. Oldenbourg Verlag GmbH, 1992.

[RMS03] RMS. Central europe flooding, august 2002. Technical report, Risk Management Solutions, 2003.

[SCC04] H.O. Sanli, H. Cam, and X. Cheng. Eqos: An energy efficient qos protocol for wireless sensor networks. In *Proc. of 2004 Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'04)*, 2004.

[YF04] Ossama Younis and Sonia Fahmy. HEED: A hybrid, energy-efficient, distributed clustering approach for ad-hoc sensor networks, 2004.

[ZG03] Feng Zhao and Leonidas J. Guibas, editors. *Information Processing in Sensor Networks, Second International Workshop, IPSN 2003, Palo Alto, CA, USA, April 22-23, 2003, Proceedings*, volume 2634 of *Lecture Notes in Computer Science*. Springer, 2003.