Versatile Support for Efficient Neighborhood Data Sharing

Andreas Lachenmann, Pedro José Marrón, Daniel Minder, Olga Saukh, Matthias Gauger, and Kurt Rothermel

Universität Stuttgart, IPVS, Universitätsstr. 38, 70569 Stuttgart, Germany
{lachenmann, marron, minder, saukh, gauger, rothermel}@ipvs.uni-stuttgart.de

Abstract. Many applications in wireless sensor networks rely on data from neighboring nodes. However, the effort for developing efficient solutions for sharing data in the neighborhood is often substantial. Therefore, we present a general-purpose algorithm for this task that makes use of the broadcast nature of radio transmission to reduce the number of packets. We have integrated this algorithm into *TinyXXL*, a programming language extension for data exchange. This combined system offers seamless support both for data exchange among the components of a single node and for efficient neighborhood data sharing. We show that compared to existing solutions, such as Hood, our approach further reduces the work of the application developer and provides greater efficiency.

1 Introduction

As sensor networks gain momentum and applications are increasingly developed by experts in the application domain rather than experts in sensor networks, there is a growing need to simplify standard tasks while achieving the efficiency of optimized applications. To address this issue both programming abstractions and efficient general-purpose algorithms have to be considered.

In sensor network applications one such standard task is data sharing among neighboring nodes. For example, the location of neighboring nodes [1, 2] or information about their current role [3] are used by several algorithms and applications. Typically, developers create application-specific protocols for this task. This approach tends to incur significant development overhead and, for example, with a tight development budget, might often lead to inefficient solutions. Therefore, a general-purpose algorithm would not only reduce the development effort but also make neighborhood data sharing more efficient. In this paper we describe such an algorithm for neighborhood data sharing that strives to minimize the number of bytes transmitted. In addition, we use this algorithm as the basis of programming abstractions to facilitate the development of efficient applications that use data from neighboring nodes.

Although neighborhood data sharing only involves communication in a limited part of the sensor network and the size of such data is often small, the data of all nodes throughout the network adds up to considerable amounts.

Therefore, optimizing such transmissions locally on each node can result in significant improvements regarding the number of messages sent and enhance the energy efficiency of the whole network. So far, however, most work has focused on disseminating data to all nodes in the network (e.g., [4–6]) or on data-centric algorithms that transmit data to a sink node (e.g., [7]). In contrast, sharing data efficiently within the neighborhood has not been studied in sufficient detail yet. Even work dealing with programming abstractions for data sharing left the actual data transmission algorithm to be created by the application developer [8], or only provided simple ones [9].

There are two classes of data sharing algorithms: push-based and pull-based approaches [10]. With push-based approaches a node providing data sends it without having received an explicit request for it. Obviously, such approaches can lead to inefficiencies when the node's neighbors do not need this data. Especially in heterogeneous networks a node cannot necessarily infer what data its neighbors need because they may execute different code. Thus nodes might transmit unnecessary data or omit data that is actually required.

The second class of data sharing algorithms is composed of pull-based approaches. Here nodes only send data when they have received a request for it. This approach is better suited for heterogeneous networks, since each node may request the data it actually needs. The only shared assumptions are that neighbors can provide the requested data and use the same naming scheme. However, a pull-based approach can incur significant overhead for sending requests.

Therefore, we have developed *Neidas* ("**NEI**ghborhood **DA**ta **S**haring algorithm"), an efficient pull-based algorithm for neighborhood data sharing. Similar to network-wide dissemination approaches, our algorithm makes use of overhearing requests and data from neighboring nodes. It leverages the advantages of both pull-based and push-based strategies: The algorithm works well with heterogeneous networks and reduces the overhead for requests.

Typically, data is not just shared among neighboring nodes but also between the software components of a single node. We address this problem with *TinyXXL* [11], an extension to the nesC programming language [12]. *TinyXXL* simplifies cross-layer data sharing and decouples the components accessing data. Its runtime component, the *TinyStateRepository*, provides efficient access to such data. We have integrated *Neidas* into *TinyXXL* to create a comprehensive approach for data sharing among components on a single node and on neighboring nodes, which reduces the effort for the developer.

The rest of this paper is organized as follows. Section 2 describes related work. In Section 3 we present our data sharing algorithm and in Section 4 its integration into *TinyXXL*. Section 5 evaluates our approach. Finally, Section 6 gives an outlook on future work and concludes this paper.

2 Related Work

Publish/subscribe systems have been used in different domains to make data available. In sensor networks several algorithms following a publish/subscribe-like paradigm have already been proposed. Perhaps the best-known example is SPIN [4], which uses such an approach to disseminate data in the network. However, these approaches typically require explicit interaction between every two nodes publishing and subscribing to data, which is not needed by our algorithm.

Gossiping algorithms are flooding-like approaches where nodes randomly forward data packets that they have received. Trickle [5] uses a gossiping variant to efficiently distribute information about code images in the whole network. It has been integrated into Deluge [6], a code distribution algorithm, and adapted for the Drip protocol [13] to transmit queries to all nodes in the network. *Neidas* is inspired by the concepts of Trickle but deals with multiple nodes requesting potentially different data. Trickle, in contrast, can assume a single or few data sources and just one kind of data. In addition, with *Neidas* changes of data are kept local whereas Trickle disseminates them through the network.

Hood [8] is a programming abstraction that tries to ease neighborhood data exchange in sensor networks. However, it leaves important parts to be added by the application developer, e.g., data transmission policies that are responsible for sending data and requests. This allows for more flexibility than our system but increases the development effort. In addition, Hood does not strive to provide a comprehensive system for both intra-node and neighborhood data exchange.

Likewise, abstract regions [9] provide programming primitives for local communication. An abstract region is defined using radio connectivity or the location of nodes, for example. Extending the neighborhood beyond immediate neighbors within radio range is something not considered by our approach yet. Like Hood, abstract regions only include a very basic data transmission algorithm. Similarly, logical neighborhoods [14] can be used for communication within a set of nodes that are not necessarily just the nodes in radio range. However, with this system a data sharing mechanism would still have to be implemented by the application developer based on other primitives.

There are numerous algorithms and applications that make use of data obtained from their neighbors. Most of them include custom solutions for neighborhood data sharing. Prominent examples are algorithms for self-organization [3], routing [1], and medium access control [15]. By factoring out the transmission of data using *TinyXXL*, developers could focus more closely on the actual purposes of their algorithms and applications.

3 Neighborhood Data Sharing Algorithm

Neidas is a data sharing algorithm that retrieves data from all neighboring nodes in radio range and continuously transmits updates when this data changes. It is based on the observation that – even in heterogeneous networks – there are typically several nodes within radio range that are interested in the same data. Therefore, our algorithm can take advantage of polite gossiping, which was first introduced in the Trickle algorithm [5]. With this approach nodes wait a random time before sending data or a request for data from neighboring nodes. If during this time k_r neighbors send the same request, polite gossiping suppresses

```
(1) In each request round:
  Wait for the listen-only period and random interval
  For each data item needed from neighbors:
    If less than $k_r$ identical requests have been received:
      Send request
(2) In each data send round:
  For each data item requested by other nodes:
    If data item not requested in last data send rounds:
      Remove request
  Wait for listen-only period and random interval
  For each data item requested by other nodes:
    For local data and data received from neighbors:
      If less than $k_d$ copies of data with same version
      number have been received from this node:
        Send data including version number
  Double duration of data send round
(3) If new neighbors arrive:
  Reset duration of data send round to one request round
(4) Request received:
  Mark data as requested
  Increment counter for request
(5) Data received:
  If data requested and data is from node in neighborhood:
    If version number > stored version number
      Store data, source node, and version number
    Else if version number == stored version number
      Increment counter for this data
```

Fig. 1. Overview of the *Neidas* algorithm

the transmission of redundant messages. Therefore, this algorithm leverages the broadcast nature of radio transmission: If several nodes have the same request, making each node send it would be unnecessary. Similarly, *Neidas* uses the same mechanism to locally forward the data provided by neighbors.

To deal with transmission failures and dynamic topologies *Neidas* periodically resends requests and data in so-called request and data send rounds. Fig. 1 gives an overview of the basic operation of the algorithm. Our algorithm is executed: periodically in each request round (1) and data send round (2), when new nodes arrive in the neighborhood (3), and when requests (4) or data packets (5) are received. The following subsections describe the *Neidas* algorithm in more detail.

3.1 Neighborhood Management

Since *Neidas* retrieves and stores data from neighboring nodes, it needs to know which nodes are in the neighborhood. Our current implementation includes an algorithm that intercepts all *Neidas* packets to build this neighborhood table. This algorithm does not incur any message overhead because it does not send

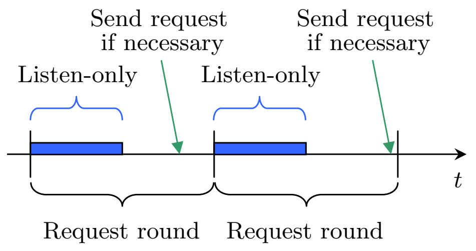

Fig. 2. Actions within request rounds

any packets itself. If it has not received any packets within a predefined interval, it removes this particular node from the table.

If there is already an algorithm available that provides the required interfaces, it can be used instead to avoid duplicate data in memory. For instance, neighborhood information can be retrieved from the *TinyStateRepository* (our cross-layer data repository [11]), SP's neighbor table [16], or accessed directly from the algorithm providing the data. To demonstrate this flexibility we implemented several such algorithms.

3.2 Sending Requests

Neidas is a pull-based algorithm, i.e., nodes send requests for data that they need. It takes advantage of overhearing messages by suppressing requests if other nodes have already sent the same one. The algorithm periodically resends requests to deal with dynamic neighborhoods and transmission failures. Therefore, it divides time into fixed-length request rounds, which are shown in Fig. 2.

Starts of rounds do not have to be synchronized on neighboring nodes. This can lead to an increased number of messages if nodes send their requests early at the beginning of a round. To avoid this problem, each round starts with a listen-only period [5] in which a node just listens for messages from its neighbors (see Fig. 2). In the rest of the round each node randomly selects a point of time at which it will send its request if by then it has not overheard at least k_r identical ones. Otherwise, it suppresses its own request in the current round.

Since neighborhoods may overlap, not necessarily all the neighbors of a node receive a request when the node overhears one. Therefore, a node might suppress its own transmission although not all of its neighbors have received the same request. This is especially a problem because there might be no other node in the neighborhood requesting that data item. Trickle can easily deal with this issue since all nodes transmit the same kind of information and because version numbers ensure that the most recent information is always sent. *Neidas*, in contrast, addresses this problem in the following way. First, the threshold k_r is set to a slightly greater value. We obtained good results with $k_r = 3$. Secondly, the random delay before sending a request ensures that not always the same nodes with the same set of neighbors send a request. Finally, following a soft-state approach with timeouts longer than a single round, nodes do not have to receive a request in every round. As shown in our simulations, in static topologies all nodes within radio range receive a request after some rounds.

If communication links are asymmetrical, i.e., node A hears node B, but B cannot receive messages from A, *Neidas* remains functional, since it does not necessarily require direct interaction between nodes to request and transmit data – given that there are other nodes with the same request. Thus *Neidas* fully makes use of the broadcast nature of radio transmission with its polite gossiping scheme.

3.3 Sending Data

Besides sending requests, *Neidas* takes care of sending the requested data itself. Nodes transmit this data in two cases. First, they send all their matching data periodically – including data received from neighboring nodes. This helps to make sure that after some retransmissions all neighbors have received it. Second, when the local data is modified, nodes send additional updates to their neighbors. This way the neighbors receive the most current data even before the next regular retransmission.

For sending data *Neidas* also takes advantage of polite gossiping: Nodes that have received data from one of their neighbors transmit it in addition to their local values. Since data is associated with a single node, only exactly the same data from the same node can suppress a transmission. In order to ensure that just the most current data is resent, the data includes a version number which is incremented whenever the data changes. To deal with version number overflows, receivers only accept data if this number is within a given range.

Since data is only relevant for the immediate neighbors and since only data originating from the same node can suppress its transmission, the polite gossiping threshold k_d for data can be smaller than k_r for requests. In addition, the version numbers define a prioritization where more recent data will not be silenced by older versions.

Nodes only accept data originating from one of the neighbors in radio range. This makes sure that data received via a third node is not disseminated throughout the network but kept within the neighborhood. Although *Neidas* currently only uses the radio range to define the neighborhood, with polite gossiping it would be easy to transmit data to differently defined groups of neighbors such as those proposed by abstract regions [9].

Data is not sent in every round in which it has been requested. The reason for this is that *Neidas* tries to reduce the number of packets. Since the data itself is often somewhat larger than a request message, it is important to minimize the number of data transmissions. Therefore, we have introduced data send rounds. All data requested in the last and current data send round is sent if k_d neighboring nodes have not already transmitted the same data. As Fig. 3 shows, a data send round is composed of one or more request rounds. The length of the data send round is doubled after each round (up to a predefined maximum

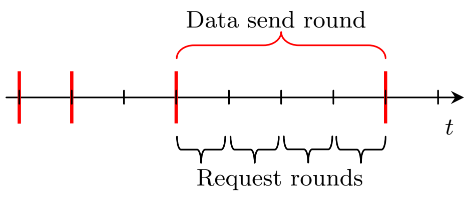

Fig. 3. Relation between request rounds and data send rounds

duration), in the figure from the length of one request round to four of them. It is reset to the length of a single request round when new nodes arrive in the neighborhood. This way these nodes receive prompt replies to their requests while greatly reducing the number of messages in static topologies. Note that changes to the length of the data send round are local to each node; they do not require any coordination among nodes. The length value reflects each node's estimate how often resends are necessary to make sure that all neighboring nodes receive a data item while keeping the rate of messages low.

We use a soft-state approach to remove requests after some time. Requested data, however, is not removed as long as the node stays within the neighborhood and there is enough memory available. So even after long data send rounds or several failed data transmissions, a node using *Neidas* still can access a previously received version of its neighbors' data from the local cache.

3.4 Further Optimizations

It is well known that radio communication consumes large amounts of energy [17]. In addition, there is also a significant MAC layer overhead associated with every packet. Therefore, reducing the number of messages is even more important than simply reducing the amount of data to be transmitted. In TinyOS and its standard MAC layer protocol [18], for instance, the MAC layer preamble, the header and the checksum included in all packets add between 17 (full duty cycle of receivers) and 2,663 bytes (low power listening with 1% duty cycle). Thus with a default data payload size of 29 bytes the overhead of sending a packet is between 58% and more than 9,000%.

Requests for neighborhood data and the data itself are expected to be comparatively small. Therefore, as an optimization *Neidas* accumulates several requests or data transmissions into a single packet. This is easily possible since *Neidas* uses small integer IDs instead of long names to identify the data and its type.

Sometimes even further optimizations are possible. Many applications and algorithms periodically send messages that do not fill the complete payload. Therefore, *Neidas* can take advantage of this free space by piggybacking its requests and data onto these messages. If the radio is operated in promiscuous mode, it does not even matter whether or not the packet is addressed to the same node as the piggybacked data, which – in our implementation – is always broadcast to all nodes in radio range. However, piggybacking is not feasible in all cases. For example, it is possible that the application does not send any data itself or that there are not enough free bytes available in the messages. Therefore, if after a time interval specified by *Neidas* the data has not been piggybacked, the piggybacking component sends a separate packet for this data.

This approach may incur some additional delays. During this time neighboring nodes might already have transmitted the same request, so that using polite gossiping it no longer has to be sent. Therefore, *Neidas* checks before actually sending the request if it is still necessary; otherwise, it cancels it.

Our implementation works with all packets sent by any TinyOS-based applications and protocols because it replaces the TinyOS components which provide the so-called active message interface immediately above the MAC layer. For both the higher-level and the MAC layer component itself piggybacking is completely transparent.

4 Programming and Runtime Support

4.1 Cross-Layer Data Exchange with *TinyXXL*

TinyXXL [11] is an extension of the nesC programming language [12] that decouples software components to ease cross-layer data exchange. With *TinyXXL* data shared among components is declared in a similar way to interfaces. Components using this data then can define dependencies without explicitly specifying the component providing it.

With automatic optimizations performed by the *TinyXXL* compiler, it is possible to develop applications that make use of cross-layer data from reusable components. For example, the compiler ensures that a single kind of data is stored only once in limited RAM and that no energy-intensive data gathering is performed redundantly. In addition, with its "virtual data items" *TinyXXL* allows the developer to create conversions and arbitrary database-like operators such as "count" and "average" to access data. This way not just the raw internal data of a component can be used by other ones but also derived data.

A pre-compiler translates *TinyXXL* source code into pure nesC code. It creates the components of the *TinyStateRepository* that stores the data at runtime. The *TinyStateRepository* offers a publish/subscribe interface with – for efficiency reasons – static subscriptions at compile-time.

4.2 Integration of Neighborhood Data Sharing

Previous versions of *TinyXXL* only allowed for data exchange among the components of a single node. To create a comprehensive system both for this kind of intra-node data exchange and for neighborhood data sharing we slightly modified *TinyXXL* to support accessing the data of neighbors and use *Neidas* in the *TinyStateRepository*. This combined system is called *TinyXXL/N*.

If a component wants to access data of its neighbors, it has to declare this property as a dependency. Then it may use the neighbors' data similar to an array

```
 1 module DataAccessM {
 2   uses interface Timer;
 3   uses xldata RoleData as RoleDataLocal;
 4   uses xldata RoleData as RoleDataN[];
 5   ...
 6   event result_t Timer.fired() {
 7     uint8_t i;
 8     for  (i=0; i<Neighbors.count; i++) {
 9       if (RoleDataLocal.role
10         == RoleDataN[Neighbors.nodes[i]].role) {
11   ...
```

Fig. 4. Accessing neighborhood data with *TinyXXL/N*

with the neighbors' node IDs. For instance, the code snippets in Fig. 4 show in line 4 how a dependency for role information [3] of neighboring nodes is declared. The brackets at the end of the line, which are not given for the dependency on the corresponding local values (line 3), denote that data is requested from neighbors and then accessed in an array-like fashion. With these declarations both local role information and that of neighboring nodes can be accessed (see lines 9 and 10 in the figure). If the data of a node is accessed which has not been received yet, a default value specified with the declaration of the data is returned (e.g., a reserved value indicating the absence of data). Since RAM is very limited on sensor nodes, *Neidas* does not store separately which nodes have already sent their data.

If data from neighboring nodes is declared to be accessed by at least one component, the *TinyXXL* compiler reserves some memory for caching this data locally. In addition, it adds calls to the *Neidas* algorithm to retrieve and continuously update the data. The compiler ensures that for each data item – even if it is requested by several components – there is only one such request sent and that the same data from a single node is stored only once in RAM. This way applications can benefit from the advantages of *Neidas* without adding the burden of implementing data exchange on the application developer. In fact, it is possible to retrieve some arbitrary data from the *TinyStateRepository*. The developer of the code running on the neighboring nodes does not have to be aware of the fact that an already existing piece of data might be needed by another node. This is an important advantage of our approach that facilitates independent development of software in heterogeneous sensor networks as well as reusability and exchangeability of components, whose data is automatically shared with neighboring nodes when necessary.

One inherent assumption of this solution is that on neighboring nodes the data is provided by a component and stored in the *TinyStateRepository*. Because of optimizations performed by the *TinyXXL* compiler, data is only gathered on a node if there is a component that needs to access it locally. Otherwise, it removes the data gathering code to reduce runtime overhead. In this case these nodes cannot answer requests for such neighborhood data. Therefore, like with manually implemented data sharing, the developer has to ensure that data needed from neighboring nodes is available.

If a node just accesses its local values and not those of its neighbors, there is almost no overhead associated with the integration of *Neidas*. In this case *Neidas* does not transmit any request messages. In addition, its RAM consumption is almost negligible: There is no need to reserve memory for local copies of neighborhood data, if the node does not use it. However, *Neidas* has to reserve two single bits for each kind of data in order to check if it has been requested by neighboring nodes within the most recent data send rounds.

Our solution offers the benefits of *TinyXXL* also for neighborhood data sharing. For example, it decouples components providing and accessing data: The component providing a piece of data that is needed by another node can be different from the one requesting data. In fact, in heterogeneous networks the component providing this data does not have to be part of the application requesting it. In addition, just like with local data it is possible to use virtual data items to transform data or perform some computations on it. Thus a neighboring node does not have to store a piece of data in order to provide it, as long as it can be converted to the target format. Furthermore, by taking advantage of the *TinyStateRepository*'s publish/subscribe mechanism the system can transmit updated data to its neighbors if it has been modified.

5 Evaluation

5.1 Experimental Setup

We have simulated *Neidas* and *TinyXXL/N* using Avrora [19], which accurately emulates the behavior of Mica2 nodes. Unless otherwise noted, each simulation scenario contains 50 nodes which are randomly placed in a 60 m $\times$ 60 m rectangular area. Since communication is only local to the neighborhood, we expect that the results are also valid for larger-size networks.

The nodes' radio model is set to a lossy model, which is based on empirical data and has a transmission range of about 15 m. The TinyOS MAC layer takes care of multiple accesses to the radio channel. The measurements shown are the average of 10 runs of 600 simulated seconds each. We have set *Neidas*'s polite gossiping thresholds k_r to 3 and k_d to 1. As described above, experiments have shown that good results can be obtained with these values. The duration of a request round has been set to 10 s for all algorithms but in long-running experiments this value can be neglected – as long as all algorithms use the same duration. Nodes are turned on randomly in the first 10 s and are not switched off before the end of the simulation. Unless otherwise noted, we have not made use of piggybacking optimizations.

5.2 Efficiency of *Neidas*

We have created straight-forward implementations of standard pull-based and push-based algorithms, which are likely to be integrated in similar form in real

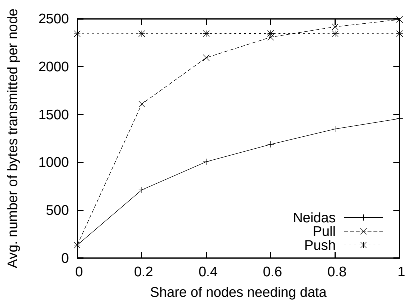

Fig. 5. Bytes transmitted, varying share of nodes requesting data

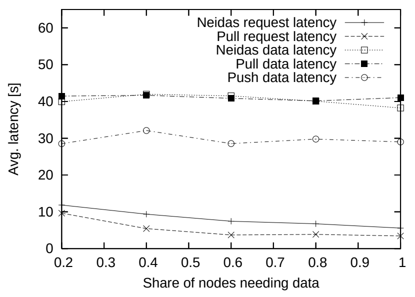

Fig. 6. Latency until requests and data have been received

applications. For a meaningful comparison, all of them use the same underlying data format and marshaling components as *Neidas*.

The pull-based algorithm periodically requests the data of neighboring nodes but does not suppress requests already heard. Similar to *Neidas* it does not service a request immediately but waits until the next round. However, it does not distinguish between request and data send rounds. The push-based algorithm periodically broadcasts its data without the need for requests. Neither of these algorithms resends data from neighboring nodes.

In our simulations all nodes provide a single data item of 10 bytes. We have varied the ratio of (randomly selected) nodes that needed this data. The only messages sent are those to request and transmit data.

Fig. 5 shows the total number of bytes transmitted by each node on average – including the packet header, preamble, etc. Since there are no big differences in the processing overhead of the three algorithms, overall energy consumption is dominated by the radio. Therefore, the energy consumed by the algorithms can be inferred from the number of bytes transmitted.

The push-based algorithm always transmits the same number of bytes because it does not distinguish between nodes that need data and those that do not. In contrast, for the pull-based algorithm the number of bytes transmitted grows with the percentage of nodes requesting data. If this percentage is greater than about 70%, the pull-based algorithm is less efficient than the push-based one because of the additional overhead for request messages. Even when all nodes request data, the overhead for these requests is relatively small. The reason for this is that the pull-based algorithm uses the efficient underlying techniques from *Neidas* to build packets. Therefore, requests are usually sent together with the data as a single message, and there is no overhead for the packet header, preamble, etc. Otherwise, the numbers of the pull-based algorithm would be up to 500% greater (not shown in the figure), because the payload is very small and thus the overhead of sending extra packets has even greater effects.

Neidas transmits much fewer bytes than these two algorithms. Depending on the number of nodes requesting data, it only sends between 30% and 62% of the

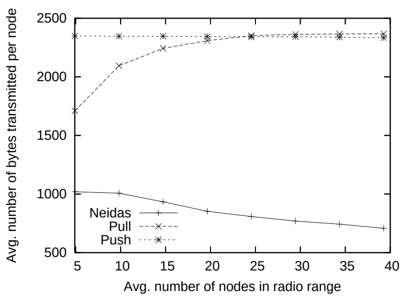

Fig. 7. Bytes transmitted varying the node density

number of bytes of the push-based algorithm and between 44% and 58% of the pull-based algorithm. Up to 20% of the savings compared to the corresponding pull-based approach are due to polite gossiping of requests. This percentage increases with higher node densities. Enlarging the length of data send rounds is responsible for the rest of the savings.

Fig. 6 compares the average latency until a node entering the neighborhood receives requests and data. The request latency of *Neidas* is up to 4 s greater than that of the pull-based algorithm because of suppressed request messages in overlapping neighborhoods. However, when comparing the latency of the data itself, the values for both algorithms are almost identical because with *Neidas* nodes are able to provide also data requested from their neighbors. The data latency of the push-based algorithm, of course, is even shorter, since with this algorithm nodes do not wait for requests before they send their data. The values for the data latency may seem comparatively high given the duration of the request rounds of 10 s. However, these numbers are average values until the data from *all* neighboring nodes has been received. Due to lossy links and collisions, some nodes have to send their data several times.

Fig. 7 shows the average number of bytes transmitted for different node densities. To get these values we have varied the total number of nodes from 25 to 200. The size of the area is kept constant and always 40% of the nodes request data from their neighbors. The figure shows that the values for the pull-based algorithm increase by about 38% with higher densities until all nodes are in the neighborhood of at least one node requesting data. For the push-based algorithm the number of bytes is constant, since each node sends its data independent of other ones. With *Neidas*, the number of bytes transmitted by each node even decreases by about 30% with higher densities although in these cases more nodes have to send their data. This is because more nodes overhear packets from their neighbors which avoids sending the same request several times.

As the results show, *Neidas* is suitable for both heterogeneous and homogeneous networks. Considering the benefits such as the small number of transmitted bytes shown in Fig. 5 and its ability to profit from high node densities (Fig. 7), for many applications *Neidas* offers a good compromise between efficiency and timely delivery of data.

5.3 Comparison with Hood

We have implemented a simple algorithm that builds a tree to route data to a sink node with both Hood and *TinyXXL/N*. In this example all nodes request their neighbors' depth in the routing tree. They then select the neighbor with the smallest depth as their parent and adjust their own depth value. Using Hood we have implemented two versions: The first one minimizes the number of messages by solely relying on Hood's auto-push policy that only broadcasts updates when the data changes. The second one is able to deal better with new nodes and transmission failures of lossy links by periodically requesting the neighbors' values in addition to the automatic updates. This version resembles more closely the properties of *Neidas* but with its forwarding of neighbor data *Neidas* is able to provide even better reliability. Depending on the properties required by the application, the solution actually implemented by the developer will probably lie somewhere within the boundaries defined by the two Hood versions. The *TinyXXL/N* implementation, however, automatically integrates *Neidas* so that the application developer does not have to deal with these low-level details.

Both the Hood versions and the *TinyXXL/N* version of the code use equivalent neighborhood management algorithms. These algorithms do not send any information themselves but use the node IDs transmitted with each request and data packet. It is an integral part of Hood's concepts that the neighborhood management algorithm has to be written by the application developer whereas in the *TinyXXL/N* version *Neidas*'s default neighborhood management algorithm is used.

Fig. 8 visualizes the total number of bytes sent by the Hood versions and the *TinyXXL/N* version for different node densities. Since the standard push-only version of Hood sends data just when it is modified, this algorithm transmits the smallest number of bytes. However, it is not able to deal with transmission failures and newly deployed nodes as it sends data only once. These two properties are fulfilled by the other two versions of the application. Therefore, these implementations offer different functionality and can hardly be compared. Thus we limit the following discussion to the *TinyXXL/N* variant and the Hood version including data pulls.

As expected, the number of bytes sent by *TinyXXL/N* decreases with high densities since it is based on *Neidas*. In contrast, the Hood version does not make use of overhearing messages and, therefore, has to transmit significantly more data if the number of nodes in radio range increases. For the highest node density the *TinyXXL/N* version sends only 24% of the number of bytes transmitted by Hood.

When compiled for Mica2 nodes, our sample application including the operating system components reserves 810 bytes of RAM in the Hood versions and just 608 bytes in the *TinyXXL/N* version (25% less). Most of *TinyXXL/N*'s savings are due to fewer variables used in the marshaling components as well as in the components storing data. With RAM sizes of just a few kilobytes such optimizations are crucial in order to be able to create complex applications.

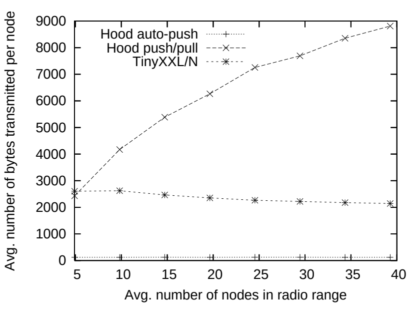

Fig. 8. Bytes transmitted compared with Hood

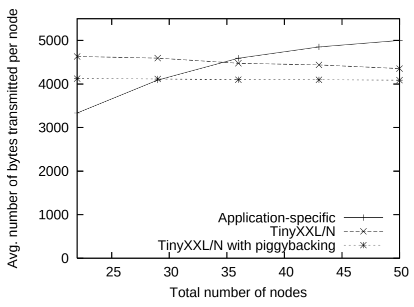

Fig. 9. Bytes transmitted for the Sense-R-Us application

From a developer's point of view creating the application with *TinyXXL/N* incurs significantly less overhead. The routing tree algorithm described above was implemented in 176 lines of code with Hood (for the version including data pulls) vs. 88 lines of code with *TinyXXL/N*. This means that the *TinyXXL/N* implementation needs 50% fewer lines of code than the Hood implementation. Most of these savings, however, are due to the fact that Hood requires the developer to implement a separate neighborhood management algorithm, which is already present in the *TinyXXL/N* solution. Although such numbers do not necessarily allow to draw conclusions about the complexity of the code, they can give a rough estimate about the effort needed by the application developer. Considering that Hood already reduces complexity compared to manual implementations [8], the overhead reductions of *TinyXXL/N* are even more significant.

5.4 Integration in Sense-R-Us

Sense-R-Us [2] is an application that uses a sensor network to provide the functionality of a smart environment where the current location of researchers in our department can be queried. In addition, Sense-R-Us is able to detect meetings using both sensory inputs and information about neighboring nodes. In this application there are stationary nodes placed in rooms and mobile ones that are carried by persons. The mobile nodes use neighborhood data from the stationary ones to localize themselves by requesting information about the location of neighboring nodes. A mobile node's location is set to the value of a neighboring node, which has been selected using the received signal strength.

We compare an implementation of Sense-R-Us that has been built using *TinyXXL/N* with the original one for which neighborhood data sharing has been implemented manually. This version uses Sense-R-Us's custom querying protocol, which tries to reduce the number of messages by intelligently selecting the nodes to be queried. However, it does not leverage broadcast communication and comes at the expense of significant development overhead.

In our experiments we simulated up to 50 nodes of which 22 ones are stationary. The remaining nodes are mobile and move using a random walk model.

Fig. 9 shows the number of bytes transmitted by the application-specific implementation of Sense-R-Us, a version using *TinyXXL/N*, and another *TinyXXL/N* version that takes advantage of the piggybacking optimization described in Section 3.4. These numbers also include packets transmitted by other components, e.g., to discover neighboring nodes. As the figure shows, for low densities with no or only few mobile nodes the performance of the *TinyXXL/N* versions is worse compared to the optimized application-specific solution. If, however, the node density is increased, the *TinyXXL/N* version can take advantage of overhearing messages and the number of bytes sent by each node decreases. For the application-specific implementation, in contrast, the number of bytes sent increases by almost 50% when adding more mobile nodes. The reason for this is that this implementation relies solely on point-to-point communication. Therefore, separate messages might have to be sent even if other nodes have similar requests. If *TinyXXL/N*'s piggybacking optimization is used, the number of bytes transmitted is reduced by between 8% and 13% compared to the other *TinyXXL/N* implementation. These savings are due to the reduced number of packets sent by this variant. Although piggybacking could also be incorporated in an application-specific solution, using *TinyXXL/N* has the advantage that it comes "for free" without requiring the developer to manually implement it.

6 Conclusions and Future Work

In this paper we have presented *Neidas*, a pull-based algorithm for neighborhood data sharing. Compared to other approaches it provides better efficiency by suppressing duplicate requests in the neighborhood. If the node density is high, the average number of bytes transmitted by each node decreases. We have integrated this algorithm with *TinyXXL*, an extension of the nesC programming language for cross-layer data sharing. The combined system, *TinyXXL/N*, is a comprehensive system for both data exchange among components and neighborhood data sharing. Using *Neidas* as its basis *TinyXXL/N* offers efficient data sharing at largely reduced development costs. For example, in heterogeneous networks the developer of a node providing data does not even have to be aware that this data might be required by another one. We are convinced that this combined system will lead to efficient applications which are developed with reduced effort.

Regarding future work we plan on making *Neidas* adaptable to the density of nodes requesting data by dynamically adjusting the threshold k_r. This will further reduce the number of requests in dense networks while increasing the share of nodes in the neighborhood that receive a request.

References

1. Karp, B., Kung, H.T.: GPSR: Greedy perimeter stateless routing for wireless networks. In: Proc. of the Conf. on Mobile Comp. and Netw. (2000) 243–254
2. Minder, D., Marrón, P.J., Lachenmann, A., Rothermel, K.: Experimental construction of a meeting model for smart office environments. In: Proc. of the Workshop on Real-World Wireless Sensor Networks, SICS Technical Report T2005:09. (2005)

3. Frank, C., Römer, K.: Algorithms for generic role assignment in wireless sensor networks. In: Proc. of the Int'l Conf. on Embedded Netw. Sensor Systems. (2005)
4. Heinzelman, W.R., Kulik, J., Balakrishnan, H.: Adaptive protocols for information dissemination in wireless sensor networks. In: Proc. of the Int'l Conf. on Mobile Computing and Networking. (1999) 174–185
5. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: Proc. of the 1st Symp. on Networked Systems Design and Implementation. (2004) 15–28
6. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems. (2004) 81–94
7. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: a scalable and robust communication paradigm for sensor networks. In: Proc. of the Int'l Conf. on Mobile Computing and Networking. (2000) 56–67
8. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proc. of the 2nd International Conference on Mobile Systems, Applications, and Services. (2004) 99–110
9. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: Proc. of the Symp. on Network Systems Design and Impl. (2004) 29–42
10. Franklin, M., Zdonik, S.: A framework for scalable dissemination-based systems. In: Proc. of the 12th Conf. on Object-Oriented Programming, Systems, Languages, and Applications. (1997) 94–105
11. Lachenmann, A., Marrón, P.J., Minder, D., Gauger, M., Saukh, O., Rothermel, K.: TinyXXL: Language and runtime support for cross-layer interactions. In: Proc. of the Conf. on Sensor, Mesh and Ad Hoc Comm. and Networks. (2006) 178–187
12. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proc. of the Conf. on Programming Language Design and Implementation. (2003) 1–11
13. Tolle, G., Culler, D.: Design of an application-cooperative management system for wireless sensor networks. In: Proc. of the Second European Workshop on Wireless Sensor Networks. (2005) 121–132
14. Mottola, L., Picco, G.P.: Logical neighborhoods: A programming abstraction for wireless sensor networks. In: Proc. of the Int'l Conf. on Distributed Computing in Sensor Systems. (2006) 150–168
15. Ye, W., Heidemann, J., Estrin, D.: An energy-efficient MAC protocol for wireless sensor networks. In: Proc. of IEEE INFOCOM 2002. (2002) 1567–1576
16. Polastre, J., Hui, J., Levis, P., Yhao, J., Culler, D., Shenker, S., Stoica, I.: A unifying link abstraction for wireless sensor networks. In: Proc. of the 3rd Int'l Conf. on Embedded Networked Sensor Systems. (2005)
17. Shnayder, V., Hempstead, M., Chen, B.r., Allen, G.W., Welsh, M.: Simulating the power consumption of large-scale sensor network applications. In: Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems. (2004) 188–200
18. Polastre, J., Hill, J., Culler, D.: Versatile low power media access for wireless sensor networks. In: Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems. (2004) 95–107
19. Titzer, B., Lee, D., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: Proc. of the Conf. on Information Proc. in Sensor Netw. (2005)