Chapter 2
Access-Centric In-Network Storage Optimization in Distributed Sensing Networks

Carsten Grenz, Sven Tomforde and Jörg Hähner

Abstract Distributed sensing networks are getting increasingly complex these days. The main reason are the changing demands of the users and application scenarios, which require multipurpose systems. Enabled by continuously improving computational and storage capacities of sensors, this development leads to an increasing number of different algorithms which run concurrently in a sensing network. Thereby, they enable sensor-actuator platforms to perform various kinds of analysis and actions in parallel. Within such a sensor network a variety of algorithms is performed simultaneously. When developing distributed vision and control algorithms, developers focus mainly on the consecutive processing stages. Such a process typically begins with perceiving raw sensor data and terminates with delivering high-level event data to responsible entities. Thereby, different stages may be performed at varying locations within the underlying network. Although the researchers may apply custom optimizations to their data flows, these are highly specific. During design time, it is impossible to anticipate each system environment or predict their algorithms' possible interactions and synergies with other data flows. We propose a generic storage architecture which separates algorithms from data storage and retrieval. By making use of the fact that most data in sensing networks refers to geographic areas, our architecture takes care of the data flow and its online optimization throughout the network at runtime. By decoupling the processing stages from the data flow, we allow for self-organizing meta-level optimizations of data placement in the network. Moreover, this approach even makes inter-algorithmic optimizations possible,

C. Grenz (✉) · S. Tomforde · J. Hähner
Organic Computing, Universität Augsburg, Augsburg, Germany
e-mail: carsten.grenz@informatik.uni-augsburg.de

S. Tomforde
e-mail: sven.tomforde@informatik.uni-augsburg.de

J. Hähner
e-mail: joerg.haehner@informatik.uni-augsburg.de

P. Spagnolo et al. (eds.), *Human Behavior Understanding in Networked Sensing*,
DOI 10.1007/978-3-319-10807-0_2

if different algorithms process similar data within their step-wise processing logic. With the introduction of the access-centric storage paradigm, we prove to reduce network load and query latency at the same time at runtime.

2.1 Introduction

The ongoing advances in the field of smart sensing applications lead to new challenges concerning requirements of communication middlewares and protocols for such systems. Especially, the trend toward integration of different kinds of sensors, on the same hardware platform as well as on different hardware platforms, can be observed. This diversification process leads to new data-aggregation and -fusion algorithms and, therefore, to many kinds of generated data and metadata.

Classical sensing networks mostly focused on specific tasks they were designed for. These networks are called *mono-tasked* systems. One representative of this class are wireless sensor networks assembled of low-powered Mica2 motes which take temperature and humidity measurements, and transport them to a base station. Typically, these systems were optimized by domain specialists to efficiently and effectively solve their specific task.

The advances in the field of computational power and network capacities led to much more elaborated kinds of sensing networks and applications, e.g., the integration of visual sensors into sensing nodes led to the development of smart camera networks. The main goal of such a *smart* camera is the extraction of high-level information from pictures taken by the camera. While the development started with vision algorithms applied to single cameras, nowadays, researchers apply various data-fusion algorithms to combine data of different cameras using networking capabilities [22]. This way, they generate contextual metadata to gain person-specific information or create situational descriptions like movement patterns. One step in the further research of smart cameras was the application of movement capabilities, e.g., pan-tilt-zoom actuators or moving robots, which enable the cameras to change their field of views (see [11, 26]). Using communication protocols from the peer-to-peer computing domain, researchers built distributed and self-organizing vision and control algorithms, which increased the efficiency and effectiveness of those systems.

Nowadays, the trend goes toward multipurpose systems integrating various sensor platforms and their specific algorithms [2]. This rise of heterogeneous systems leads to sensor data being fed into the systems from different kinds of sensors. Simultaneously, an increasing number of algorithms works concurrently to fuse and aggregate many kinds of descriptive metadata. Originally, the algorithms working in these heterogeneous systems were not developed with such an integration in mind. Moreover, they may even come from very different domains and it is not feasible for any developer to anticipate every application scenario of their algorithms during design-time, since an engineer naturally focuses on the main application scenario.

Due to this different focus, today's sensing networks are occupied with uncountable flows of data and control packets of various applications which leads to new challenges on different layers of the networks' protocol stacks. This challenge is arising in all complex smart sensing networks [6, 19].

A main concern of developers of distributed sensing algorithms is the emitted network load and, from an application's point of view, the latency of queries as well as the responsiveness of their applications during runtime. Latency is especially an issue, if the system serves some security-related function or if human operators are part of the interaction loop. Although developers of distributed algorithms put in huge efforts to optimize their own algorithms' data flows, the effects of interaction resulting from various distributed algorithms working on different subsets of the available data is not predictable. Moreover, the algorithms have their own, and often different, ways of coping with situations like node churn and other disturbances. However, there mostly exists no notion of how the algorithms may share their data. Therefore, each algorithm's optimizations take only its own expected data flows into account. Since the set of running algorithms accessing specific data items cannot be determined, this may lead to unnecessary high network traffic and contention. In the end, it is not possible for a developer to anticipate any future application scenario.

As a conclusion, it is unfeasible and often impossible for any developer to anticipate their algorithms' application context, which consists of the system's hardware-parameters, the concurrently running other algorithms, and the behavior of people interacting with the system and triggering events.

In this chapter, we meet these challenges presenting algorithms for a distributed online storage optimization build into a storage framework. On the one hand, our approach is general enough to be easily applied to various application scenarios of smart sensor systems. On the other hand, it is specialized enough to allow dynamic optimizations during runtime. Therefore, our framework offers a generalized interface for georeferenced data items.

After characterizing our system model, we present the storage-latency search problem. Afterwards, we introduce a generic storage layer for smart sensor nodes. We investigate the architecture's design space by discussing relevant parameters and their dependencies, and analyze the interactions with the underlying routing model. In the evaluation, we present appropriate metrics and show how our framework optimizes the storage allocation during runtime proving the benefits of our approach.

2.1.1 Case Study: CamInSens

One example for a smart sensing system which uses multistage processing is the CamInSens system [6, 10], which runs different high-level algorithms at the same time relying on each others data. It integrates PC-based smart cameras with pan-tilt-zoom functionality with Mica2 motes [3]. The system's main goal is the online extraction of persons' trajectories and their annotation with adjacent event data. The trajectories are acquired by single-camera multi-person trackers processing images

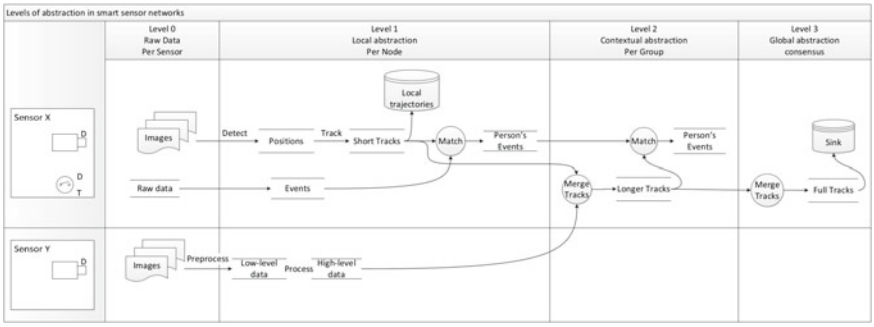

Fig. 2.1 Example data flow of a multistaged algorithm

next to the source—meaning as closely to the visual sensor as possible [18]. Distributed multi-camera trackers, which use the output of trackers on different cameras, build overall trajectories [19]. The smart cameras base their viewshed-planning on current and predicted trajectories [12, 13]. In addition, another set of algorithms annotates these trajectories with other events based on data from nearby sensors, i.e., recognized events in the surroundings of the Mica2 motes.

To illustrate this statement, Fig. 2.1 depicts an exemplary data flow for two sensors. The figure shows the data flows of two smart cameras with the upper one being accompanied by an additional mote sensor. From left to right, one can see the different levels of abstractions that match different communication primitives used. Level 0 represents the local recording of raw sensor data. At level 1, this data is processed locally in order to extract first metadata, i.e., people's positions and events like glass breakage. This information is merged into short annotated tracks of people. On the next level, geographically adjacent sensors merge their findings into contextual abstractions. This behavior is applied gradually until a global abstraction is reached. Depending on the user's needs, he may access the merged global data at the control room, or the contextual data stores of the different sensors. The latter case is especially useful, if the user moves through the region under observation and accesses the system with an ad hoc connected device.

This example shows how people from different domains cooperate to develop high-level algorithms for such systems. A modern sensor middleware should offer a universal interface to integrate such heterogeneous entities. Moreover, it shows a dimension of the incalculability of access patterns on data during design time of system components, since the number of nodes, their distribution in the environment and their interconnection is not set at design-time.

We are going to research this new area of interacting algorithms and present a generalized self-organizing storage framework for smart sensing networks. Our work enables developers and maintainers of self-organizing networks to concentrate on their algorithmic developments while our distributed storage framework manages and optimizes storage locations adaptively based on the current demand.

2.2 Related Work

Our work originates from the idea of using a virtual coordinate overlay for fast and transparent access to distributed resources which offers the interface of a distributed hash table (DHT). Well-known examples of distributed hash tables for peer-to-peer systems like the internet are Chord [24] and content-addressable network (CAN) [20]. These approaches form an overlay network whose topology is independent from the underlying network. Although they offer a general approach for the distributed implementation of hash tables, they may impose significant detours in the underlying network. This leads to unwanted network load and is unfeasible for sensor networks with their limited capabilities (e.g., bandwidth, energy consumption).

Considering in-network storage algorithms for sensor networks, the authors of [23] proposed a widely adopted data-centric storage (DCS) paradigm. It makes use of the position of the nodes and the data to store. Therefore, it uses a geographic routing protocol, i.e., *Greedy Perimeter Stateless Routing* (GPSR) [14], to determine actual storage positions. Depending on the position that is assigned to data, different storage patterns can be reached. One example is location-centric storage, which ensures that data is stored near its origin [7].

The authors of the *Geographic Hash Table* (GHT) combined the idea of determining storage positions using hash functions and the routing on the underlay network using GPSR [21]. Their application of a hash function leads to an equal distribution of data on the network nodes (given an uniform distribution of the nodes in space), but they do not consider the imposed load on the network or individual nodes caused by their distribution. Moreover, the authors of GHT consider a fixed number of queries per second independently of the number of participating nodes. Consequently, they argue that the overall usage of nodes decreases with an increasing number of nodes. This differs from application scenarios such as smart camera networks, where many nodes are considered to produce and consume data. We also make use of a virtual coordinate space in our Lookup table while routing on real geocoordinates using geographic routing protocols. But our approach uses a mutable virtual coordinate space to optimize the storage allocation w.r.t. latencies.

Since GHT does not take the actual network topology into account, different efforts have been made to distribute the workload equally upon suited nodes: The authors of ZGHT [16] try to improve the storage behavior by introducing zones, which are responsible for similar amounts of replicated data. The goal when choosing the size of a zone is to achieve a load balancing in terms of contained number of nodes. In contrast to our approach, the ZGHT algorithm computes all zones centrally with the knowledge of all nodes' positions. A similiar approach is Q-NiGHT [4] which uses nonuniform hash functions to meet the challenge of unequally distributed sensor nodes. Another load-balancing approach is presented in [17] proposing a temporally rotating hash function which changes the storage location in a predefined way during runtime. Our approach, in contrast, focuses on the actual access patterns to data as primary optimization objective and adapts the storage locations to a dynamically changing system in a self-organizing way during runtime.

The authors of [1] also consider the hop count as a criterion for data placement. As part of their approach, a multicast-tree is generated. Nodes have to proactively join and leave this tree. As opposed to our approach, there may only be one producer of a data item and they solve the problem to optimally place successive replications of the measured data items. Furthermore, they assume a fixed update rate of each data item by its producer. Our approach, in contrast, is significantly more flexible as it supports any number of producers and consumers with changing access frequencies. Moreover, our algorithm has a smaller message overhead since no tree management is necessary.

The authors of [5] argue for a layered architecture representing standardized interfaces and semantics. Within our concept, we utilize their notion and representation, while presenting an approach with a varying focus.

2.3 System Model

Our system model extends a previously developed system model of smart camera networks, see [9, 11]. The smart cameras and security personnel's devices are distributed in a convex area of surveillance. All participating entities in the network, i.e., smart cameras and the security personells' hardware, are connected using IEEE 802.11b/g wireless LAN. The smart cameras are equipped with processing and networking capabilities. Therefore, all cameras are able to communicate with each other using appropriate routing protocols.

As outlined above, all participating devices may acquire, aggregate, and store data at the same time. Therefore, we introduced the notion of *producers* and *consumers* in smart sensing networks [8]. The common case of these roles is the following: The sensors, i.e., cameras in the system, are the producers of data. The security personnel consumes this data using their workstations. These workstations may be central control rooms as well as moving mobile entities, such as tablets or smartphones, which are part of the wireless network. Furthermore, different algorithms on the sensors may be *consumers* of each other's data (see Fig. 2.2a, b).

Since all data in a smart camera network refers to geolocations, the data model consists of two parts: the *geolocation of an event* and the *event* itself. The *event* represents any event and may originate from an extraction from a visual algorithm, e.g., a person's position in space, or even an aggregated vector of data, e.g., a person's trajectory. Since the actual data is not of interest to our algorithm, we adapt the notion of a distributed key-value store or distributed hash table (DHT). All anticipated data items are relatively small compared to the storage of images, so they can easily reside and migrate throughout the network.

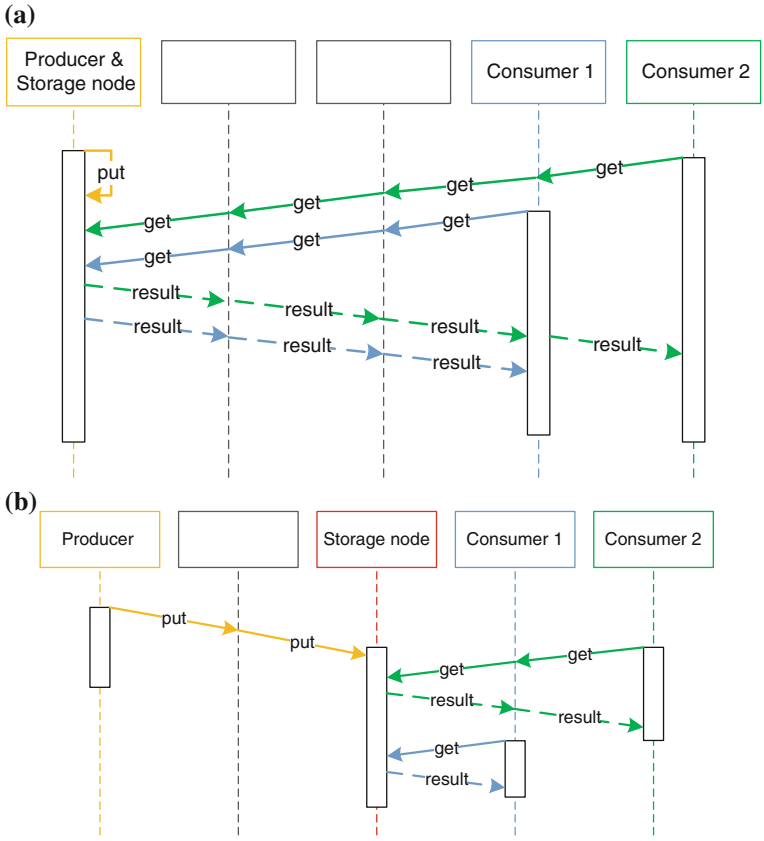

Fig. 2.2 **a** Initial query latency. The figure depicts a producer (*left, yellow*) and two consumers (*blue* and *green, right*). Access is realized using *put*, *get*, and *result* messages. The latency expressed in terms of hops in the network, i.e., the number of messages sent, sums up to 14 in this example. **b** Optimized query latency. After data migration took place, the data is now allocated to the storage node in the middle (*red*). Considering the current access pattern this leads to an optimized hop count of 8. Thus, the overall network load is reduced significantly

2.4 Problem Definition

In order to demonstrate the applicability of our framework to different classes of sensor networks, we present a formal representation of the storage allocation problem. By examining the key parameters of this problem, we give an overview of possible application scenarios. Afterwards, we formulate the optimization problem resulting from the storage allocation problem. Thereafter, we present our storage framework for peer-to-peer systems and show how the problem becomes feasible if it is solved in a distributed manner.

The formal problem definition models the problem of assigning suitable storage positions as a graph-based optimization problem [9]. It is based on an undirected graph $G = (V, E)$, namely the *connectivity graph*. Therefore, the graph contains a set of vertices v_i for every node i in the network. Furthermore, edges $e_j = \{v_k, v_l\}$ exist iff node k and node l can bidirectionally communicate.

The nodes have the capability to store (atomic) data items σ_i consisting of a *key* and a *value*, i.e.,

$$\sigma_i = \langle key, value \rangle$$

The `key` represents the geographical reference-position of the data item, the `value` may be any kind of data. The overall data stored in the network is denoted by $\Sigma = \cup \sigma_i$.

The goal of an in-network data storage algorithm is to assign each data item $\sigma \in \Sigma$ to a node $v \in V$, which will be responsible for its storage, delivery, and update management. Therefore, the data-item-to-node assignment can be expressed as a function

$$Allocate : \Sigma \rightarrow V$$

which represents the storage location for all data items.

Furthermore, the nodes' limitations are modeled as constraints. For example, the storage limitation of each node is expressed by a function $Mem: V \rightarrow \mathbb{N}$ representing the number of objects σ_i, which can be stored at a node at the same time. Using this function, the problem's memory constraint can be expressed as:

$$\forall v_i \in V : |\{ \sigma \mid Allocate(\sigma) = v_i \}| \leq Mem(v_i) \tag{2.1}$$

Each data item σ_i can be accessed by various nodes with varying frequencies and different operations, i.e., *put*, *get*, or *delete* operations. Therefore, the number of accesses n of a specific access-*type* to σ_i from a node $v \in V$ may be modeled as a tuple:

$$acc_{\sigma_i} = \langle v, type, num \rangle$$

$$\text{with } v \in V, type \in \mathbb{N}_+, num \in \mathbb{N}_+$$

A set of n different accesses to σ_i is denoted as

$$ACC_{\sigma_i} = \{\langle v_1, type_1, num_1 \rangle, \ldots, \langle v_n, type_n, num_n \rangle\}$$

To calculate the costs of an access, the following definitions from graph theory are used:

A path between two nodes from u to w is defined as a tuple:

$$p(u, w) = (\{u, v_1\}, \{v_1, v_2\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, w\})$$

The length of a path $p(v_1, v_2)$ is equal to the number of edges in the path and is denoted by $|p(v_1, v_2)|$. The set of all paths connecting nodes u and w is $P(u, w)$.

Let the shortest path between two nodes v_1 and v_2 in G be denoted as $d_G(v_1, v_2)$ and be defined as:

$$d_G(v_1, v_2) = \begin{cases} 0 & v_1 = v_2 \\ \min_{p \in P(v_1, v_2)} |p(v_1, v_2)| & P(v_1, v_2) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

The costs of an access represent the impact it has on the network's resources.

Since we are going to minimize the latency of accesses, the cost of an access is defined as the minimal number of hops needed to access a data item given an instance of a graph G. Moreover, the *access type* determines the messaging pattern used, i.e., the number of messages exchanged for a successful transaction. Therefore, the costs for one access pattern $acc_{\sigma_i} = \langle v, type, num \rangle$ given a graph G and an allocation $Allocate$ is defined as

$$costs(G, Allocate, acc_{\sigma_i}) = type \cdot num \cdot d_G(v, Allocate(\sigma_i))$$

2.4.1 Storage Latency Search Problem

The goal of the access-centric storage paradigm is to find an optimal solution for the placement of data items in a distributed peer-to-peer system with the primary goal to minimize the latency of issued queries measured by their hop counts. Given the problem description and constraints above, the *storage latency search problem* is defined as follows: Given a graph G and an access pattern ACC, find a complete data allocation function $Allocate$ which minimizes the following access costs:

$$\min \sum_{\sigma_i \in \Sigma} \sum_{acc \in Acc_{\sigma_i}} costs(G, Allocate, acc)$$
$$= \min \sum_{\sigma_i \in \Sigma} \sum_{acc \in Acc_{\sigma_i}} acc.type \cdot acc.num \cdot d_G(acc.v, Allocate(\sigma_i))$$

and meets all constraints in the form of formula (2.1).

2.5 Storage Framework

In the following section, we present our storage framework as a middleware which can be integrated into any sensor node model by implementing its interfaces. Figure 2.3 depicts an exemplary three-layered sensor node model with the storage middleware

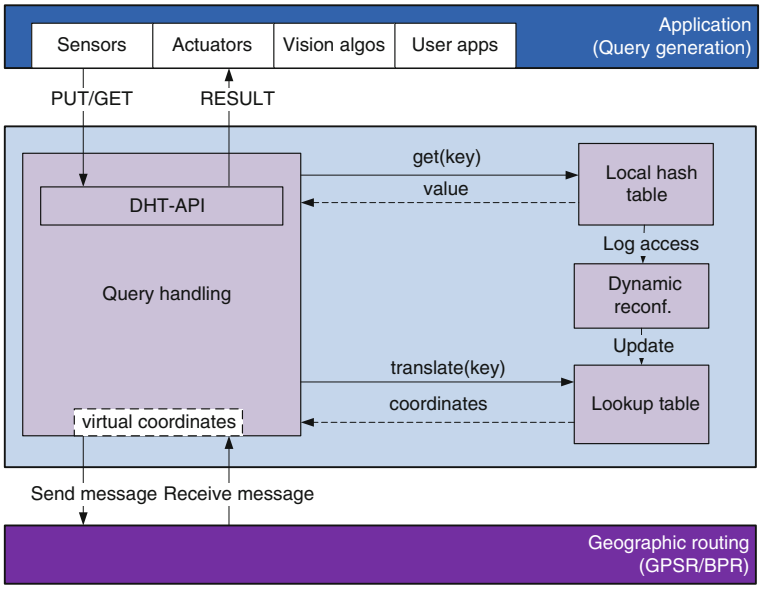

Fig. 2.3 Node architecture. The figure shows a three-layered sensor node with the storage middleware located between the *application* and the *routing* layer. The *arrows* represent the collaboration between the modules

placed between the application and the routing layer. In Sect. 2.5.2 we explain our distributed online optimization algorithm for self-organizing data storage.

2.5.1 Architecture

Our *storage middleware* is located between the *application layer* and the *routing layer* (see Fig. 2.3). As will be explained, our algorithm makes as few assumptions about these other layers as possible.

The *application layer* encapsulates any sensing or control algorithm that stores and retrieves georeference-based data. These algorithms may interact with any kind of sensor or abstract data. To make use of the storage framework, an application has to use the sleek interface of a distributed hash table as described in Sect. 2.5.1.1. Therefore, the placement and distribution of data in the network is fully transparent to the application. We abstract certain kinds of applications' behavior in *producer* and *consumer* modules.

The *routing layer* contains a geographic routing protocol, which operates on the nodes' positions during packet forwarding. In particular, our algorithms make use of the capability of these protocols to determine so called *home nodes*. A node is considered a *home node* for a specific coordinate, if it is the nearest node available

w.r.t. the euclidean distance. However, it is important to note that the home node might change due to node churn or movement.

The *storage middleware* is located in the layer between. It translates coordinates, which are the keys of the queries, to virtual coordinates, which represent the actual storage location of data items. This is done using a Lookup table that is available at each node (see Sect. 2.5.1.2). Due to this translation, the whole optimization and migration process is transparent to both, the routing and the application layer. Therefore, standard geographic routing algorithms may be used on the lower layer. However, cross-layer optimizations may improve the performance as will be discussed in Sect. 2.5.1.3.

To offer a general and extensible framework, we follow a clean tier-based approach as suggested in [5]. We chose a three-tiered approach which well be presented in the following sections: *Tier 2* contains the interface to the application layer. *Tier 1* contains the key-space transformation which translates real geocoordinates to their respective virtual coordinates, which represent the storage location of the associated data items. And *Tier 0* implements the interface to a geographic routing protocol.

2.5.1.1 Tier 2—API

In order to achieve a generic applicability for all kinds of application-level algorithms, the storage layer offers a concise and general-purpose frontend API of a distributed hash table (DHT-API). This way, the applications can make use of a highly standardized interface. The available operations are:

- `put(key,value)`
- `value = get(key)`
- `delete(key)`

In the context of network performance and workload, the main difference between these operations is whether they are one-way messages with no return value, in this case *put* operations, or if they require a returning message, e.g., *get*. Therefore, delete messages are considered as to resemble one of the two kinds. In fact, this behavior is represented by an *access type* parameter (see Sect. 2.4) which contains the number of messages necessary for a complete interaction. This way, even more complex query-response-acknowledgment-patterns can be represented.

2.5.1.2 Tier 1—Key-Space Transformation

The key-space transformation tier represents the main part of the algorithm. Its main purpose is to translate keys, i.e., their geopositions, to their virtual pendants in order to determine the current storage positions. Therefore, it implements the following function:

```
geoposition = translate(key)
```

While the `translate` operation is the main external function, this layer needs some internal helper modules to offer its adaptive service. These three modules are (see Fig. 2.3):

- The layer's **Lookup table** provides valid, up-to-date storage nodes for queries at any point in time.
- The **dynamic reconfiguration module** contains a statistics module. It takes care of creating access models and determines the time and amount of migrations (see Sect. 2.5.2).
- The **local hash table** stores data for which the node is responsible and logs accesses.

2.5.1.3 Tier 0—Key-Based Routing

Although the storage position is determined based on virtual coordinates, the routing happens on the actual network using geographic routing mechanisms, e.g., GPSR [14] or Bi-Perimeter Routing (BPR) [7]. The implementation of the virtual address translation allows for using standardized and thoroughly researched geo-routing protocols on the routing layer.

Since all changes of storage locations are transparent to the routing layer, a mechanism is necessary to enable the storage layer to update packets so that they reach their correct destinations. Depending on the implementation of the storage layer and the possibilities to extend the geo-routing protocol in use, different options are available (see 1 and 2 in the following):

1. *No changes to the routing protocol*: If it is unfeasible to modify the routing protocol's behavior, this means that every packet handed over to the routing layer will be routed all the way to its destination, i.e., the home node for the destination's coordinates. If the destination is no longer responsible for a data item, it has to reroute the arriving packets. As long as the key-space transformation information is kept synchronized on all nodes, this standard behavior of a routing protocol is sufficient for the algorithm to work efficiently.
2. *Cross-layer design*: There are several reasons why every intermediate node on a route of a network packet should be able to update its destination. The most important reason is that a fully synchronized state of the Lookup table during runtime is unfeasible for most application scenarios. While there is only a small probability for them to happen while a packet transmission is in the progress, many events may lead to an invalid destination coordinate in a packet, e.g., data migration, node churn, node's memory exceeded. Moreover, as will be pointed out below, due to performance and scalability reasons, most implementations of the proposed storage framework will include a distributed solution of the key-space transformation. For instance, the usage of an information distribution strategy, which only ensures *eventual consistency*, may lead to inconsistent states of the Lookup table.

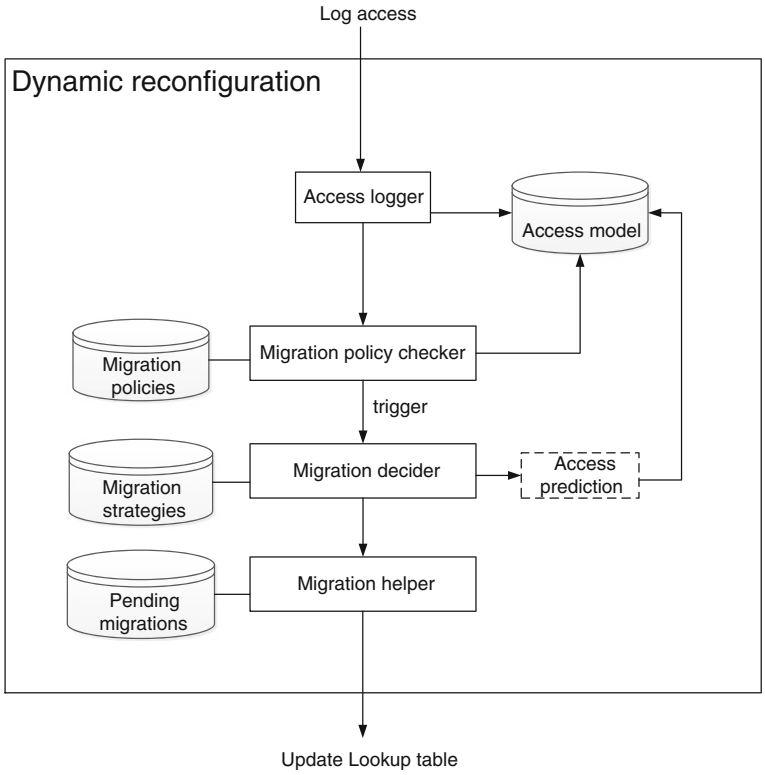

Fig. 2.4 Dynamic reconfiguration module. This module is a submodule of Fig. 2.3 and contains the distributed online optimization algorithm

2.5.2 Dynamic Online Storage Reconfiguration

The main goal of access-centric storage is to reach an optimal storage allocation that minimizes the network load according to Sect. 2.4.1. Our algorithm solves this problem in a distributed way as each node is responsible for the initialization, handling, and announcement of migrations of data items which it currently stores.

The main functionality of our algorithm is encapsulated in the *dynamic reconfiguration module* of our architecture (see Fig. 2.4). The data migration algorithm is separated into two phases:

1. At first, the *migration policy checker* identifies potential migration pressure in the access-log and decides *when* to perform a migration.
2. Afterwards, the *migration decision* module is triggered to decide on the actual migrations, i.e., *where* to migrate data.

Such a separation is reasonable especially to offer different sets of configurations for both submodules. While the migration policy checker and the migration

decision modules implement the main functionality of the access-centric storage allocation, additional helper modules take care of the access-logging and the migration execution.

2.5.2.1 Data Migration Algorithm

Since no assumption about the behavior of possible consumers of data is made at start-up time, a location-centric storage approach is chosen at first. This means each node is responsible for its surrounding key space, i.e., they store data associated with close-by events. Considering the locality of the sensing ranges from applied sensors, this means that most data will be stored at the sensing node itself or a nearby node. From the algorithm's point of view, this means that the *Lookup table* is empty during startup. For as long as no migration has taken place, it simply returns the real-world location of an event leading to the desired location-centric storage behavior.

During runtime, the accesses to data items for which the node is responsible are recorded. This is done by the *access logging* submodule which offers an interface to log the source coordinate, the destination coordinate, and the access type. This information resembles the access format from the formal problem description from above. Its main configuration parameters are its spatiotemporal resolution and the size of its backlog. Depending on the application domain and its requirements, this module can be used for converting or quantizing the coordinates to a specific resolution.

The migration policy checker regularly applies data mining techniques to recognize potential optimizations through data migration. When a data item σ_i is accessed, it tests the accesses against the given migration policies. If the number of accesses surpasses a given threshold $\boldsymbol{\tau}$, the *migration decider* module is triggered. Based on the current access model, the optimal position for a data item is determined using the following formula:

$$\text{Optimal Position}(\sigma_i) = \frac{\sum_{acc \in Acc_{\sigma_i}} acc.pos \cdot acc.n}{\sum_{acc \in Acc_{\sigma_i}} acc.n}$$

Typically, the calculated coordinate of the optimal position is located somewhere between nodes. Therefore, the actual target position for the migration is determined by sending a soliciting message toward the optimal position. The geographic-routing protocol delivers it to the nearest node next to this position which is nominated the new home node of the data. After the migration took place, the Lookup table is updated accordingly.

The migration policy checker may not only evaluate accesses to single data items but also to chunks of them. The size of the data mined key-space around an accessed data item is given by the *migration chunk size m*. If a group of data items gets migrated, the optimal position is computed over the whole set of σ_i in the chunk. As the evaluation will show, *m* is an important design parameter (see Sect. 2.6.6).

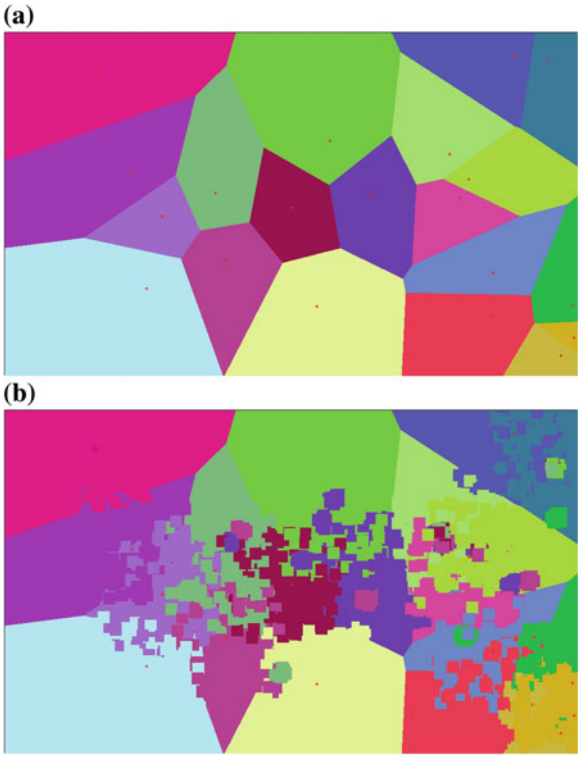

Fig. 2.5 **a** Initial storage allocation. The nodes are represented by *red dots*. Their surrounding areas represent their current area of responsibility for which they store data. Since the Lookup table is empty at start-up, the storage allocation resembles a Voronoi distribution. **b** Exemplary storage allocation after a certain simulation period of Fig. 2.5a scenario. Again, the positions of the nodes are indicated by *small red dots*. Still, the different responsibilities are illustrated by different colors, where each specific color defines one of the nodes. Each migration resembles an entry in the Lookup table

2.5.2.2 Example

An initial storage allocation of the algorithm, which results from an empty Lookup table, is shown in Fig. 2.5. The map represents the distribution area of the sensors. Each sensor is represented by a red dot surrounded by its region of responsibility, i.e., it stores all events whose geolocation lies inside this area. This location-centric behavior resembles a Voronoi partition.

Figure 2.5b depicts the virtual coordinate space after a certain simulation period. With the colors still representing the nodes' responsibilities, one can see that a number of migrations took place. The frayed borders where the colors mix may originate from neighboring nodes surveying events in the other nodes' vicinities. In the given example, access-optimization with respect to *put* operations is responsible for

migrating the data items from one node to another. These are short-way migrations optimizing the initial assignments. One example of a farther migration due to various nodes accessing the data using *get* operations is the navy-green area in the yellow region in the lower middle. This migration has taken place due to frequent accesses to a data item that originally resided in the yellow region. However, these virtual coordinate maps do not allow reliable conclusions on how many and which nodes have actually accessed certain data items. Rather, they provide a qualitative overview of the migration results.

2.5.2.3 Discussion of Design Parameters

One major aspect of storage allocation optimization during runtime is the kind of algorithm used. While some components can be implemented in a distributed fashion without loss of consistency, the Lookup table needs a certain level of internode synchronization. A key decision which has to be made by any developer is whether he wants to create a system which relies on centralized entities like servers or synchronized network nodes. Depending on the application's constraints it will be necessary for the storage system to offer certain guarantees of consistency. In distributed systems, this most often means that there are elected nodes or servers which act as synchronization points, i.e., by using synchronization algorithms like distributed barriers and election algorithms that enforce the requested level of consistency. However, all of these algorithms increase the message transfer overhead. That is why, most often, weaker consistency models are applied, e.g., eventual consistency. So the decision for a consistency model may have great impact on the network load, which is caused by the exchange of synchronization messages.

Another important design parameter of our algorithm is the choice of statistics which the nodes use during runtime. This choice has a direct impact on the algorithms performance. A main design parameter is the *quantization* or *accumulation* of key space which is the size of distinct regions for which access models are build. With an increase of detail, the local access models will grow and, proportionally, the algorithm's memory footprint. At the same time, the increase in detail enables for much more fine-grained migration decisions. Therefore, the key-space quantization is one important tradeoff criteria between the nodes' local resources, i.e., memory usage and computation time, and lowered network utilization as benefit from more fine-grained migration decisions. Since memory is getting cheap, the benefits of the proposed algorithm will overhaul its costs.

To show the applicability of the algorithm, the access-log uses a *per data item* statistical approach. This resembles the notion in the problem description (see Sect. 2.4). We analyze the design space of the parameter key-space quantizations for the access models in Sect. 2.6.6, showing that this parameter has a great impact on the performance. Therefore, not only single data items but also chunks of adjacent data items are considered.

Another design parameter of the statistics module is the model-building characteristic of accesses, i.e., the *sample size* of Acc_{σ_i} (see Sect. 2.4). The current approach

considers the number of updates since the last migration of σ_i. More elaborate statistic modules can easily be applied to the presented architecture.

2.6 Evaluation

In this section, we explore the main characteristics of our proposed solution. We created a reference implementation of the storage framework presented in Sect. 2.5.

As has been discussed in Sect. 2.5.2.3, some components of the framework need a thorough examination of whether they are implemented in a centralized, synchronized, or distributed way. To get a grasp of the framework's main attributes and to maintain as application scenario independent as possible, the Lookup table is implemented in a fully synchronized data structure. In a real-world application, this could either be a centralized server component or a data store which is synchronized between all nodes. Depending on the actual implementation, this leads to different types and amounts of message exchanges. Since these decisions are more application specific, they are not explicitly modeled here. More specific evaluations can be performed for specific application scenarios.

The metric used to quantify the results is the hop count of messages issued for the different access types, i.e., *put* and *get* operations. After explaining the experimental setup, we present the results of different scenarios.

2.6.1 Experimental Setup

Our experiments are carried out using the discrete-event simulator *OMNeT++* [25] extended by the *MiXiM* [15] simulation package, which offers simulation models for IEEE 802.11b/g wireless LAN. All network nodes are placed randomly for each run and were connected through their wireless LAN interface with a radio range of 160 m. On the routing layer, we use an implementation of GPSR [14]. During a start-up period of 60 s the *producer* and *consumer* applications are paused so that the graph planarization of the routing protocol can take place. Although graph planarization runs all the time to handle nodes joining and leaving, this period ensures a reasonable starting point.

To model the characteristics of smart sensing systems, all *producers* output sensing data with keys originating in a surrounding 100 m^2 square. Each *consumer* queries equally distributed geolocations in up to five regions. These regions are chosen randomly by each consumer during initialization. Both types of applications generate queries at regular time intervals according to Table 2.1. The number of *producers* and *consumers* are varied to demonstrate the adaptability of the system to different and dynamically changing scenarios. The migration threshold τ is set to 10 accesses. The scenario-specific parameters of the experimental setup are listed in Table 2.1.

Table 2.1 Simulation setup

Parameter	Scenario 1 + 2 dynamic appl.	Scenario 3 network size	Scenario 4 query rates	Scenario 5 migration
Field size	1,000 × 600 m^2	1,500 × 1,500 m^2	1,000 × 600 m^2	1,000 × 600 m^2
Number of nodes	20	50, 100, 150	30	20
Simulation time	2.75 h	4 h	4 h	4 h, 80 h
Number of producers	**5→10→15→20**	**50, 100, 150**	**30**	**20**
PUT generation rate	12/min	12/min	12/min	**20/min**
Size of PUT fields	100 × 100	100 × 100	100 × 100	100 × 100
Number of consumers	**5→10→15→20**	**10**	**10**, **20**, **30**	**10**
GET generation rate	6/min	6/min	6/min	12/min
Size of GET fields	20 × 20	40 × 40	40 × 40	20 × 20
GET areas per consumer	5	5	5	**1**
Migration field size *m*	20 × 20	20 × 20	20 × 20	1 × 1, 4 × 4, 10 × 10, 16 × 16, 20 × 20, 60 × 60

The following graphs show the average number of hops across the network for *get* and *put* operations, respectively. The hop counts for *get* operations accumulate both, the queries and the resulting replies. Furthermore, we show the moving average of the accumulated hop counts for *puts* and *gets* to research the overall hop count optimization.

2.6.2 Scenario 1: Fixed Behavior of Applications

At first, we present results from an experiment in which the characteristic behavior of the *producers* and *consumers* are chosen at the start of the simulation and remain unchanged during the simulation's run. This way, the general behavior of our framework can be observed.

This scenario contains 20 sensing nodes, which acquire data from their surroundings and store it using the storage framework. Five of the nodes are equipped with *consumers*, which represent algorithms consuming data from the storage layer. Therefore, each consumer chooses five regions of interest with a size of 20 × 20 m^2 during start-up. They frequently query their regions of interest by accessing randomly chosen keys from them using *get* operations issued to the storage framework.

Figure 2.6 shows the results from this experiment. At the beginning, one can observe the hop counts that arise from the initial distribution of the data. Since the Lookup table is empty at start-up, all data is stored on the node that is next to the event's position. This resembles the behavior of a location-centric storage paradigm. The average hop count for the put operations is greater than zero, since the sensing node is not necessarily the nearest node to the event. The mean number of hops for

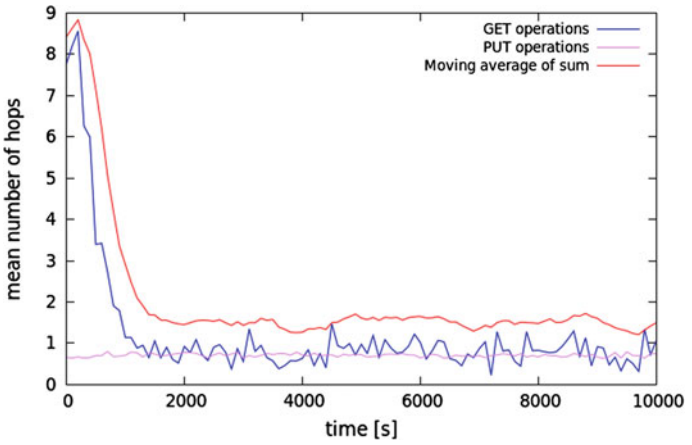

Fig. 2.6 Fixed behavior of applications. The figure depicts the development of the mean hop count over time for fixed application behavior. The *red line* shows the accumulated and smoothed sum of the hop counts for PUT and GET operations. It can be seen that the system adapts to the given access behavior by migrating accessed data to optimal positions between producers and consumers leading to a huge reduction in network load

get operations of consumers querying the data is high. It can be seen that the network load is mainly caused by the get operations. Due to the huge migration pressure issued by the accesses, the system optimized the storage locations quickly and the mean hop count decreases by about 75 % until 2,000 s.

Since the nodes generate queries that are equally but randomly distributed in their regions of interest, small amounts of migration pressure arise and vanish throughout the remainder of the experiment. The algorithm's continuous optimization leads to localized short-way migrations. This behavior causes the jitter of the hop counts. It could be avoided by more elaborated migration policies which adapt to this situation and avoid short-way migrations.

2.6.3 Scenario 2: Dynamic Behavior of Applications

In order to demonstrate the adaptability of our algorithm, we applied dynamically changing application behavior during runtime. While the *put* operations mainly depend on the sensors' characteristics and settings, the *consuming* applications may appear and vanish unpredictably during runtime. We modeled this behavior by spawning five new *consumers* periodically every 2,500 s. While we only change the behavior of the consumers in this experiment, the results would be similar if we changed the producers accordingly. As stated above, from the framework's point of view, the access type only determines the number of message exchanges necessary. This way, the actual operation is in fact reduced to this relative value.

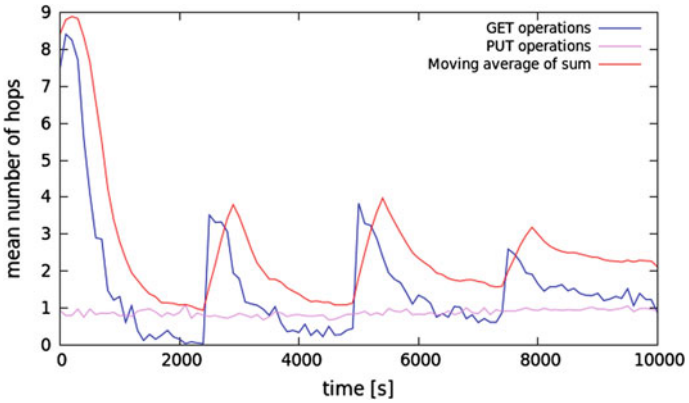

Fig. 2.7 Dynamic behavior of applications. This scenario extends the scenario of Fig. 2.6 by adding five consumers at $t = 2{,}500\,\text{s}$, $t = 5{,}000\,\text{s}$, and $t = 7{,}500\,\text{s}$, respectively. The changing access patterns lead to a rising mean number of hops. The algorithm reacts to the changing behavior with new migrations which quickly lowers the mean hop count again

The results of these runs are depicted in Fig. 2.7 and show the fast adaptations taking place. After start-up, the first 2,500 s yield results that qualitatively resemble the results in Fig. 2.6. The periodic introduction of new consumers (first time at $t = 2{,}500\,\text{s}$) leads to new access patterns and, thus, the hop count increases. The dynamic reconfiguration of our algorithm reacts to the migration pressure with storage reallocations. As a consequence, it drastically reduces the hop counts for subsequent access operations. The same qualitative performance increases are achieved after the subsequent introductions of new consumer nodes at $t = 5{,}000\,\text{s}$ and $t = 7{,}500\,\text{s}$.

One can observe that the lower limit of hops required after the optimizations increased slightly, e.g., by comparing the mean hop counts at $t = 2{,}000\,\text{s}$ and at $t = 7{,}000\,\text{s}$. This is due to the increased number of consumers which lead to an absolute increase in access operations. Moreover, it can be observed that the mean hop counts of put operations increase only slightly. The reason for this is that still much data is stored locally, since only accessed data items (and their surrounding key space) are migrated. This shows the main advantage of our framework which only migrates data which is actually accessed or in the vicinity of accessed data.

2.6.4 Scenario 3: Different Network Sizes

This scenario shows how the framework scales in networks of different size. A larger field size of $1{,}500 \times 1{,}500\ \text{m}^2$ is chosen for these experiments to investigate networks containing up to 150 nodes. This way, the node density of the network with 100 nodes has a similar node density compared to the other scenarios. The experiments with

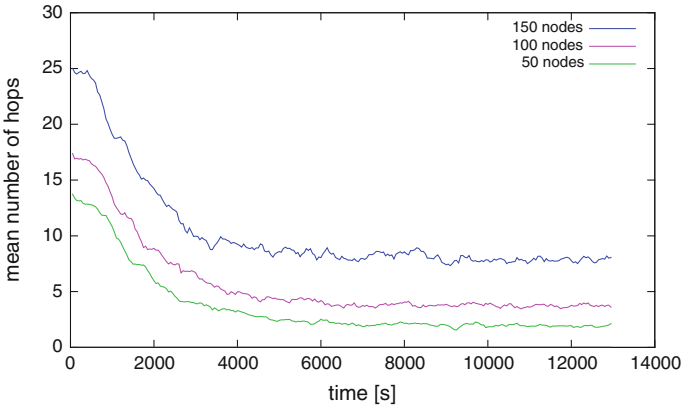

Fig. 2.8 Different network sizes. The figure depicts the development of the sum of the mean hop counts over time. It shows the results for three scenarios containing 50 (*bottom line*), 100 (*middle line*), and 150 nodes (*top line*). It can be seen that the algorithm's performance scales with the size of the network and always leads to huge latency reductions

50 and 150 nodes show the behavior in relation to this value. While increasing the number of nodes (each one acting as a producer), we keep the number of consumers constant. This models a given set of accessing algorithms with different numbers of sensors available.

We present the results for networks of 50, 100, and 150 nodes in Fig. 2.8. The graphs show the sum of mean hop counts for put and get operations in each scenario. It is obvious that the qualitative reduction of the hop counts is similar in all optimizations and leads to reductions from 60 to 80 % of the initial values. The absolute hop counts increase proportionally to the number of nodes in the network. This is caused by the higher density of nodes which directly leads to longer shortest paths between two nodes in the connectivity graph.

2.6.5 Scenario 4: Different Query Rates

After examining the consequences of different networks sizes given a fixed number of consuming algorithms, this experiment researches the optimization results for different numbers of consumers in a network with 30 nodes. Therefore, the number of consumers is increased from 10 up to 30 consuming nodes. This drastic increase in consumers in comparison to the number of sensing nodes enforces more overlaps in the areas of interest of the consumers. This leads to migration decisions that take more access nodes into account.

As can be seen in Fig. 2.9, the optimization leads to a reduced mean number of hops between 40 and 60 %. This shows that the algorithm even copes with a large number of algorithms consuming data and reduces the network load significantly.

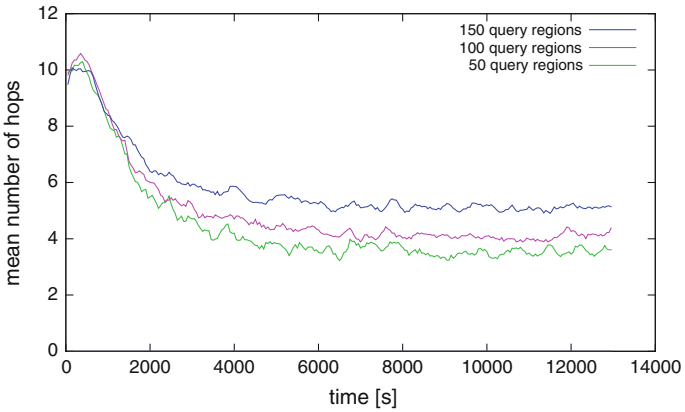

Fig. 2.9 Different query rates. This figure shows the evaluation of a scenario with 30 nodes and with 10 (*bottom line*), 20 (*middle line*), and 30 consumer applications (*top line*). It shows that the algorithm performs very well with a large number of consumers

2.6.6 Scenario 5: Migration Field Size

As stated in Sect. 2.5.2.3, an important parameter of the optimization algorithm is the chunk size of migrations performed. The migration chunk size m is a parameter for the migration policy checker as well as the migration decider. If a data item is accessed, itself and its neighborhood of size m is considered as one chunk and checked by the migration policy checker. If the number of accesses in this area reaches the access threshold τ, a migration is triggered and the migration decider calculates the new storage location. Therefore, it takes into account all accesses to data items in the chunk and initiates a migration of the whole chunk. While the theoretical optimum of the search problem can only be reached if the storage position of every single key is optimized, such a behavior is unfeasible for most applications and scenarios.

Figure 2.10 shows the results for a given network configuration with migration chunk sizes m from $4 \times 4\,\mathrm{m}^2$ up to $60 \times 60\,\mathrm{m}^2$. The results for $m = 1 \times 1\,\mathrm{m}^2$, which represents the exact solution that considers every data key on its own, are depicted in Fig. 2.11. Since the applications scatter their accesses using an equal distribution in each area of interest, the exact solution has the drawback that it runs a long time until the access threshold τ is reached. To show its long-term development, the exact scenario has been run over 80 h of simulation time.

As can be seen, the migration chunk size directly influences how fast a migration is triggered. This is because all keys in the chunk are considered by the migration policy checker. While the exact migration is the slowest, it should reach the best result in the long run. However, this setup already shows its drawback. Especially in a real-world setup, the measured location of an event will be subject to different kinds of noise, e.g., a tracking event which gets converted into world coordinates. To handle this kind of noisy data, the framework should quantize the key space.

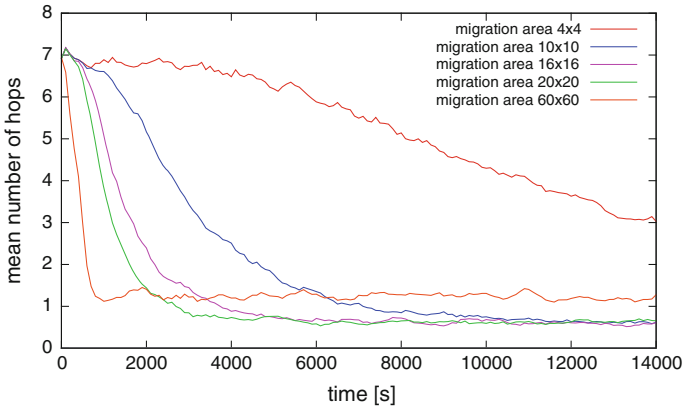

Fig. 2.10 Different migration chunk sizes. The figure shows the algorithm's behavior for the same scenario using different migration chunk sizes from 4×4 (*red line*) up to 60×60 (*orange line*). An increasing chunk size leads to a much faster decrease in the mean number of hops. A drawback of too large chunks are suboptimal migration decisions by taking to many different accesses into account, as can be seen from the *orange line*, which does not reach the minimum

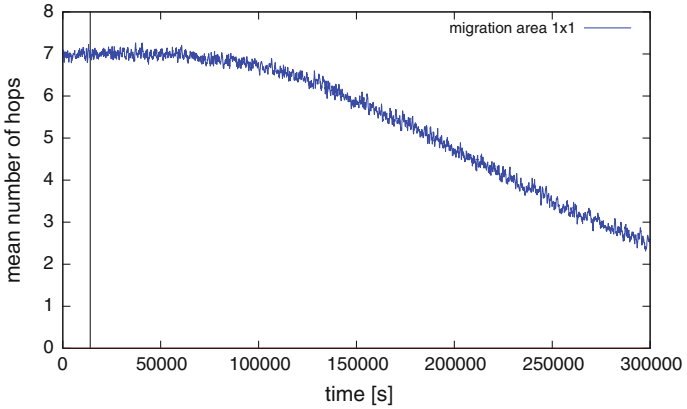

Fig. 2.11 Different migration chunk sizes—the exact solution. This scenario shows the algorithm's behavior if accesses to single data items are considered. The vertical line marks the runtime of the other scenarios. It shows the same qualitative behavior of the runs in Fig. 2.10 but requires a long time for the optimizations to take place. It could be speeded up by lowering τ

On the other hand, if the chosen chunk size gets to large, the resulting optimizations may be worse than for smaller chunk sizes. This can be observed for chunks of $60 \times 60\,\mathrm{m}^2$ which lead to a suboptimal storage allocation. This happens since too many unrelated accesses are considered at once. This leads to suboptimal migration decisions.

The evaluation shows that a reasonable chunk size, between $10 \times 10\,m^2$ and $20 \times 20\,m^2$, offers very good storage location optimizations incorporating the advantages of being responsive as well as robust towards noisy data. Migration chunks are especially useful if the consuming algorithms issue queries concerning certain ranges or to quantify the key-space to a scale applicable for the application scenario, e.g., appropriate for the magnitude of noise induced by other system components.

2.7 Conclusion

This chapter presented a novel approach to usage-centric storage and access of sensor data. Based on the insight that current geolocated sensor networks perceive large amounts of data while simultaneously accessing this data within the network, we discussed different approach of the state of the art to optimise this data access. We presented a distributed storage algorithm whose design goal is the online optimization of in-network storage allocation of georeferenced data.

The evaluation demonstrates the potential benefit of this novel approach. We investigated the effects of varying consumers (i.e., applications accessing data items in the network) and varying network size. The result shows a significantly improved hop count—meaning that the latencies in terms of visited nodes until the desired information is available to the requesting application are decreasing significantly. Thereby, the algorithm is robust against joining and leaving nodes. It also scales with the number of nodes contained in the network and is not characterized by drawbacks such as single-points-of-failure.

Our current and future work investigates possibilities to apply distributed algorithms which rely on eventual consistency. We aim for a robust algorithm which is able to offer the presented benefits with a reduced overhead. Therefore, we are going to model the caused costs by overhead messages and their relationship to the degree of synchronization. Moreover, we want to apply the system to different technologies. An important aspect will be the latency difference in mixed wired and wireless networks. Since latency in wireless networks may be magnitudes worse and less predictable compared to wired networks, wired nodes could be used as a dynamic backbone for queries issued from mobile units. Therefore, such queries may be handled by a query proxy which is chosen w.r.t. node characteristics. Another direction of research will cover the extension to accompanying replication algorithms to handle unexpected node failure.

References

1. Abdelzaher T, Bhattacharya S, Kim H, Prabh S (2003) Energy-conserving data placement and asynchronous multicast in wireless sensor networks. In: Proceedings of mobisys 2003: the first international conference on mobile systems, applications, and services

2. Aghajan H, Cavallaro A (eds) (2009) Multi-camera networks-principles and applications. Elsevier
3. Akyildiz IF, Vuran MC (2010) Wireless sensor networks. Wiley, New York
4. Albano M, Chessa S, Nidito F, Pelagatti S (2007) Q-NiGHT: adding QoS to data centric storage in non-uniform sensor networks. Sensors (Peterborough, NH), pp 166–173
5. Dabek F, Zhao B, Druschel P, Kubiatowicz J, Stoica I (2003) Towards a common API for structured P2P overlays. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS03), vol 2735, pp 33–44
6. D'Angelo D, Grenz C, Kuntzsch C, Bogen M (2012) CamInSens-an intelligent in-situ security system for public spaces. In: International conference on security and management (SAM), Las Vegas, Nevada
7. Dudkowski D (2009) Fundamental storage mechanisms for location-based services in mobile ad-hoc networks. Ph.D. thesis, Universität Stuttgart
8. Grenz C, Hähner J (2011) PhD forum: adaptive storage management in highly heterogeneous smart sensor systems. In: 5th ACM/IEEE international conference on distributed smart cameras, ICDSC 2011
9. Grenz C, Hähner J, Asam F (2013) Application-independent in-network storage optimization for distributed smart camera systems. In: Seventh international conference on distributed smart cameras (ICDSC)
10. Grenz C, Jänen U, Hähner J, Kuntzsch C, Menze M, D'Angelo D, Bogen M, Monari E (2012) CamInSens-demonstration of a distributed smart camera system for in-situ threat detection. In Proceedings of international conference on distributed smart cameras (ICDSC)
11. Hoffmann M, Wittke M, Hähner J, Müller-Schloer C (2008) Spatial partitioning in self-organizing smart camera systems. IEEE J Sel Top Signal Process 2(4):480–492
12. Jaenen U, Feuerhake U, Klinger T, Muhle D, Hähner J, Sester M, Heipke C (2012) Qtrajectories: improving the quality of object tracking using self-organizing camera networks. In: ISPRS annals of the photogrammetry, remote sensing and spatial information sciences, vol I-4, pp 269–274
13. Jaenen U, Spiegelberg H, Sommer L, von Mammen S, Brehm J, Haehner J (2013) Object tracking as job-scheduling problem. In: Seventh international conference on distributed smart cameras (ICDSC) IEEE, pp 1–7
14. Karp B, Kung HT (2000) Greedy perimeter stateless routing for wireless networks. In: Proceedings of the sixth annual ACM/IEEE international conference on mobile computing and networking (MobiCom), Boston, pp 243–254
15. Köpke A, Swigulski M, Wessel K, Willkomm D, Klein Haneveld PT, Parker TEV, Visser OW, Lichte HS, Valentin S (2008) Simulating wireless and mobile networks in OMNeT++ the MiXiM vision. In: Proceedings of the first international conference on simulation tools and techniques for communications networks and systems (ICST)
16. Kumar B (2008) ZGHT- a zonal hash-table for data-centric storage. TAMU Comp Sci, College station, TX 77840
17. Nam Le T, Yu W, Bai X, Xuan D (2006) A dynamic geographic hash table for data-centric storage in sensor networks. In: IEEE wireless communications and networking conference (WCNC), pp 2168–2174
18. Monari E, Pollok T (2011) A real-time image-to-panorama registration approach for background subtraction using pan-tilt-cameras. In: International conference on advanced video and signal based surveillance (AVSS), IEEE computer Society, pp 237–242
19. Monari E, Voth S, Kroschel K (2008) An object-and task-oriented architecture for automated video surveillance in distributed sensor networks. In: IEEE fifth international conference on advanced video and signal based surveillance (AVSS), pp 339–346
20. Ratnasamy S, Francis P, Handley M, Karp R,Shenker S (2001) A scalable content addressable network. In: Proceedings of the 2001 conference on applications., technologies, architectures, and protocols for computer communications (ACM SIGCOMM), vol TR-00-010, University of Berkeley,CA, pp 161–172

21. Ratnasamy S, Karp B, Yin L, Yu F, Estrin D, Govindan R (2002) GHT: a geographic hash table for data-centric storage. In: Proceedings of the first ACM international workshop on wireless sensor networks and applications (WSNA)
22. Rinner B, Wolf W (2008) Proc IEEE 96(10):1565–1575
23. Shenker S, Ratnasamy S, Karp B, Govindan R, Estrin D (2003) Data-centric storage in sensornets. ACM SIGCOMM Comput Commun Rev 33(1):137–142
24. Stoica I, Morris R, Karger D, Frans Kaashoek M, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, New York, pp 149–160
25. Varga A, Hornig R (2008) An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops (Simutools)
26. Wittke M, Grenz C, Hähner J (2011) Towards organic active vision systems for visual surveillance. In: Berekovic M, Fornaciari W, Brinkschulte U, Silvano C (eds) Architecture of computing systems-ARCS 2011. Springer, Berlin, pp 195–206

Springer

http://www.springer.com/978-3-319-10806-3

Human Behavior Understanding in Networked Sensing
Theory and Applications of Networks of Sensors
Spagnolo, P.; Mazzeo, P.L.; Distante, C. (Eds.)
2014, XVIII, 459 p. 233 illus., 208 illus. in color.,
Hardcover
ISBN: 978-3-319-10806-3