Self-Organizing Access-Centric Storage Optimization in Smart Sensor Networks

Carsten Grenz, Uwe Jänen, Jonas Winizuk, and Jörg Hähner

Organic Computing, University of Augsburg, Augsburg, Germany
carsten.grenz@informatik.uni-augsburg.de

Abstract. Sensor networks are getting much more complex these days. The mixture of various low-cost sensors together with increasing computational power enables for whole new systems running a lot of different analysis and control algorithms concurrently. It is impossible to anticipate their composition and data flows a priori. Although the actual data flows are hardly predictable during design-time, we present a lightweight and self-organizing approach on how shared data stores are used to optimize the storage allocation of data during run-time. While mostly using the existing traffic to disseminate routing information, we show that our distributed algorithm significantly reduces query latencies by placing data according to the access-centric storage paradigm.

Keywords: Distributed Algorithm, In-Network Storage, Routing.

1 Introduction

Classical sensor networks often consist of many homogeneous nodes which are targeted on specific goals like collecting observations from their physical environment and regularly report them to a sink, or occasionally report specific events. Much research has been done on all layers of the protocol stack to optimize these systems for various configurations and environments. However, these systems were mostly optimized during design-time by domain specialists to efficiently and effectively solve their specific tasks.

The ongoing improvements in the fields of sensor hardware and networking capabilities lead to whole new compositions of sensors and their integration into multi-purpose sensor networks. One example are Smart Cameras (SCs) which incorporate a visual sensor, a capable computation unit, and a (wireless) network interface [14]. SCs are able to perform elaborated vision algorithms right on the sensor itself to extract high-level information like the detection and tracking of persons, or identifying objects and situations [6]. One example for a smart surveillance system has been presented in [4] which integrates algorithms from different application domains into a self-organizing ad hoc network. The image sensing and processing is handled by *vision algorithms* running on the SCs. Other algorithms exchange messages over the ad hoc network to track people across the entire camera network. Distributed *control protocols* take care of the reconfiguration and alignment of the cameras' field-of-views to establish the best

N. Mitton et al. (Eds.): AdHocNets 2015, LNICST 155, pp. 161–172, 2015.
DOI: 10.1007/978-3-319-25067-0_13

recording conditions. Elaborated data processing and fusion algorithms use data from SCs to perform pattern detection and 3D reconstruction [2]. User terminals which are arbitrarily distributed in the surveillance region are also part of the network. All these algorithms exchange data using an ad hoc network.

A main difference to classical sensor networks is the way the data is accessed: While some applications issue periodic queries that may cover whole geographic regions, the number of algorithms and applications that perform random accesses on data in the network is on a rise. This is especially the case for systems whose users want to access the information during run-time. That is why, the latency of queries to data stored in the network becomes a major design goal to be responsive and perform in real-time. Due to the concurrent execution of an increasing number of distributed algorithms, it is impossible to anticipate the actual data flows during design-time.

Our contribution is a routing protocol for a self-organizing storage allocation algorithm which migrates data in ad hoc networks and routes requests to the data accordingly. The primary goal of the data placement heuristic is to minimize the average route lengths for queries taking recent accesses into account. The routing protocol is embedded in our storage middleware implementing migration policies to realize the access-centric storage paradigm [5].

2 System Architecture

The nodes in the network are connected through an ad hoc capable wireless LAN network interface with a transmission range which is small compared to the region the sensors are deployed in. Each node is a smart sensor whose software architecture is depicted in Fig. 1. Our *storage middleware* is located between the *application* and the *network layer*. It is generally applicable to sensors as well as other devices since it makes as few assumptions about the other layers as possible. The *application layer* encapsulates any sensing or control algorithm that stores and retrieves georeference-based data. These algorithms interact with connected sensors or process data from the data stores. This layer also contains applications for user interaction. Each data item gets annotated with a geographic position which represents the *key* of this data towards the storage layer. For evaluation purposes we model certain kinds of applications' behavior in *producer* and *consumer* modules (see Sec. 4). The *storage middleware* contains our self-organizing storage reconfiguration algorithms and offers the interface of a distributed hash table (DHT) for each data store to the application layer. The message types for the interaction between application and storage layer are:

Put(Key k, Value v)	Request to store data item v with position k
Get(Key k)	Request to retrieve date item from position k
Result(Key k, Value v)	Returns the data item of a $Get(k)$ request

It contains a *local hash table* which is responsible for certain coordinate ranges which change during run-time. Each data item is accessed using its *key* which represents a geographic coordinate. The *local Lookup table* translates key coordinates to their current storage locations. The *dynamic reconfiguration module*

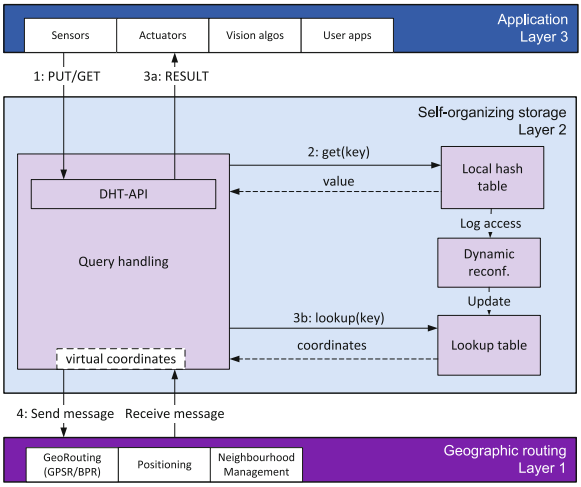


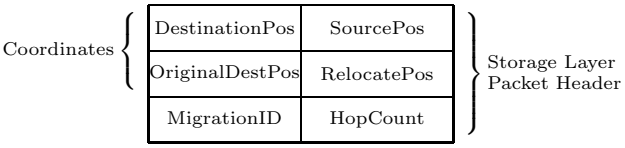
Fig. 1. Node architecture

logs accesses to data a node is responsible for. This log is analyzed and data is migrated when a more suitable storage position has been determined. All nodes becoming aware of new storage locations add a coordinate tuple to their Lookup table which reflects the new positions to optimize the routing process.

The *routing layer* contains a geographic routing protocol, i.e., Greedy Perimeter Stateless Routing (GPSR) [7], which operates on the nodes' positions during packet forwarding. Therefore, each node has to acquire its position, e.g., by using GPS. The nodes exchange beacons to announce their positions and the RNG algorithm is used to planarize the resulting connectivity graph.

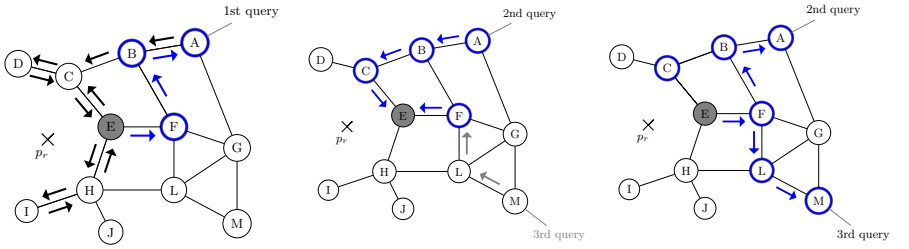
3 Algorithm

Our Distributed Access-Centric Storage Algorithm (D-ACS) optimizes the storage positions of data items to minimize the latency caused by queries. This is achieved by migrating data and caching information about known data migrations in distributed Lookup tables. Upon receiving a DHT request ((1) in Fig. 1), the storage layer checks the node's responsibility and migration information. At first, it checks its local hash table (2). If the node is responsible for the data item, the hash table will return the valid data item and it can be handed over to the application layer using a *result message* (3a). Otherwise, it will encapsulate the query into a *storage layer packet* which contains the following header fields:



The *DestinationPosition* and the *OriginalDestinationPosition* are set to the key position. Then, it checks its Lookup table (3b) by running the packet update algorithm (see Alg. 1 on page 167). The algorithm ensures that depending on the actuality of the data either the local Lookup table is updated with the packet's header information or vice versa. The actuality of the data is represented by the *migration ID* which is incremented for each migration of the data. Finally, the packet is handed down to the routing layer (4) and is sent towards the current *destination position* (initially its location-centric home node). During packet forwarding, each intermediate node also checks its Lookup table for newer information. This way, the actual *destination* of a query packet may change several times before reaching its destination (the current data node) while the *original destination* always stays the same.

Consider the network in Fig. 2a with node A accessing an data item σ. To access data, an application generates a DHT request, e.g., a *get request*. It contains a *key* which represents the coordinates of the request, i.e., $key = p_r = (x_r, y_r)$, which is the *original destination position*. Since all Lookup tables are empty, initially, the storage layer packet is routed towards position p_r without being rerouted (black arrows). Most often p_r lies between nodes. We make use of the face routing mechanism of the geographic routing protocol to find the node which has the smallest distance to p_r (the *location-centric home node*). This is achieved by exploring the nodes around p_r (the *home perimeter*) [7]. This causes the traversal of the path $E \to C \to D \to C \to E \to H \to I$ determining node E as location-centric home node. This node is the *current data node (CDN)* and adds an entry to the Lookup table which resolves p_r to its own position p_E. Afterwards, it logs the access and sends the response back to the querying node. Therefore, it sets the *destination pos* to the request's *source pos*, but keeps the *original destination pos* at the *key*. Since this is the first access, the *migration ID* is set to 1 and the *relocate pos* is left empty. The response packet is handed down to the routing layer which delivers it to the originating Node A. Fig. 2a shows that the result packet does not necessarily take the same route as the request



(a) 1st query from A to p_r. (b) Subsequent queries. Node C gets updated. (c) More information spread.

Fig. 2. Initial query routing. With no routing information available geographic routing is used (black arrows). The response leads to dissemination of the data's current location (blue arrows and circles, 2a). Subsequent packets get updated (2b) from the Lookup table and lead to more information spread (2c).

(blue arrows). The nodes on the packet's path add a tuple to their Lookup table containing the key, the CDN's position, and the *migration ID* 1. The knowledge of this information is represented by the blue ring around nodes. The color of the arrow represents the version of migration information the packet carries.

Figure 2b shows how subsequent accesses are updated from the nodes' Lookup tables and the information is spread even further (Fig. 2c). While the query from node A is already updated in step (3b) in Fig. 1, the query issued by node M also reaches the CDN directly since it gets updated by node F. A query is updated by setting the *destination pos* to the one stored in the Lookup table while keeping the *original destination pos* at the key.

3.1 Migration Module

To optimize the storage allocation during run-time, we introduced a *dynamic reconfiguration module* in [5]. This module is part of every node's storage layer and is responsible to periodically analyze the access structure to the data a node stores and identify potential migration pressure representing suboptimal data placement. Therefore, each node keeps short backlogs of accesses to data items. After a key has been accessed ten times, the *dynamic reconfiguration module* optimizes the storage location and performs data migration if the reallocation leads to a decreased query latency, thus, ensuring the access-centric storage paradigm. The *access model* (Acc_{σ_i}) contains a list of the origins of queries in combination with their access frequencies. For each entry $acc \in Acc_{\sigma_i}$ the originating coordinate is accessible via $acc.x$ and $acc.y$, the number of accesses via $acc.n$, and the access type (i.e., the number of message exchanges necessary for a query) via the relativity value $acc.rel$. The optimal coordinates are calculated using the following formula for x (the other coordinates are calculated accordingly):

$$\text{Optimal.x}(\sigma) = \frac{\sum_{acc\ in Acc_{\sigma_i}} acc.x \cdot acc.n \cdot acc.rel}{\sum_{acc\ in Acc_{\sigma_i}} acc.n \cdot acc.rel}$$

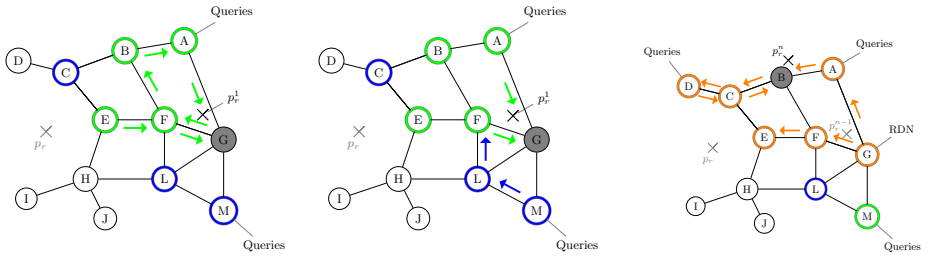
This formula calculates the optimal position which would minimize the access latency in hops. Since nodes may be arbitrarily distributed, this method does not guarantee optimal results. However, the evaluation shows that this heuristic produces good results while only imposing very small overhead. For a thorough description of the migration decision process and an evaluation of parameters like the access threshold, the reader is kindly referred to [5].

3.2 Data Migration

The calculated optimal storage position of a data item is denoted by p_r^j with $j \in \mathbb{N}$ indicating the j-th migration. To migrate data, a CDN sends out two types of messages: a *migration message* and a *relocate message*. The *migration message* contains the data and is sent towards the new reference position p_r^j incrementing the *migration ID* to $j + 1$ denoting the newer information. The message gets delivered to the node next to p_r^j. If this node denies the migration,

e.g., due to small remaining memory or battery power, the CDN would have to retry. If the node accepts the migration, it stores the data and becomes the new CDN. Subsequently, a *relocate message* is sent to the first (location-centric) home node at p_r (or p_r^0). This ensures that the *home node* can be used as a fallback in cases when queries do not reach the CDN because of missing routing information. By examining these messages, intermediate nodes also learn about the recent migration which increases the information spread.

Figure 3 shows two examples of migrations. The CDN E performs a migration towards p_r^1, which is located next to Node G (see Fig. 3a) and determined by the message traversing the perimeter around p_r^1. For the first migration, no *relocate message* is necessary, since the recent data node (RDN) is itself the original home node. The following queries shown in Fig. 3b are already nearly optimal since the query from Node M is rerouted by Node F. Considering another migration by Node G to p_r^n, both, the migration and the relocate message are sent and Node B becomes the new CDN. Because of the information spread, the access paths from nodes A, M, and I will be optimal although nodes M and I have no or outdated information.



(a) The migration message determines node G as new CDN.

(b) Resulting query paths. The query from M gets updated by node F.

(c) Node G sends a migration message to B and a relocate message to E.

Fig. 3. Data migrations. Node E migrates the data to p_r^1. In (3c) node G performs n-th migration of the data towards p_r^n. New migration information is represented by green and orange circles, respectively.

4 Evaluation

We used extensive simulations to show our algorithm's performance, explore its parameter space, and compare it to location-centric storage (LCS) [3]. Our experiments were taken out in the discrete-event simulator *OMNeT++* [17] together with the *MiXiM* extension which offers models to simulate the characteristics of wireless network interfaces. All nodes are equipped with an IEEE 802.11b/g wireless LAN interface in ad hoc mode which has a transmission range of 160 m and run an implementation of the *greedy perimeter stateless routing (GPSR)* protocol [7]. The requirement for GPSR to operate on a planar graph has been met by implementing the *Relative Neighborhood Graph (RNG)* planarization algorithm proposed by the authors. To focus the evaluation on our algorithms,

Algorithm 1. Update Packet

```
 1: procedure onUpdatePacket(StorageLayerPacket msg)
 2:     localInfo = retrieve entry from Lookup table for msg.originalDestPos
 3:     if No local info found then
 4:         Add information to Information Vector
 5:         return
 6:     if localInfo.MID[1] < msg.MID then            ▷ Packet's information is newer
 7:         Update local information from the message header
 8:     if msg is of type RELOCATE or RESULT then     ▷ Do not update these
 9:         return
10:     if localInfo.MID[1] > msg.MID then            ▷ Local information is newer
11:         Update packet
12:         if msg is of type PUT then     ▷ compensate for (potential) packet loss
13:             Resend RELOCATE to home node
14:     return
```

[1] localInfo.MID is the stored MigrationID

each run has a startup phase of 60s in which the nodes exchange beacon packets and perform the graph planarization (RNG).

Our simulation setups are summarized in Table 1. During startup, the nodes are placed randomly in a simulated area with the size of $1,200 \times 1,200\,\mathrm{m}^2$ and the applications are setup. Nodes run different application models depending on the experiment. To represent the behavior of smart sensors, the *producing application* stores data that is associated with the nodes' surrounding space, i.e., it issues *put* request to keys in its geographic vicinity. The *consuming application* represents any kind of algorithm querying sensor data by issuing *get* requests, e.g., to analyze the data and subsequently store its results, or to process the data and display the results to a user. Each *consumer* randomly chooses five geographic regions of interest upon startup. During run-time, it periodically queries equally distributed geolocations in these regions (see Table 1). The query period is varied randomly by $\pm$ 1 second to avoid synchronization effects. To research our algorithms, the number of producing and consuming nodes is varied on startup as well as dynamically during run-time. The following graphs show the average route lengths (quantified by the number of hops) for *put* and *get* queries, respectively.

Table 1. Simulation setup

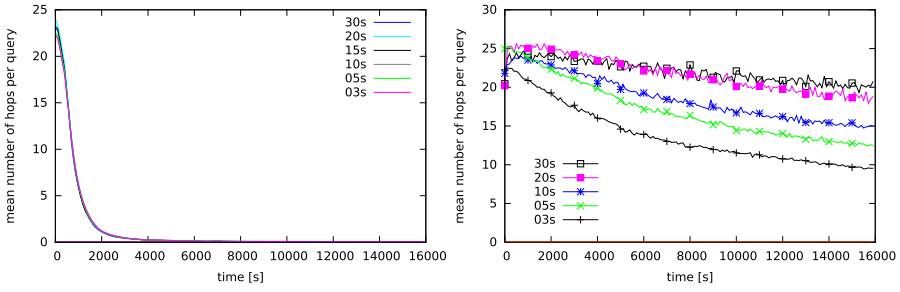
Parameter	Static access patterns	Dynamic patterns	One data item
Run-time	16,000 s	16,000 s	1,000 s
Repetitions	8	10	9
Number of nodes	100	70 + 5 every 2,000 s	100
Put period	10	10	*n/a*
Get period	{3s,5s,10s,15s,20s,30s}	{3s,5s,7s,10s,15s,20s}	15s
Put region size	3x3	3x3	*n/a*
Get region size	8x8	8x8	1

Static Access Patterns

The first set of experiments shows the general performance of our algorithm. Therefore, the parameters of the producing and consuming applications are chosen upon startup and are not changed throughout the simulation run. All 100 nodes create *put* requests with a period of 10 seconds. 30 of these nodes also run consuming applications with fixed *get* request periods which are varied from 3 s to 30 s in the different setups (see Table 1).

Fig. 4a shows the resulting average route lengths for the *put* requests. Obviously, the impact of these types of requests on the network load is very low. This is due to the locality-preserving nature of the modeled sensors, which repeatedly generate sensor events in their direct vicinity. The initial long routes of above 20 hops are due to the home-perimeter runs around previously not addressed positions. Up to 2,000 seconds, one can observe a huge reduction to nearly zero which is mainly caused by nodes storing information about their neighboring nodes in their Lookup tables. Initially, the location-centric storage paradigm leads to the storage of data items either on a producing node itself or a nearby neighbor. Due to migrations of data items towards their optimal position performed by our algorithm, the mean hop count only increases slightly what is invisible in the graph since it is averaged out by the many local storage operations.

The right graph (Fig. 4b) shows the route lengths of the *get* requests and shows a huge reduction in mean hop counts. After the migration threshold of 10 accesses is met, data gets migrated towards its estimated optimal position. The mean hop count is reduced continuously due to the *get* accesses which are equally distributed in the chosen regions of interest. A higher access frequency leads to faster optimizations and also to better results since data gets moved closer to the querying nodes: With a request period of 30s (top line), the mean hop count is reduced by 18.5%, while a request period of 3s (bottom line) leads to a reduction of 57, 8%.

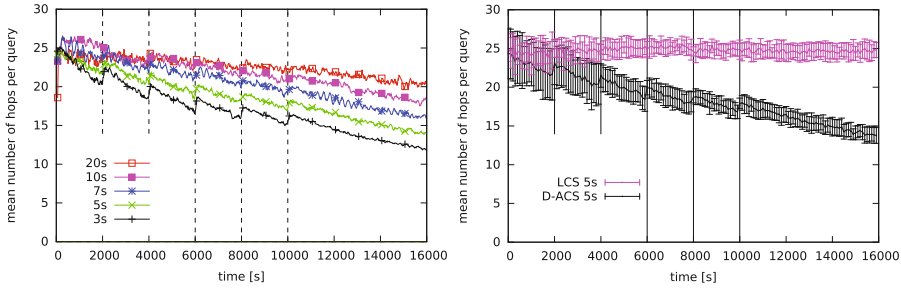


(a) Put requests - Avg. route lengths (b) Get requests - Avg. route lengths

Fig. 4. Static access patterns with different periods. The data from 100 randomly placed sensors is accessed by 30 nodes running consuming applications each with 5 regions of interest. The graphs show the optimization of queries' route lengths over time. A higher request frequency leads to better results (3s, bottom line).

Dynamic Access Patterns and Comparison with LCS
These experiments show the adaptability and robustness of our algorithm towards changing access patterns. Initially, only 70 nodes produce and store data. After startup, the consuming applications are activated in groups of five nodes every 2,000s until 30 consumers are running at $t = 10,000s$ (marked by dashed lines in Fig. 5a). The first 2,000 seconds in Fig. 5a resemble the prior measurements in Fig. 4b. Each introduction of new access patterns of the five joining nodes leads to an increase in the average route lengths. Our algorithm quickly reacts with data reallocations and optimizes the route lengths again.

Fig. 5b compares the runs with a *get* period of 5 s with location-centric storage (LCS). While the initial storage allocation is similar, our algorithm specifically optimizes the positions of accessed data leading to a huge decrease in access latency by 44%. This significantly decreases network traffic. Furthermore, our algorithm not only shortens the route lengths but also minimizes detours of packages over time which is shown by the standard deviation of the mean route lengths in Fig. 5b. While each change in the applications' behavior leads to a short rise in the standard deviation, it is obvious that the routes stabilize again over time leading to a much lower deviation.



(a) Get requests - Avg. route lengths (b) Get requests - Std. deviation

Fig. 5. Dynamic access patterns with varying rates. Five consumers are added every 2,000 s (vertical lines). Fig. 5a shows how our algorithm copes very well with changing access patterns by quick reallocations. Fig. 5b adds the standard deviation to the results with a period of 5s and the results of location-centric storage (LCS) as comparison.

One Data Item
This scenario analyses the optimization potential of our algorithm when only exactly one data item is queried by a varied number of nodes. After startup, only a fixed number of nodes (1 to 30) query the same position. Fig. 6 shows that only one querying application leads to a migration onto the node itself which results in an average route length of zero (and resembles local storage). With 2 queriers the data gets migrated between the nodes which leads to an average route length of 8.5. With an increasing number of consumers the optimization potential for our algorithm naturally decreases because the optimal position lies in between these nodes.

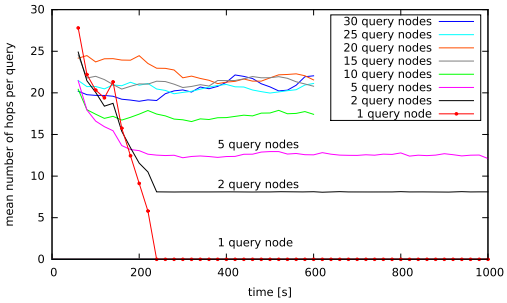


Fig. 6. Different numbers of consumers accessing one data item. The optimization potential depends on the number of queriers. With only one query node it itself becomes the storage node. With more query nodes added the optimization potential decreases.

5 Related Work

Our work originates from the idea of using coordinate translations for fast and transparent access to distributed storage which offers the interface of a distributed hash table (DHT) like Chord [16] and CAN (content-addressable network) [11]. But these approaches form an abstract overlay network which may impose significant detours in the underlying network which is unfeasible for sensor networks with their limited capabilities.

Considering in-network storage algorithms for sensor networks, the authors of [15] proposed a widely adopted data-centric storage (DCS) paradigm. In DCS, a data item is stored on a node which is chosen based on the event's name. In contrast to our work, the authors consider the sensor network to only be queried using one or more fixed access points. Moreover, their approach needs a naming scheme which has to be announced a priori to all nodes before the storage of data can take place. We overcome this drawback with our dynamic reallocation algorithm. The authors propose *Geographic Hash Tables* (GHT) that supports data-centric storage [15,12]. GHT offers a DHT-like interface for key-value-pairs. Data's position is determined by passing the key through a hash function which returns geographic coordinates. Then, GHT uses a geographic routing protocol to route the query to the node that is geographically closest to this position. Their application of a hash function leads to an equal distribution of data on the network nodes, but they do not consider the imposed load on the network. Moreover, the authors only consider queries from a fixed sink. This significantly differs from our application scenarios where we optimize the storage allocation during run-time. The authors of ZGHT [8] try to improve the storage allocation of GHT in nonuniform dense networks by introducing zones, which are responsible for similar amounts of replicated data. By adjusting the size of a zone, they achieve load balancing in terms of storage usage on the nodes. However, the ZGHT algorithm computes all zones centrally with the knowledge of all nodes' positions and floods the calculated hash function into the network. In contrast, our approach offers a fully decentralized storage allocation optimiza-

tion without a single-point-of-failure. A similar approach is Q-NiGHT [1] which uses nonuniform hash functions to meet the challenge of unequally distributed sensor nodes. Moreover, their algorithm creates a fixed number of replicas of a data item. Another load-balancing approach is presented in [10] proposing a temporally rotating hash function. By changing the storage location in predefined ways during run-time, the system ensures a balanced resource utilization of the nodes. Furthermore, they introduce so-called "potential-based location selection" where nodes report their "internal contribution potential" considering their remaining storage space and energy level [9]. Periodically, potential information is distributed to gain the potential of the cells. This information is then centrally used by a sink to modify the hash-function to point to the more potential nodes. Our approach, in contrast, focuses on the distributed optimization of query routes based on current access patterns to data as primary optimization objective and scales very well. The authors of [13] perform load-balancing by analytically creating a hash function a priori based on expected probability density functions of queries. The online version of their algorithm which optimizes the storage assignments during run-time by collecting load statistics at a central server which then floods the new assignments in the network. This approach becomes unfeasible in large or busy networks.

A fundamental paradigm for in-network storage is location-centric storage (LCS). It combines the expected locality of accesses with the DCS paradigm. Thus, a storage node is determined by evaluating its proximity to a geometric reference location specified by the spatial data which can also be determined using GPSR [3]. We compare LCS to our algorithm.

6 Conclusion and Future Work

We presented a distributed self-organizing algorithm for access-centric storage in smart sensor networks whose primary design goal is the online optimization of in-network storage allocation of georeferenced data. Our novel approach is very lightweight w.r.t. message overhead and achieves a huge decrease in route lengths of up to 57%. This way, the query latency as well as the overall network load is decreased significantly. To function, our algorithm mainly requires some additional storage capacity, which is getting increasingly cheap even for smaller devices, to maintain its local state.

The speed and amount of migrations is a design parameter of our algorithm. Depending on the application domain, different migration policies may lead to much faster optimizations. In this respect, we are going to extend the results of our work in [5]. In the future, we want to investigate the theoretical bounds of access-centric storage w.r.t. its optimization potential compared to the required overhead. Moreover, we want to research different extensions of our algorithm which cover an explicit dissemination of information with CDNs advertising their responsibilities. Another field of our research are suitable replication strategies for access-centric storage.

References

1. Albano, M., Chessa, S., Nidito, F., Pelagatti, S.: Q-NiGHT: adding QoS to data centric storage in non-uniform sensor networks. In: International Conference on Mobile Data Management, pp. 166–173 (2007)
2. D'Angelo, D., Grenz, C., Kuntzsch, C., Bogen, M.: CamInSens - An intelligent in-situ security system for public spaces. In: International Conference on Security and Management (SAM), Las Vegas, Nevada, pp. 60–66 (2012)
3. Dudkowski, D.: Fundamental Storage Mechanisms for Location-based Services in Mobile Ad-hoc Networks. PhD thesis, Universität Stuttgart (2009)
4. Grenz, C., Jänen, U., Hähner, J., Kuntzsch, C., Menze, M., D'Angelo, D., Bogen, M., Monari, E.: CamInSens - Demonstration of a distributed smart camera system for in-situ threat detection. In: Proc. of Int. Conf. on Distributed Smart Cameras (ICDSC) (2012)
5. Grenz, C., Tomforde, S., Hähner, J.: Access-centric in-network storage optimization in distributed sensing networks. In: Human Behavior Understanding in Networked Sensing, pp. 19–44. Springer International Publishing (2014)
6. Hoffmann, M., Wittke, M., Hähner, J., Müller-Schloer, C.: Spatial partitioning in self-organizing smart camera systems. IEEE Journal on Selected Topics in Signal Processing 2(4), 480–492 (2008)
7. Karp, B., Kung, H.T.: Greedy perimeter stateless routing for wireless networks. In: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), Boston, MA, pp. 243–254 (2000)
8. Kumar, B.: ZGHT- A Zonal Hash-Table for Data-Centric Storage. TAMU Comp.Sci, College Station, TX 77840
9. Le, T.N., Xuan, D., Yu, W.: An adaptive zone-based storage architecture for wireless sensor networks. In: IEEE Global Telecommunications Conference (2005)
10. Le, T.N., Yu, W., Bai, X., Xuan, D.: A dynamic geographic hash table for data-centric storage in sensor networks. In: IEEE Wireless Communications and Networking Conference (WCNC), pp. 2168–2174 (2006)
11. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 161–172 (2001)
12. Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., Shenker, S.: GHT: A geographic hash table for data-centric storage. In: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications, New York, NY, USA, pp. 78–87 (2002)
13. Renda, M.E., Resta, G., Santi, P.: Load Balancing Hashing in Geographic Hash Tables. IEEE Trans. on Parallel Distributed Systems 23(8), 1508–1519 (2012)
14. Rinner, B., Wolf, W.: An Introduction to Distributed Smart Cameras. Proceedings of the IEEE 96(10), 1565–1575 (2008)
15. Shenker, S., Ratnasamy, S., Karp, B., Govindan, R., Estrin, D.: Data-centric storage in sensornets. SIGCOMM Computer Commun. 33(1), 137–142 (2003)
16. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Commun., New York, NY, pp. 149–160 (2001)
17. Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, Simutools (2008)