

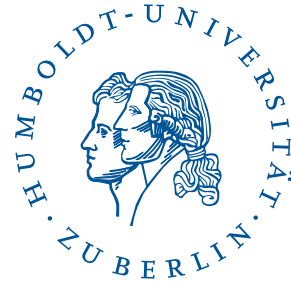
# Institut für Informatik

HUMBOLDT-UNIVERSITÄT ZU BERLIN

Unter den Linden 6

10099 Berlin

Germany



---

**Studienarbeit**

## **Distributed Calculation of Local Averages in Sensor Networks**

**Björn Schümann**

[schuemann@gmail.com](mailto:schuemann@gmail.com)

---

Betreuer : Timo Gläßer, Prof. Ulf Leser  
Wissensmanagement in der Bioinformatik  
Eingereicht am : 26. September 2008



Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt zu haben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

Berlin, 26. September 2008



---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	Objective . . . . .	2
2.2	Local Aggregation Queries . . . . .	3
<b>3</b>	<b>Theoretical framework</b>	<b>5</b>
3.1	Parameters of the world . . . . .	5
3.2	Analysis of four different approaches . . . . .	7
3.3	Comparison of the different approaches . . . . .	11
<b>4</b>	<b>Implementation</b>	<b>12</b>
4.1	Preliminaries . . . . .	12
4.2	Architecture . . . . .	16
4.3	Message types . . . . .	17
4.4	Process of the query execution . . . . .	19
4.5	Application classes . . . . .	20
4.6	Routing . . . . .	21
<b>5</b>	<b>Evaluation</b>	<b>24</b>
5.1	Parameters . . . . .	25
5.2	Results of the Simulation . . . . .	26
<b>6</b>	<b>Conclusion</b>	<b>32</b>
6.1	Open questions . . . . .	33
	<b>Bibliography</b>	<b>35</b>



# 1 Introduction

In recent years Wireless Sensor Networks have become an important and useful tool for a wide area of applications. WSN consist of a bulk of sensor nodes, i.e., small battery powered devices equipped with a set of sensors and a wireless communications interface. WSNs have been used in applications like monitoring the habitats of birds [SPMC04], the structural health of buildings [KPC<sup>+</sup>07] or the areas near eruptive volcanoes [WAJR<sup>+</sup>05].

Since an increasing number of sensor nodes provides a finer resolution of data points the networks are getting bigger. Sharing workload between nodes provides opportunities to extend the lifetime of the network and reduces the risk of data loss in case of failing nodes[CD05]. Previous solutions tried to execute most of the processing workload at the gateway node. This often leads to a congestion of workload on some nodes, resulting in a high energy consumption on the nodes. This becomes unfeasible and inefficient in growing networks. As a consequence the processing of the raw data in the nodes becomes more and more important. At the same time the nodes are build on hardware platforms that have more processing power and memory, making query processing in the nodes feasible.

Large-scale networks have to be flexible enough to handle changes in the network topology, caused for example by failing and moving nodes and require the ability to run multiple queries from different users distributed over different gateway nodes. There is still a need for better tools providing a simple way of programming networks that fulfil these requirements for data gathering and monitoring tasks.

Finding an efficient way of answering queries over data in a specific subarea in these networks is one of the challenges. With an increasing number and density of nodes the single sample of a single node gets less and less important. Integrating the samples of a group of nodes often improves the data quality. Consequently aggregating and averaging over the data of the nodes surrounding a specific location is a common and important but expensive task. Figure 2.1 shows a possible situation. The query needs to be routed on a direct path to the area in question. The nodes located there can work together to process the data, before the answer is sent back. There are a lot of ways to route, spread and process the query and finding an efficient and flexible solution is an interesting and demanding problem.

## 2 Background

Various architectures have been proposed as a Middleware for Sensor Networks. They try to close the gap between the users questions and applications and the low level programming of the sensor devices. The ideas range from special languages [MWM06] and compilers to virtual machines running a specific byte code interpreter[LC02].

Cougar [YG02] and later TinyDB [MFHH03], introduced a new dimension into Middleware research by proposing a database approach. Instead of writing embedded C Code the user can use a SQL-like interface to extract the data from the network. Sensor readings are represented in a virtual database table. A gateway node parses SQL-like queries and distributes an optimized query execution plan to all nodes. Each sensor node has its own query processor and processes the sensor data before sending his part of the answer to the parent node.

TinyDB introduced many important ideas for query optimization in sensor networks, but the project was finished in 2003. The project is only weakly documented and very difficult to port to new sensor platforms [MA05]. TinyDB uses a central approach for parsing and optimizing in the gateway node restricts the optimization to single queries. Multiple queries running simultaneously are processed independently including duplicate sampling of sensors and transmission of results.

### 2.1 Objective

A query processing system should provide the basis to run distributed aggregation queries in a specific subarea of the network and return the answer to all nodes subscribed to this query.

The objective of this thesis is to develop and compare different execution plans for queries testing the local average around each node in a specific area of the network. Testing local averages or other aggregation directly in the subarea of interest could reduce the transmissions to the gateway node, but computing an aggregated average from different nodes demands a lot of transmissions between the nodes in the area. Finding a way to get the results for

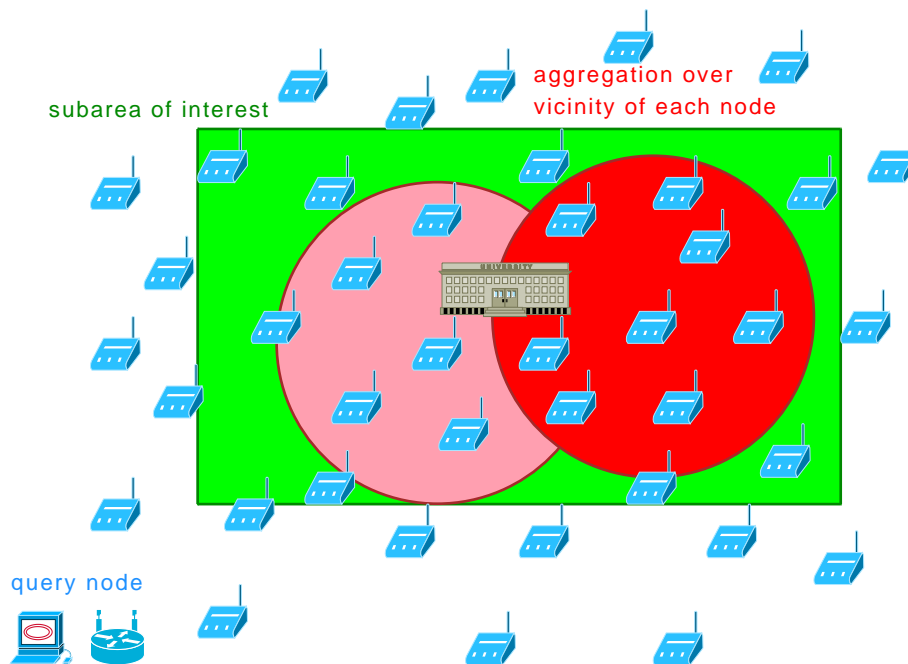


Figure 2.1: Concept of local aggregation: Aggregation over the vicinity of each node to improve data quality. Instead of using a single sensor reading we aggregate over the local neighbourhood of each node. We only trigger a event, if the aggregation result fulfills a certain condition.

each node with less transmissions saves energy and therewith extends the lifetime of each node.

## 2.2 Local Aggregation Queries

Testing an aggregation result would be a **HAVING**-clause in a SQL-Query. Finding nodes with an high local average temperature in their vicinity, would be an example of such a query listed in Listing 2.2.

In this thesis we study queries of this structure. The variable parameters are:

- the **size of the query region**

This query area defines which part of the world the user is interested. In the query above this would be the rectangular area between the points  $(x_1, y_1)$  and  $(x_2, y_2)$ .

- the **aggregation radius**

The data of all nodes within this radius will be included in each aggregation.

- the selectivity of the **HAVING**-condition

---

```
SELECT A.LOC, AVG(B.TEMP) FROM Nodes A, Nodes B
WHERE A.LOC IN (1000,1000,2000,2000)
      AND B.LOC IN (1000,1000,2000,2000)
      AND DIST(A.LOC, B.LOC) < 100
GROUP BY A.LOC
HAVING AVG(B.TEMP) > 50
SAMPLE PERIOD 60s FOR 24h
```

---

Listing 2.1: An Example query testing the average temperature 100m around each node in a specific subarea of the network

This selectivity will control how many result tuples have to be transmitted. Each result above **threshold** will result in an event to be submitted to the sink and displayed at the user terminal.

- The **SAMPLE PERIOD** splits the lifetime of the query in epochs. The processing of each epoch is independent from its preceding epochs. Hence no data has to be stored between the epochs.

Clearly the network topology, the total number of nodes and the distance between the query region and the sink node will influence the performance of the query execution. Concentrating on this kind of queries and comparing the performance of different execution plans under different sets of parameters should provide useful information on how to find a optimal execution plan for a specific local average query.

## 3 Theoretical framework

This section will state some assumptions used in the following discussion and the simulation.

The network consists of regular and dense spread sensor nodes capable of sensing at least one environment parameter. The following queries will use the temperature as environmental parameter. The nodes can communicate with each other using a wireless network. The network should be dense enough, so that each node has at least one neighbour within radio range in each direction. A dense networks simplifies the routing algorithm. More complex routing algorithms as described in [Hei05] can be combined with the results of this thesis to reroute messages around voids in sparse networks.

Each node knows its own position in a 2D grid and can use this information to identify relevant queries restricted to a specific location or area. Furthermore each node knows the position of all neighbor nodes which it can communicate with (1 hop distance). This information facilities efficient routing as shown later. In a static network injecting the position information into each node may be part of the setup procedure. Mobile nodes can use GPS or other systems to determine the location in the grid.

### 3.1 Parameters of the world

The following parameters listed in Table 3.1 should help to estimate the transmission costs of different ways to execute the queries.

Let  $l_r$  be the range of the radio. This is the maximal distance two nodes can be apart, but still be able to communicate reliably and bidirectional with each other. Our theoretical model will assume a stable communication without packet loss and congestion. In case we have to transmit a message over some distance  $\omega > l_r$  intermediate nodes have to relay the message. Let  $h(\omega)$  be the average number of relay hops needed to transmit a message over a distance  $\omega$ . Using an efficient geographic routing algorithm in a dense network this value

$l_r$	radio range
$l$	length of the quadratic query area
$n$	number of nodes in the query area
$s$	selectivity of the query
$r$	aggregation radius
$d(c, s)$	distance between the center of the query area and the sink
$d_s$	avg. distance between the nodes in query area and the sink
$d_c(l)$	avg. distance nodes in a square of length $l$ have to the center
$h(\omega)$	retransmission hops needed to cover a distance $\omega$

Table 3.1: Parameters and functions used in the analysis

should usually be close to  $\frac{\omega}{l_r}$  or at least be linear in  $\omega$ . Using  $h(\cdot)$  we can estimate the number of transmissions needed to transmit a message between two nodes by their distance.

Our model will estimate the number transmission needed to process one epoch of a single query with the structure described in Section 2.2. Let  $(x_s, y_s)$  be the position of the sink node. The sink node is usually the gateway node, where the user injected the query into the network. It can also be every other node in the network which decided to subscribe to the query to act on these events or to use the data to derive other events. The query defines a quadratic area<sup>1</sup> of the world as query area. Let  $l$  be the side length of this area.

Let  $n$  be the total number of nodes in the query area. Nodes outside of this area won't sample their sensors and are only used to transmit messages to the sink.

Whenever the local aggregation around a node fulfills the **having** condition of the query, an event message with the result has to be transmitted to the sink node. Let the selectivity  $s$  be the fraction of all query executions in the network resulting in such a event. The value of  $s$  will obviously not be constant over the lifetime of the query, but an estimation is  $s = \frac{\text{number of events}}{\text{number of recent query executions}}$ . In our queries this would depend on the *THRESHOLD* parameter of the **having** condition. A high *THRESHOLD* would produce less resulting events and therefore have low selectivity.

Let  $d_s$  be the average distance between each node in the query area and the sink. The exact value of  $d_s$  depends on where the sink node is located relatively to the area. Assuming that the sink node is outside of the query area in some distance away one can take so following estimation.  $d_s = \sqrt{\left(\frac{l}{4}\right)^2 + d(c, s)^2}$ , where  $d(c, s) = \sqrt{(x_c - x_s)^2 + (y_c - y_s)^2}$  is the distance between the sink node and the center of the area  $(x_c, y_c)$  and  $\frac{l}{4} = \int_{-\frac{l}{2}}^{\frac{l}{2}} |x| \frac{1}{l} dx$  is the approximation of the average offset the nodes have from the direct line between the center

<sup>1</sup>We stick to quadratic query areas only to keep the formulas simply. The implementation allows any polynomial area.

and the sink. When  $l$  is small compared to  $d(c, s)$  we can approximate  $d_s \approx d(c, s)$ .

Later we will need an estimation of the average distance each node in query area has to the center of the area. Let  $(x, y)$  be a two-dimensional random variable holding the relative position of a random node to the center of the area in question. As stated the nodes are spread uniformly over a query area with side length  $l$ . Hence the probability distribution of this random variable is represented by the following density function  $f : \mathbb{R}^2 \rightarrow [0, 1)$ :

$$f((x, y)) = \begin{cases} \frac{1}{l^2} & \text{for } |x| \leq \frac{l}{2} \text{ and } |y| \leq \frac{l}{2} \\ 0 & \text{otherwise} \end{cases}$$

Let  $\delta : \mathbb{R}^2 \rightarrow \mathbb{R}$  be the euclidean distance function  $\delta(x, y) = \sqrt{x^2 + y^2}$ . The average distance is then the expectation value of this function over all points in the area:

$$\mathbb{E}\delta(x, y) = \int_{\mathbb{R}^2} \delta(x, y) f(x, y) d(x, y) = \int_{-\frac{l}{2}}^{\frac{l}{2}} \int_{-\frac{l}{2}}^{\frac{l}{2}} \delta(x, y) f(x, y) dy dx$$

It is sufficient to examine only the positive quarter  $(x, y) > (0, 0)$  of the query region since the  $\delta$  function is symmetric in  $x$  and  $y$  and hence the average distance will be the same. Let  $l' = \frac{l}{2}$

$$\mathbb{E}\delta(x, y) = \int_0^{l'} \int_0^{l'} \delta(x, y) f(x, y) dy dx = \int_0^{l'} \int_0^{l'} \frac{\sqrt{x^2 + y^2}}{l'^2} dy dx$$

By calculating this integral we get a nice estimation which is a linear function in  $l'$ :

$$\mathbb{E}\delta(x, y) = \frac{1}{3} \left( \sqrt{2} + \operatorname{arcsinh}(1) \right) \cdot l'$$

So  $d_c(l) = \rho \cdot l$  with  $\rho = \frac{1}{6} (\sqrt{2} + \operatorname{arcsinh}(1)) \approx 0.382598$  gives the average distance that randomly distributed nodes in a square have to its center.

Observe that the average hop count a transmission to the center needs  $h(d_c(l)) = \frac{\rho}{l_r} \cdot l$  is a linear function in  $l$ .

## 3.2 Analysis of four different approaches

The following sections will describe four possible approaches of processing local aggregation queries. The expected total transmission costs per epoch can be estimated using the parameters of the previous section.

### 3.2.1 Central calculation in the sink node

The baseline for all following ideas and execution plans will be the traditional way of using a sensor network as a simple collection of sensing nodes. Every nodes does nothing more than sampling its sensors and transmitting the result to a base station. Using a geographic routing algorithm each sample needs a number hops to get there that defence on the distance between the node and the sink. The base station collects all data. Using the full information about all nodes in the region in question it is possible to calculate the set of result tuples for the query and return it to the user. Ignoring any conflicts which may be the result of sending a lot of messages on the route to the sink, the number of transmission of this procedure can be estimated as

$$c_{inSink} \approx n \cdot h(d_s),$$

since each of the  $n$  nodes has to send its result back to the sink node using geographic routing over approximately  $h(d_s)$  hops. This kind of query execution will be called the *inSink* approach.

### 3.2.2 Central calculation in the center of aggregation area

If the distance to the sink node  $d_s$  is large and if the query has a low selectivity  $s$  moving the workload of the query in the center of the examined area might reduce the total transmission cost. This approach works very similar to previous described in Section 3.2.1. The only difference is the destination of the sensor samples. The node closest to the center of the area collects all data and executes the query. Only the results fulfilling the query conditions are sent to the subscriber <sup>2</sup>. If the distribution of nodes in the area in question is nearly uniform, using geographic routing would need approximately  $n \cdot h(d_c(l))$  transmissions to send all sensor samples to the center. The central node then has to evaluate the query for each node to check whether the conditions in its neighbourhood fulfills the conditions of the query. When ever this is the case it has to send another message to the sink which is  $d(c, s)$  away. So the transmission costs for this procedure can be estimated with

$$c_{inCenter} \approx n \cdot h(d_c(l)) + n \cdot s \cdot h(d(c, s))$$

This kind of query execution will be called the *inCenter* approach.

---

<sup>2</sup>Each result tuple gets sent in a separate event message. Packing several events for the same subscriber into one bigger message might reduce the transmitting costs, but raises the risk that the message gets lost.

### 3.2.3 Distributed calculation on each node

Performing the calculation for each node separately and directly on the node is another way of processing the query. The sensor information has to be flooded with a broadcast into the neighbourhood of each node, so that each node gets the information needed to aggregate the data in its surrounding. The transmission costs  $c_{fld}(r)$  for flooding the circle around each node largely depends on the size of the aggregation radius  $r$ . Using traditional flooding each neighbour less than  $r$  away from the data source has to transmit the message once. The number of neighbours in a distance less than  $r$  can be estimated by  $\rho\pi r^2$ , if we assume a uniform node density  $\rho$ . Each of these neighbours broadcasts the message once hence the total cost is  $c_{fld}(r) = O(\rho, r^2)$ .<sup>3</sup> After flooding the sensor value, each node has to wait and collect the data from the other nodes. There has to be some time limit for this waiting period as the number of nodes in vicinity of radius  $r$  is unknown. Each node can then execute the query using the local information and decide whether a new event has to be triggered. If this is the case the messages goes directly to the subscribers using geographic routing.

$$c_{Distributed} \approx n \cdot c_{fld}(\rho, r) + n \cdot s \cdot h(d_s)$$

This kind of query execution will be called the *Distributed* approach.

### 3.2.4 Clustered Aggregation

Another possible idea lays in between the distributed and the centralized approach. Instead of sending every raw sensor value a long way to a processing node and instead of broadcasting the values too often to cover the neighborhood in question one could split the query area into cluster of nodes. This approach splits the query area using a grid of rectangles between the points  $(x_1, y_1)$  and  $(x_2, y_2)$  and running several subqueries similar to those in Section 3.2.2. As before we assume a quadratic query region. Let  $\gamma$  be the length of each square cluster. An optimal value for  $\gamma$  must be chosen by the application.

First each node has to send its data to its cluster center which is on average  $d_c(\gamma)$  away. This value decreases as the cluster-size  $\gamma$  shrinks. The limiting factor are the nodes at the border of each square. The data of nodes with a distance of less then  $r$  to a neighbour square is needed for the calculation of the local averages of the nodes in this neighbour cluster (See Figure 3.1). Hence border node has to send a second message to the center of the neighbour

<sup>3</sup>Cai, Hua and Philips have described in [CHP05] a flooding algorithm that uses the location information of the nodes to perform flooding independent of the node density by suppressing transmissions that would only reach nodes already covered by other broadcasts.

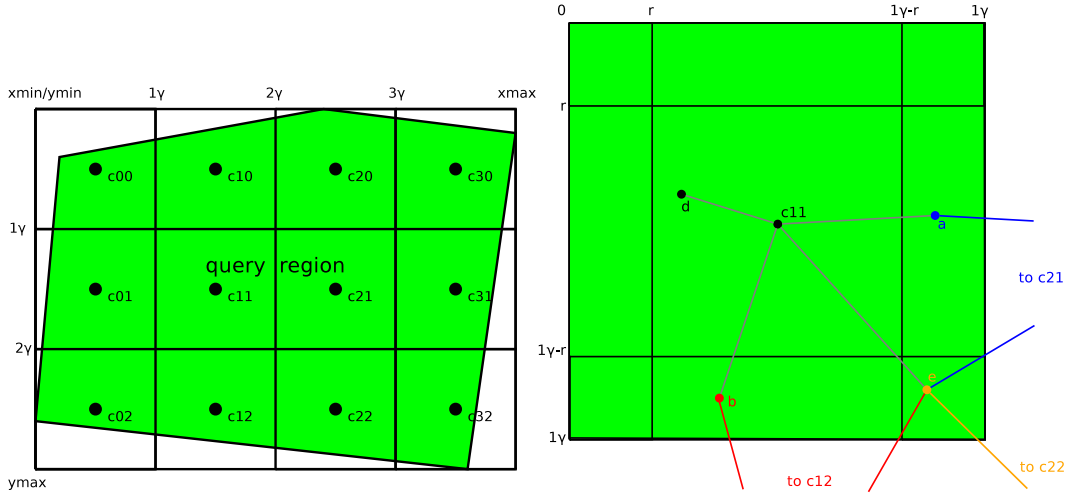


Figure 3.1: Calculation in a grid of rectangles with grid length  $\gamma$ . Nodes inside of the rectangle (d) only have to send their message to their own center. Nodes close to the borders (a,b) have to send two messages because their aggregation radius overlaps the nearby rectangle. Nodes in the corner (e) have to send four messages to all adjacent grid-centers.

cluster, which is in average approximately  $\sqrt{\left(\frac{\gamma}{2}\right)^2 + \left(\frac{\gamma}{4}\right)^2}$  away<sup>4</sup>. The nodes less than  $r$  away from the corners have to send their data value to all three adjacent cluster centers, which are in approximately  $\sqrt{\left(\frac{\gamma}{2}\right)^2 + \left(\frac{\gamma}{2}\right)^2}$  away<sup>5</sup>. As the nodes are spread uniformly over the region one can expect  $n \cdot \frac{4r^2}{\gamma^2}$  nodes to be in the corners and  $n \cdot \frac{\gamma^2 - (\gamma - 2r)^2 - 4r^2}{\gamma^2}$  nodes in the border outside of the corners.

Thus we may estimate the total cost of this approach by

$$c_{Cluster}(\gamma) \approx n(h(d_c(\gamma)) + \frac{\gamma^2 - (\gamma - 2r)^2 - 4r^2}{\gamma^2} h(\sqrt{\left(\frac{\gamma}{2}\right)^2 + \left(\frac{\gamma}{4}\right)^2}) + 3 \cdot 4 \frac{r^2}{\gamma^2} \cdot h(\sqrt{\left(\frac{\gamma}{2}\right)^2 + \left(\frac{\gamma}{2}\right)^2})) + n \cdot s \cdot h(d_s)$$

Since  $h(\cdot)$  is a homogeneous linear map this can be simplified to

$$c_{Cluster}(\gamma) \approx n \cdot h\left(\rho \cdot \gamma + \sqrt{5}r + \left(\sqrt{72} - \sqrt{20}\right) \frac{r^2}{\gamma}\right) + n \cdot s \cdot h(d_s)$$

This function has its minimum in  $\gamma = \frac{\sqrt{\sqrt{72} - \sqrt{20}}}{\sqrt{\rho}} \cdot r \approx 3.238703r$ . Using this  $\gamma$  as a grid

<sup>4</sup>A more precise approximation would include  $\frac{r}{2}$ . We can assume  $r \ll \gamma$ , since using a cluster size close to  $r$  would result in large overhead of additional transmissions. By omitting  $\frac{r}{2}$  we simplify the calculation of a good  $\gamma$ .

spacing simplifies the input of  $h$  to a proportion of  $r$ .

$$c_{Cluster}(\gamma) \approx n \cdot \phi \cdot h(r) + n \cdot s \cdot h(d_s)$$

$$\text{with } \phi = \frac{1}{3} \left( \sqrt{6 \left( \sqrt{72} - \sqrt{20} \right) \left( \sqrt{2} + \operatorname{arcsinh}(1) \right) + \sqrt{45}} \right) \approx 4.714310$$

This kind of query execution will be called the *Cluster* approach.

### 3.3 Comparison of the different approaches

Using the equations from above we get the following approximations for the number of transmissions needed per each execution cycle. Table 3.1 summarizes the variables used. Recall that  $d_s \approx d(c, s)$  in most cases.

$$c_{inSink} \approx n \cdot h(d_s)$$

$$c_{inCenter} \approx n \cdot h(d_c(l)) + n \cdot s \cdot h(d(c, s)) \quad \text{with } h(d_c(l)) = \frac{\rho}{l_r} \cdot l$$

$$c_{Distributed} \approx n \cdot c_{fld}(r) + n \cdot s \cdot h(d_s) \quad \text{with } c_{fld}(r) = O(r^2)$$

$$c_{Cluster}(\gamma) \approx n \cdot \phi \cdot h(r) + n \cdot s \cdot h(d_s) \quad \text{with } \phi \cdot h(r) = \frac{\phi}{l_r} \cdot r$$

This model ignores different congestion levels, robustness, reliability and workload distribution of the approaches.

Comparing these equations one can draw the following conclusions regarding the expected performance of each approach under different parameters.

1. If the event selectivity  $s$  is high and the sink is close to the query region simply sending the raw data to the sink should be the most effective solution.
2. If the aggregation radius is large compared to the size of the whole query region, the centered approach should work best.
3. If the aggregation radius is small compared to the size of the whole query region, the distributed approach should work best.
4. If the aggregation radius is too large causing a lot of retransmission during flooding and the whole query region is too large to aggregate in the center ( $l > \frac{\rho}{r} \approx 12.321841r$ ), the cluster approach should work best.

## 4 Implementation

All four approaches have been implemented in sensor network simulator to validate the theoretical estimations about the performances drawn in the previous section. This implementation is based upon SIDnet-SWANS, a Java-based visual tool providing "an exploratory-design environment [...] for wireless sensor networks". SIDnet itself is "built on top of the JiST-SWANS simulator which, in turn, guarantees the underlying performance and validity of the simulations"[GTS<sup>+</sup>06].

### 4.1 Preliminaries

#### 4.1.1 Jist/SWANS

The SWANS simulator runs over JiST (Java in Simulation Time) [BHvR05a], a Java based discrete event simulation engine. Instead of implementing a new simulation kernel, library or language it brings simulation semantics into the Java virtual machine. It runs on top of an unmodified Java, rewriting the Java-coded application automatically to incorporate simulation time primitives. This provides a highly efficient and transparent solution for simulating event driven systems.

Jist consists of four components. Two of them are standard Java language components. The other two are Java classes.

- A standard Java Compiler translates the simulation code into Java bytecode.
- The JiST rewriter intercepts all class load requests at runtime and modifies the bytecode to include simulation time operations. This happens only at the first load of each class.
- This modified bytecode runs on a standard java virtual machine.
- The JiST simulation kernel keeps track of simulation time, scheduling of events and synchronisation between different entities. The simulation classes can interact with

this kernel through injected and modified operations.

During the simulation the simulation time remains unchanged until the application code uses a system call like `sleep(n)`. Using these calls the developer can specify which parts of his simulation should take how long. Everything else runs virtually in zero time. The simulation kernel runs an event loop collecting and ordering the events on simulation entities. When the time of the event has come the kernel invokes the appropriate methods, only advancing the simulation time after all events have been processed.

While Jist could be modelling other simulation programs, the only existing application is SWANS (Scalable Wireless Ad hoc Network Simulator) [BHvR05b]. SWANS provides a full set of event driven components that can be composed to form a complete wireless sensor network. Each of these components are encapsulated in JiST entities having their own local state and interacting with other components over event-based interfaces.

Important components of Swans are

- a physical layer** using a field entity to transmit the signals between all radios. An hierarchical binning implementation provides an efficient signal propagation algorithm. Different path-loss models like Rayleigh and Rician fading models, free-space and two-ray (with ground reflection) have been implemented.
- a radio** can be configured with both independent and additive radio noise models and parameterized by frequency, transmission power, reception sensitivity and threshold, antenna gain, bandwidth and error model. It receives calls from the field and passes all successful received messages to the link layer. Calls from the link layer are passed to the field for propagation.
- a link layer** offers an implementation of the ad hoc part of IEEE 802.11b including DCF functionality, with retransmission, NAV and back-off functionality [CWKS97]. It receives network packets from the network layer encapsulates them in frames and passes them to the radio entity.
- a network layer** uses an IPv4 implementation which is capable of handling multiple network interfaces with separate packet queues. It gets messages from the network layer, determines the next hop using the routing component, encapsulates the message with an IP header and sends it to the link layer.
- a routing entity** that determines the next hop for each message. SWANS implements a few simple ad hoc routing protocols, none of them uses location information that we assumed to be available in our simulation. Hence we had to implement our own location

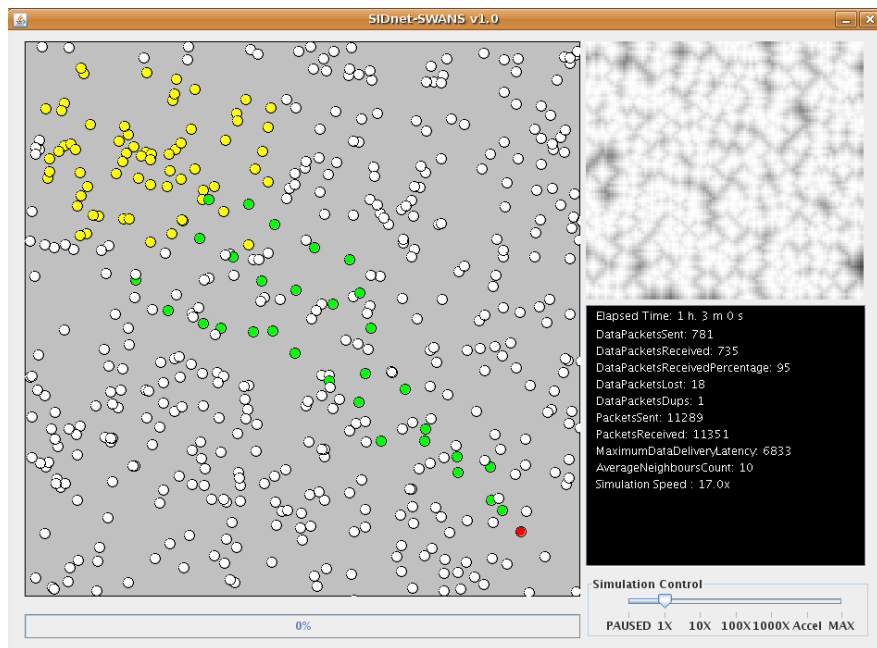


Figure 4.1: SIDnet adds a graphical user interface to the SWANS framework making it possible to follow and interact with the running simulation.

aware routing algorithm.

**a transport layer** that receives messages from the network entity and passes them to the appropriate protocol handler. SWANS implements both UDP and TCP transport protocols.

Jist and SWANS together offer a flexible and efficient platform for simulating large sensor networks. Other possible candidates that have been considered are TOSSIM [LLWC03], which offers an emulator designed for running unmodified TinyOS Code, and NS-2, an established widely known network simulator, which has been extended to support mobility and network protocols. Both of them lack the performance needed to simulate wireless networks with more than a few hundred nodes [ELVAMS<sup>+</sup>06].

### 4.1.2 SIDnet-SWANS

SIDnet[GTS<sup>+</sup>06] adds a graphical user interface wrapper to the JiST-SWANS core. It provides a flexible graphical user interface for the wireless sensor network including visual feedback of the sensor network's state and statistical monitoring of its performance. This extension takes the simulation framework provided by SWANS and extends it into a fully integrated platform for testing and developing new sensor network applications. The most

import extensions implemented by SIDnet are

**a GUI** with a sensor panel showing the placement and status of each node (See Figure 4.1).

This makes it possible to observe which nodes are currently running a query and which of them are transmitting or receiving messages. Furthermore one can define further status values, which will be displayed beside each node. Two more utility panels can be used to display further statistical information.

**a phenomenon layer** offering a dynamically changing grid of data values, providing a constantly changing environment. It is possible to place a map of the current status of this layer in the background of the sensor panel. Further details how this simulated environment works can be found in Section 5.

**a GPS module** makes it possible to expose the node location to the application layer, making it possible to use this information in a query execution system as stated in the assumptions in Section 3.

**an energy management** provides all extensions needed to monitor the energy consumption of each node. A map showing the energy status of all nodes can be placed in one of the utility panels.

**a node terminal** can be used to examine the state of a signal node and to define new queries to be injected at this node. One can define a polynomial region of the network as the query region and specifying other parameters of a query including aggregation function and where condition.

**a simple query execution example** shows how query execution in SIDnet could be implemented. This example is very simple. It sends a message containing the query object to the closest node which is inside of the query region. The nodes checks the WHERE conditions against its own sensor reading and sends a message back if the condition is matched.

**a separate batch execution programs** allows to launch several simulation runs. It can read configuration parameters from a csv-file, run a separate instance for each row and collects the results in log files.

The modular design of both SIDnet and Jist/Swans facilitates the further extensions needed to build a system executing the queries of this thesis. To do this the node terminal and the query execution example have been extended to run and process local aggregations queries as defined in Section 2.2.

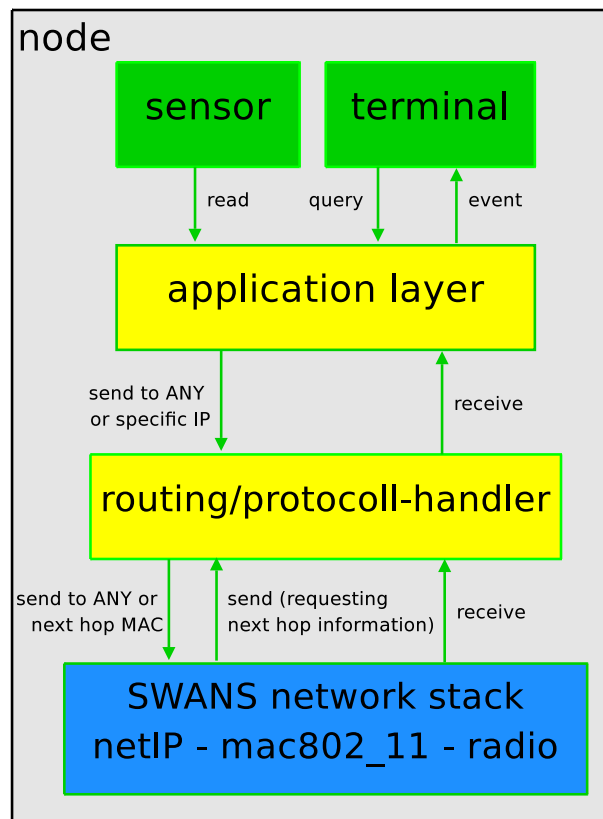


Figure 4.2: Layers inside a simulated node. The network stack (blue box) uses the original implementations from Jist/SWANS. The terminal and the sensor (green boxes) are extensions provided by SIDnet. The routing and applications layers (yellow boxes) have been implemented to provide the functionality needed to execute the local aggregation queries.

## 4.2 Architecture

Figure 4.2 shows the stack of components running on each node.

As described in Section 4.1.1, Swans provides a full set of layers, separating different levels of the sensor nodes code. At the lower levels a network stack implements IP and MAC 802.11b protocols using the radio to communicate with the field of nodes. Those components came directly from Jist/SWANS.

Each sent or received message passes the routing module, which decides whether to retransmit the message to the next hop or to hand it over to the application layer. None of the existing routing protocols in Jist/SWANS uses geographic routing. We therefore implemented a simple routing layer which will be described in Section 4.6.

The application layer implements the specific query execution system. Changing the applica-

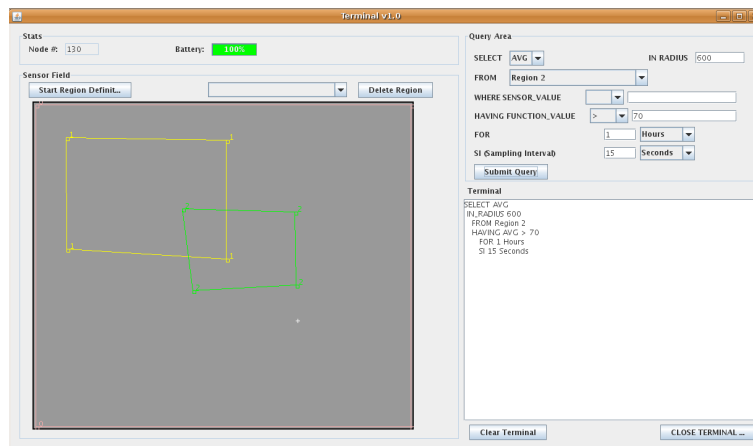


Figure 4.3: The node terminal allows to connect to a node and to specify a new query, which should be injected into the network.

tion layer allows to use the different query execution plans described in Section 3. This layer waits for an incoming query, starts reading the sensor periodically, generates data messages from the readings, collects incoming data messages and triggers event messages from the data collected.

The nodes sensor reads from a phenomena layer. SIDnet uses a transient data grid as phenomena layer.

A terminal as shown in Figure 4.3 provides the possibility to display further node specific data, like the current status and received event messages. Furthermore it provides controls to define new query regions and specify query-parameters for a new query. This allows to inject new queries during the runtime of the simulation. An implementation of such a terminal is included in SIDnet we only had to adapt it to our query parameters.

Another possibility to start a new query is to generate a new query object directly in java code. We use this to inject the same predefined query in different simulation runs. This makes it possible to get reproducible simulation results.

## 4.3 Message types

Three different message types are used to transport the data inside the network. Figure 4.4 shows a class diagram of the java classes of these message types. All have some fields in common, which are mostly used for the geographical routing. The super class `Message` stores these properties:

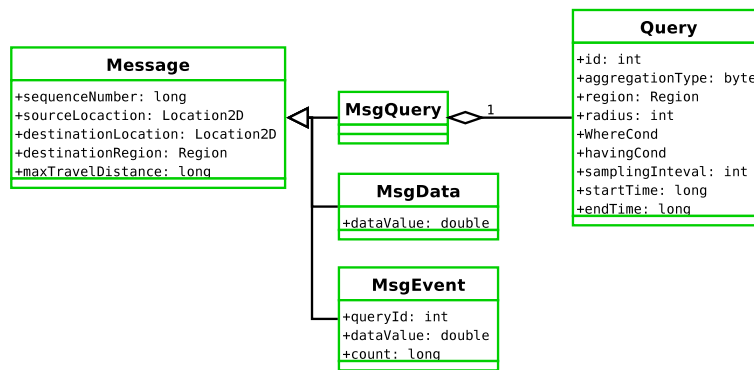


Figure 4.4: Message types used in the network

- a **sequence number** which together with the IP address of the source identifies each packet making it possible to drop messages, which have been received more than once. This is essential when flooding is used.
- the coordinates of the **source location** makes it possible to locate where the message and hence the data, event or query came from.
- the coordinates of the **destination location** defines the destination of the message used for geographical routing.
- alternatively one can define a **destination region**, that is a polynomial area, where all nodes should receive the message
- furthermore one can define a **maximal travel distance**, to restrict the range of the message to a specific circle around the source location.

The first specific message type is `MsgQuery` used to spread a new query in the networks. Each query has an ID, all the parameters mentioned in Section 2.2 and timing parameters defining how frequent a new sampling should happen. All nodes running this query will store this query object for potential use in computation of the events.

`MsgData` is the second message type, holding a raw data sample from a single node. When ever a data sample needs to be sent to another node a message of this type is transmitted.

After processing a `MsgEvent` is used to send the result. This contains the id of the query, it has an answer for, the actual value and a counter reflecting how many sensor readings have been used to produce this aggregated result.

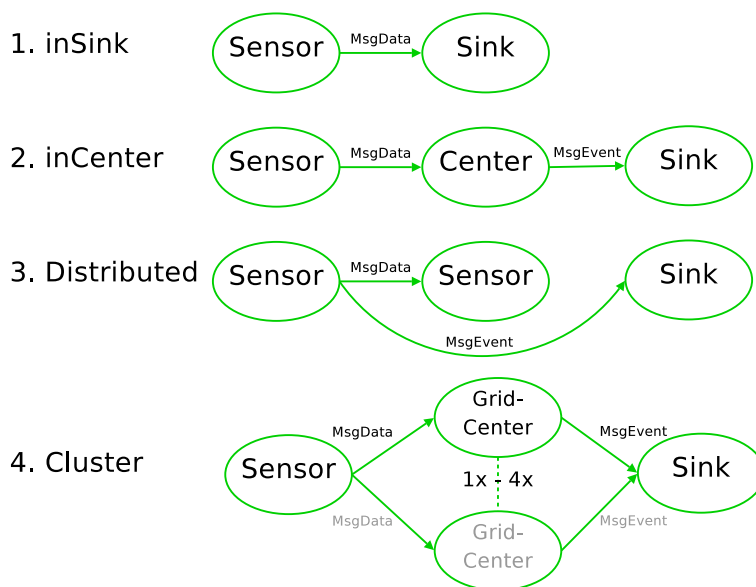


Figure 4.5: Flow of the data in the different approaches

## 4.4 Process of the query execution

The sink node generates a `MsgQuery` message from the query submitted by the user. The header of this message includes the query region as destination, so that the routing layer can use a combination of geographical routing and flooding to distribute the new query in the network (See Section 4.6).

When receiving a new query request, nodes inside of the query region start a new timer using the sampling interval of this query. Nodes wake up on the expiration of the timer and sample their sensors. Each execution plan implemented in the specific application classes has now its own way of handling the new sensor data, which differs in the destination the `MsgData` message should be sent to and when to generate new events. Figure 4.5 shows all four approaches.

The baseline approach used in the first class send all data messages directly to the sink of the query. A geographic routing can be used as the sink location is known to all nodes participating in the query. The actual result calculation happens at the sink node.

The second approach sends all data messages to the center of the query region. This center can be calculated from the coordinates of the query region. The node closest to the center coordinates of the query region performs the calculation of the results. We assume the node density is high enough so that all nodes close to the center can communicate with each other to determine the one closest. For each node location in the region this node has to calculate

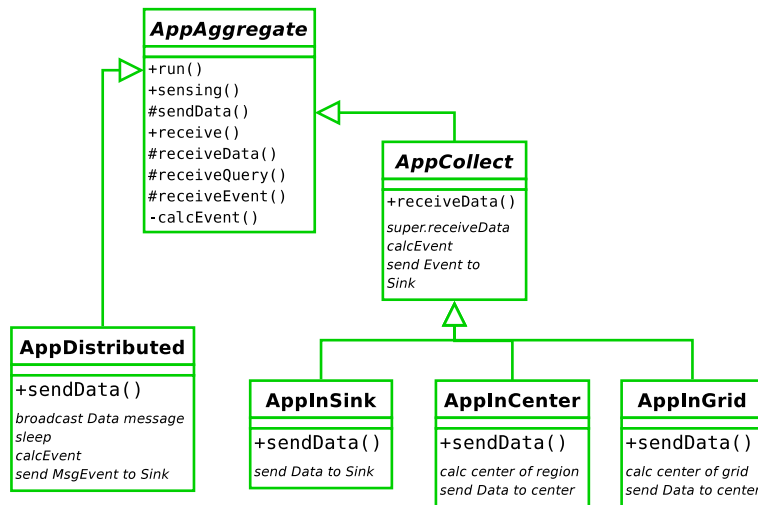


Figure 4.6: Application classes

the aggregated result and check the having clause on this result. If the result fulfills the having conditions a `MsgEvent` message is used to send to result to the sink.

The third approach is a distributed solution to the problem. Each node calculates its own aggregation result. Therefore, it needs the data from its neighbours. Each node sends a broadcast `MsgData` message to all nodes less than the aggregation radius  $r$  away. Nodes inside of the radius store the location and the value in their cache and relay the message. Nodes outside of the radius discard this `MsgData` message. After the local cache has been filled with the data values of all other nodes in the aggregation radius, the result is calculated and a `MsgEvent` message sent to the sink if the having condition is fulfilled.

The last approach works similar to the second approach, but instead of sending all data to a single node the calculation task is spread between several cluster centers. This should reduce the congestion, which would otherwise happen in large query regions. Because the data of nodes close to the borders between the clusters is needed in more than one cluster, several data messages might be sent to different cluster centers. Section 3.2.4 describes the details of this process works.

## 4.5 Application classes

Query execution happens at application level. Each of the four different ways of executing the queries in question implements a different class at the application layer. Figure 4.6 shows a class diagram of the application classes.

Most of the common application logic is concentrated at the abstract super class `AppAggregate`. This includes the `sensing` method that implements the loop to sample the sensor in each interval for all running queries. The `receive` method is the interface to the routing protocol, called every time the radio receives a new messages for the node. Depending on the message type a specific submethod is called.

The `receiveData` method stores the source location and the data value for each `MsgData` message in a local cache. Then the node sleeps for a few seconds to receive more data messages from the neighbours of the data source. This timing is critical because the node needs to wait long enough to collect all the data needed to have get a high precision, but waiting to long will result in an additional delay before the result can be reported to the user. The fill level of the data cache can be used to approximate how far the data collection has progressed. On the beginning of each epoch the number of entries from the last epoch is stored in local variable and the cache is swept to remove all old entries. The calculation of the events does not start before the fill level of the cache has reached 90% of the old value. After waiting a few more seconds for delayed messages the processing can start. Waiting for more than 90% can be problematic because some data messages might get dropped because of collisions.<sup>1</sup>

## 4.6 Routing

This set of application classes requires two different ways of sending and routing messages through the network

1. query messages are for all nodes in a specific subarea of the network. These should be sent as efficient as possible towards the region in question and spread to all nodes inside.
2. event messages are for a specific destination node at a known location and should be sent over the most efficient path directly to this destination
3. data messages may have a specific destination location and should be sent to the node as close as possible to this location
4. other data messages are broadcast messages restricted to a specific range from the source location

---

<sup>1</sup>This procedure might result in erosion, if the delay of the data messages is widespread. Waiting additional 5 seconds after the 90% threshold has been reached, reduced this risk to an acceptable level in our simulations.

Neither SIDnet nor SWANS are directly providing an implementation of routing algorithms capable of handling these requirements. As routing is not the focus of this thesis a combination of two very simple algorithms should be appropriate. The first part is a simple greedy shortest-path algorithm sending messages with a specific destination location to the neighbour node, which is as close as possible to the destination and dropping the message if no node closer can be found<sup>2</sup>. For every other situation a simple flooding algorithm is used. Figure 4.7 shows a decision tree for routing messages with different types of destinations.

---

<sup>2</sup>Dropping the messages if no direct path can be found is the most dumb way of handling this situation, more complex geographic routing algorithms exist [Kar00][Hei05], that try to reroute around void areas in the network.



## 5 Evaluation

Nodes are placed randomly over the terrain area and sample their sensor values from a random phenomenon layer.

This layer provided by SIDnet provides a virtual dynamic phenomenon, which has a continuous distribution in place and time. It starts with a 64 random values, between 0 and 100, placed in a regular 8x8 grid spanning the world. A second random grid represents the target situation after 10 minutes. During the next 10 minutes the phenomenon morphs from the start grid to the target grid. When the target grid has been reached a new target is generated using a new set of 64 random values. The nodes sample their sensor readings by interpolating between the values of the adjacent grid points. The result is a behavior similar to temperature. This layer does not create sudden changes.

The statistic layer of SIDnet allows to customize the monitored values during the simulation run. As explained previously using the radio is the most expansive operation. Therefore the number of transmitted messages including retransmissions is the most important metric to consider. Since all messages have roughly the same size this is proportional to the amount of time the nodes have to use their radios. Additionally we determined precision and recall of the event detection.

<b>Parameters</b>	<b>Setting</b>
Radio Communication Radius	41m
Bandwidth	40 kbit/s
Terrain	850m x 850m
Nodes	650
⇒ Node Density	$9 \frac{\text{nodes}}{100m \times 100m}$
⇒ avg. Neighbour Count	10

Table 5.1: System Parameters

Parameters	Setting
Aggregation Radius	40m
Query Region	600 x 600 $m^2$
Distance between Region Center and Sink	700m
Event Selectivity	15%

Table 5.2: default query parameters

## 5.1 Parameters

A range of different query and environment parameters have to be simulated to validate the conclusions from section 3.3 under different conditions. SIDnet provides an utility to run batches of simulations fed by a csv-parameter-file. We simulated over 40 different parameter sets. Each set was simulated for 60 epochs during 1 hour of simulation time. The following results are the average of at least 9 repetitions per approach using different node distributions.

For most runs a fixed set of default parameters (see Tables 5.1 and 5.2) have been used, varying only one parameter at a time. The effect of the following query parameters were examined:

- selectivity of the query - how many of the data readings in a node result in a new event. The HAVING argument of the query controls this parameter.
- size of the query region - the area of the query region. This can easily be controlled by defining a specific region in the query.
- distance to sink - how far do new event messages have to travel. This can be controlled by defining a specific region in that distance.<sup>1</sup>
- aggregation radius - the area around each node where the aggregation should happen. This is a direct argument of each query.

Even so the following two parameters are not part of the estimations in Section 3.3, testing them should provide further useful information.

- node density - how many nodes are in a specific area. This can only be controlled by changing the number of nodes on the field keeping the size of the field.
- packet loss - how robust are the algorithms against lost packets. An extra parameter

<sup>1</sup>The center of the query area and the sink node have be placed along one of the diagonals through the network using this distance parameter.

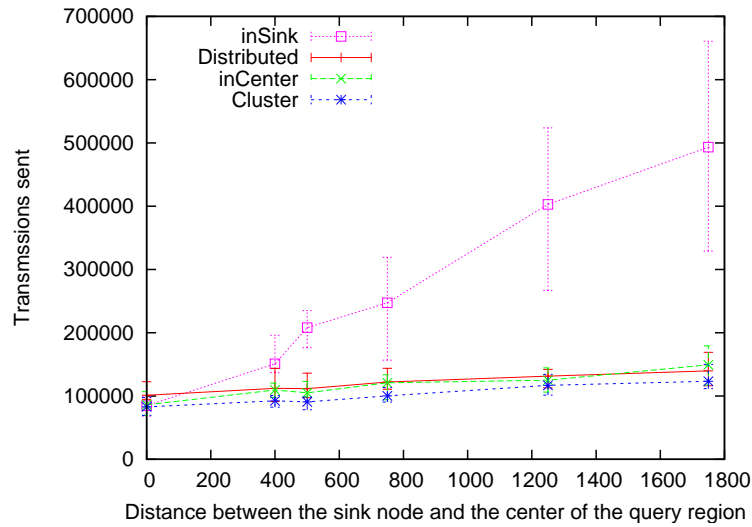


Figure 5.1: Transmission costs under increasing distances between the sink node and the center of the query area.

of the network layer provides an option of dropping a portion of all sent and received packets at random. Additional drops can always happen due to congestion.

## 5.2 Results of the Simulation

### 5.2.1 Increasing sink distance

From the model we expect a linear dependency between the sink distance and the number of transmissions, since more hops are needed to transmit the results. For all approaches besides *Central Calculation in Sink* the increase should be much lower since only the actual events are transmitted to the sink, everything else happens inside the query region and is independent from the sink distance.

Figure 5.1 shows the expected results. Since the default event selectivity of 15% is quiet low, the increase of all but *Central Calculation in Sink* is really slow. *Central Calculation in Sink* shows a high and linear dependency as long as the sink distance is large enough. During the experiments with low distances the sink was inside of the query region which is a situation not properly represented in our model.

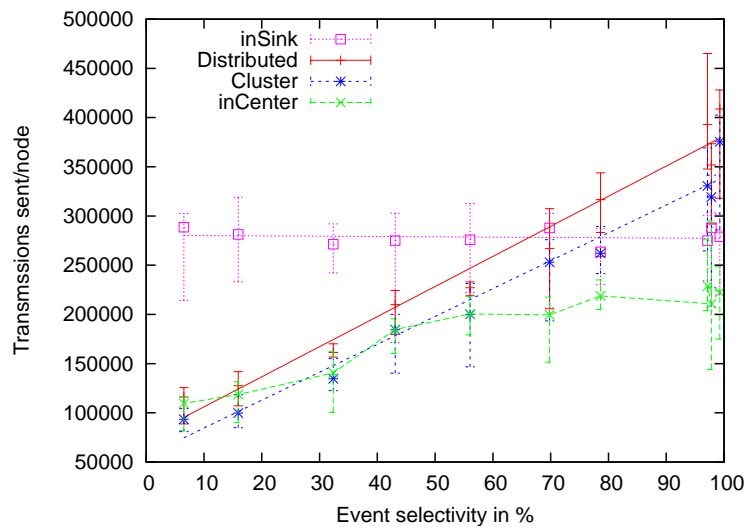


Figure 5.2: Transmission costs under an increasing event selectivity. Note, that the decrease of the inCenter approach under high selectivity comes along with break down of the recall value (See Figure 5.3).

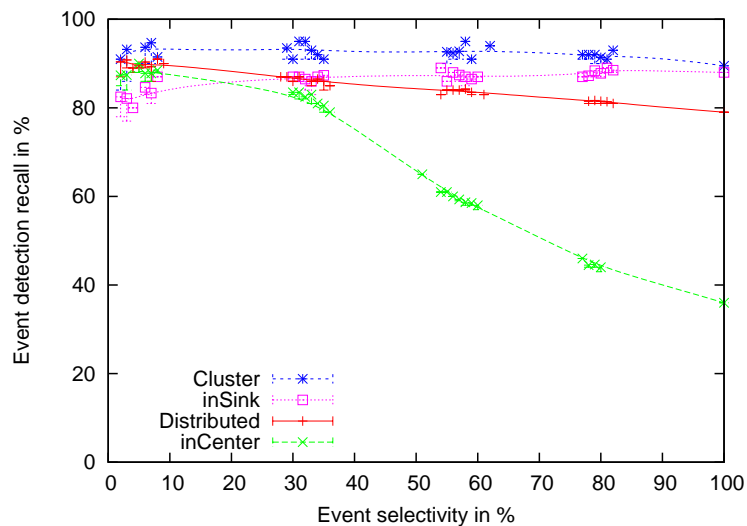


Figure 5.3: Event detecting recall under an increasing event selectivity. The inCenter approach drops a high portion of the event messages if the event selectivity raises above 30%.

## 5.2.2 Increasing Event Selectivity

From the model we expect that all methods besides *Central Calculation in Sink* will show a linear dependency between the event selectivity and the number of transmissions, since more event messages have to be transmitted to the sink as the selectivity increases. The *inSink* approach should not be effected by the number of events.

Figure 5.2 shows that this only worked out for *Distributed* and the *Clustered* approach. Further analysis using other monitored values shows that in the *inCenter* approach the number of dropped packets grows and that the recall falls under 35% as the central node has to handle a rising number of events (see Figure 5.3). This approach collects all data before calculating for individual results for all nodes. Then all new events are transmitted in individual messages from the center node to the sink node over one path, leading to congestion in this area when to many events have been triggered<sup>2</sup>. This approach could not detect much more than 200 events per epoch. This explains why the number of transmissions stops growing.

These two experiments together show that the aggregation directly in the sink performs best for high event selectivity or low sink distance. This confirms the first hypothesis from Section 3.3.

## 5.2.3 Increasing Radius

From the model we expect that the performance of the *inSink* and the *inCenter* approach is not effected by the size of the aggregation radius. The workload of the *Distributed* approach should grow quadratic with an increasing radius, while the *Cluster* approach always using an optimal cluster size should grow linearly.

Figure 5.4 shows the effects observed in the simulation. The only irregularity can be seen in the *Cluster* approach, but the explanation is simple. A raising radius results in larger clusters, and at a certain point the query region is to small to be filled with four or more clusters. Then, this approach falls back to the same behaviour as the *inCenter* because the whole query region is one cluster. As a result both approaches perform similarly.

For high radii the increase of the *Distributed* approach slows down. This again is the result of a decrease in recall. Flooding an increasing area for each nodes results in raising congestion. This shows that this approach is only useful for a small aggregation radius.

<sup>2</sup>Packing multiple events in one message should help here, but causes larger packet sizes.

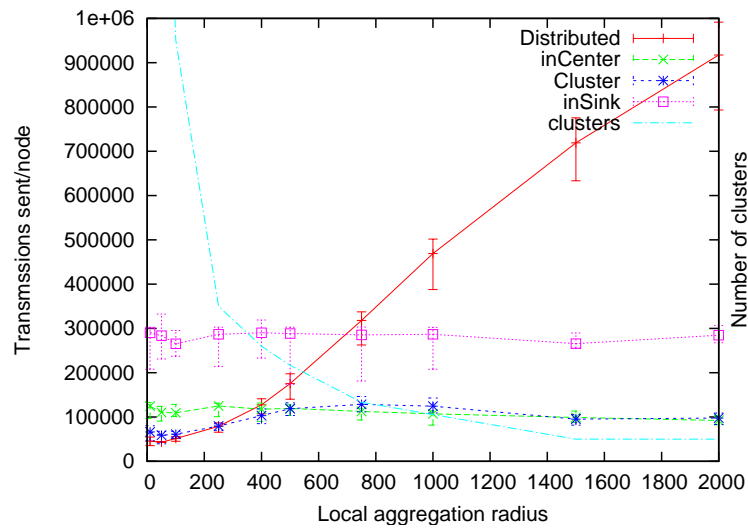


Figure 5.4: Transmission costs under increasing aggregation radii. The fifth graph shows the number of clusters used by the Cluster approach.

### 5.2.4 Increasing query region

With an increasing size of the query region more nodes are included in the query. Therefore we will use the number of transmissions per node as a performance indicator, which is independent of the varying number of nodes. From the model we expect that only the performance of the *inCenter* approach is effected by size of the query region. The workload should grow linear with an increasing size.

Figure 5.5 shows what can be observed in the simulation as the size of the query region gets larger. The *Cluster* and the *Distributed* approach show a stable performance per node, while transmission costs of the *inCenter* approach rises. The only irregularity can be seen in the *inSink* approach, but the raising number of nodes in huge query regions leads to congestion in this approach forcing more retransmission.

### 5.2.5 Nodes in query region

From the model we expect that all four approaches have a total number of transmissions that is proportional in the number of nodes in the query area. Hence the transmission costs per node should theoretical be constant.

Figure 5.6 shows that this conclusion holds for nearly all approaches. Only the total number of the transmissions of the *Distributed* approach shows a linear dependency. The reason is the performance of the simple flooding algorithm used. In the evaluation of the distributed

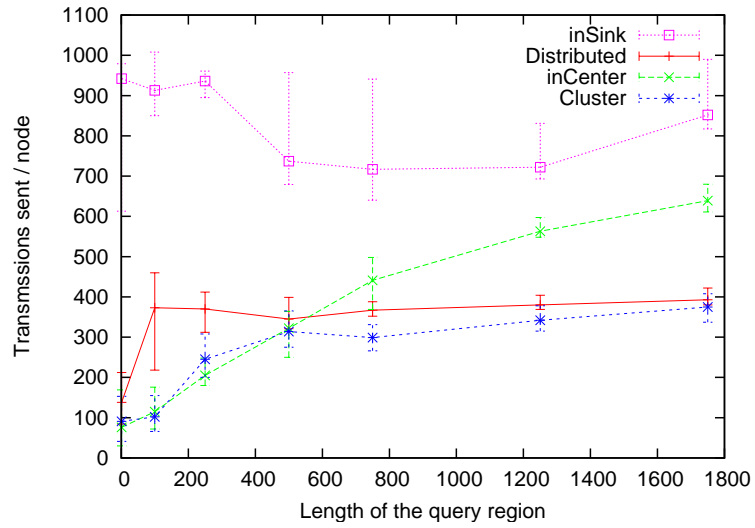


Figure 5.5: Transmission costs per node under increasing size of the query region.

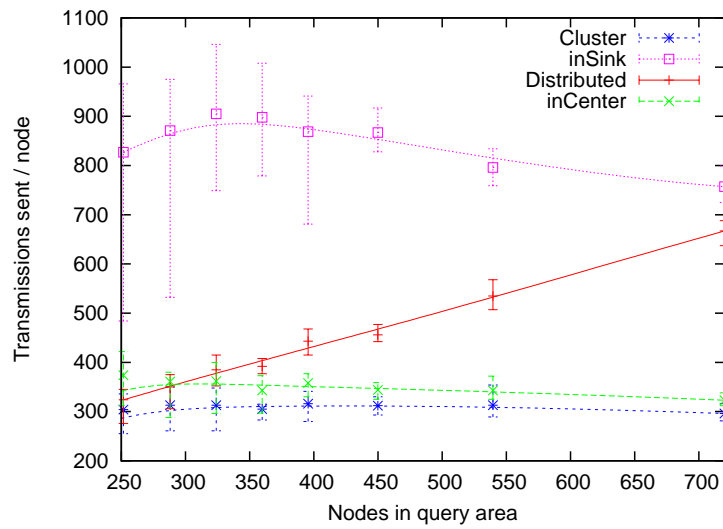


Figure 5.6: Transmission costs per node under increasing number of nodes in the query region.

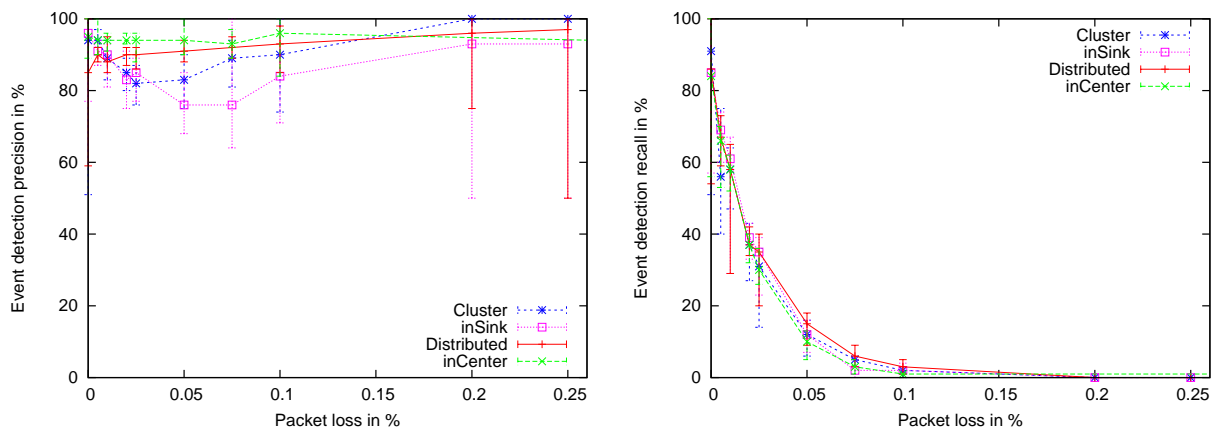


Figure 5.7: Precision and recall of the event detection under an increasing packet loss rate.

approach in Section 3.2.3 we assumed that the transmission costs of the algorithm only depends on the area to flood not on the number of nodes in this area. Cai, Hua and Philips have shown in [CHP05], that this is possible. The simple implementation used in our simulation uses simple retransmission as flooding algorithm, where each node in the flooding area retransmits the message once, even if all nodes in the transmission radius have already received it.

### 5.2.6 Performance under packet loss

Finally we tested how the four approaches perform under higher packet loss by randomly dropping some packets in the network layer. As shown in Figure 5.7 the precision stays high even under conditions with high loss rates. Only the recall is strongly effected by the increasing packet loss. The difference can be explained by the continuous nature of the phenomena layer. In a high packet loss situation less nodes are used to calculate the local average. It is unlikely that the average of these nodes differs largely from the average of all nodes in the aggregation radius. Hence only in a few cases this difference results in false positive event, so that the precision stays high. Each dropped event message results in a false negative, because the routing layer does not include retransmission of dropped messages. This explains why the recall drops very fast under high packet loss. There is no significant difference between the four approaches. Using more control messages to acknowledge received and retransmit lost messages could improve the recall in situations where a high packet loss is likely.

## 6 Conclusion

The results of the simulation validate the hypotheses drawn from the theoretical considerations (See Table 6.1).

Transmitting all of the raw data to the sink and calculating the aggregation in the sink makes only sense, if the **having**-condition filters very few aggregation results and nearly all results have to be transmitted to the sink.

A distributed approach where each node calculates its own aggregation result works good as long as the aggregation radius is small.

Collecting the raw data in the center of the query area, calculating the aggregation results and checking the **having**-condition there works as long as the query region is small, but it leads to congestion in the central area.

A clustered approach, that splits the query area in a grid of cells and calculates the result for all nodes in a cell on one of the nodes, performs best in all other situations.

The model developed in Section 3 fits to the results of the simulation and should be useful when designing an execution planer capable of running queries similar to those analysed in this thesis.

	inSink	inCenter	Distributed	Cluster
sink close to query area	o	o	o	o
larger distance to sink	-	+	+	+
lower event selectivity	-	+	+	+
higher event selectivity	+	- -	-	-
smaller aggregation radius	o	o	++	+
larger aggregation radius	o	o	- -	-
smaller query area	o	+	o	o
larger query area	-	- -	o	o
low node density	o	o	o	o
high node density	-	-	-	o

Table 6.1: Summary of the performance the different approaches under different conditions

## 6.1 Open questions

- How good works an implementation of an automatic query execution planer, that chooses the most efficient plan using the conclusions made in this thesis?
- How would more sophisticated routing algorithm influence the performance of the different execution plans? Will the assumptions in this thesis still hold? They should since the theoretical framework of section 3 did not use any details of the routing implementation.
- How would energy consumption influence the results? And how can the different execution plans handle failing nodes? The inCenter and the Cluster approach might need load balancing to spread the work load of calculating and transmitting the events over more than one node.

## List of Figures

2.1	Concept of local aggregation queries . . . . .	3
3.1	Calculation in a grid of rectangles . . . . .	10
4.1	GUI of SIDnet . . . . .	14
4.2	Layers inside a simulated node . . . . .	16
4.3	Node terminal . . . . .	17
4.4	Message types used in the network . . . . .	18
4.5	Flow of the data in the different approaches . . . . .	19
4.6	Application classes . . . . .	20
4.7	decision tree of routing algorithm . . . . .	23
5.1	Transmission costs under increasing sink distance . . . . .	26
5.2	Transmission costs under an increasing event selectivity . . . . .	27
5.3	Event detecting recall under an increasing event selectivity . . . . .	27
5.4	Transmission costs under increasing aggregation radii. . . . .	29
5.5	Transmission costs per node under increasing size of the query region. . . . .	30
5.6	Transmission costs per node under increasing number of nodes . . . . .	30
5.7	Precision and recall of the event detection under an increasing packet loss rate. . . . .	31

## Bibliography

- [BHvR05a] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist: an efficient approach to simulation using virtual machines: Research articles. *Software-Practice and Experience*, 35(6):539–576, 2005.
- [BHvR05b] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Scalable wireless ad hoc network simulation. In Jie Wu, editor, *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad hoc Wireless, and Peer-to-Peer Networks*, chapter 19, pages 297–311. CRC Press, 2005.
- [CD05] Mihaela Cardei and Ding-Zhu Du. Improving wireless sensor network lifetime through power aware organization. *Wireless Networks*, 11(3):333–340, 2005.
- [CHP05] Y. Cai, K. A. Hua, and A. Phillips. Leveraging 1-hop neighborhood knowledge for efficient flooding in wireless ad hoc networks. In *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*, pages 347–354, 2005.
- [CWKS97] B. P. Crow, I. Widjaja, L. G. Kim, and P. T. Sakai. IEEE 802.11 wireless local area networks. *Communications Magazine, IEEE*, 35(9):116–126, 1997.
- [ELVAMS<sup>+</sup>06] E. Egea-Lopez, J. Vales-Alonso, A. Martinez-Sala, P. Pavon-Mario, and J. Garcia-Haro. Simulation scalability issues in wireless sensor networks. *Communications Magazine, IEEE*, 44(7):64–73, 2006.
- [GTS<sup>+</sup>06] Oliviu Ghica, Goce Trajcevski, Peter Scheuermann, Zachary Bischoff, and Nikolay Valtchanov. Sidnet-swans: A simulator and integrated development platform for sensor network applications. Technical Report NWU-EECS-08-05, Northwestern University, USA, June 2006.
- [Hei05] Marc Heissenbüttel. *Routing and Broadcasting in Ad-Hoc Networks*. Dissertation, Universität Bern, 2005.
- [Kar00] Brad Nelson Karp. Geographic routing for wireless networks. Technical report, Harvard University, 2000.
- [KPC<sup>+</sup>07] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 254–263, New York, NY, USA, 2007. ACM Press.

- [LC02] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.
- [LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.
- [MA05] Rene Müller and Gustavo Alonso. A query processing middleware for sensor networks. Master's thesis, ETH, Zürich, CH, 2005.
- [MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM.
- [MWM06] Geoffrey Mainland, Matt Welsh, and Greg Morrisett. Flask: A language for data-driven sensor network programs. Technical Report TR-13-06, Harvard Univ., Boston, MA, USA, 2006.
- [SPMC04] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, pages 307–322, jan 2004.
- [WAJR<sup>+</sup>05] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *European Workshop on Sensor Networks*, pages 108–120, 2005.
- [YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.