Collaborative Information Processing and Query Evaluation in Wireless Sensor Networks

A Dissertation Presented

by

Xianjin Zhu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2008

Stony Brook University
The Graduate School

Xianjin Zhu

We, the dissertation committee for the above candidate for the degree of Doctor of Philosophy, hereby recommend acceptance of this dissertation.

Dr. Himanshu Gupta, Dissertation Advisor
Assistant Professor, Department of Computer Science

Dr. Samir R. Das, Chairperson of Defense
Associate Professor, Department of Computer Science

Dr. Jie Gao, Committee Member
Assistant Professor, Department of Computer Science

Dr. Goldie Nejat, Outside Committee Member
Assistant Professor, Department of Mechanical Engineering,
Stony Brook University

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Collaborative Information Processing and Query Evaluation in Wireless Sensor Networks

by

Xianjin Zhu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2008

Data-centric is one of the most important features that make wireless sensor networks distinct from other types of communication networking systems. A sensor network usually generates massive amount of data, but users only query quite high-level summarized information. Thus, information processing and query evaluation become fundamental problems in sensor networks.

The goal of this dissertation is to explore the potentials of sensor networks as collaborative data processing engines. It is expected that in the near future, sensor networks will reactively impact the physical world and interact with end users, who stay in the same physical domain and query the sensor network anytime anywhere. In that context, it requires that sensor nodes collaboratively process information in an ad hoc manner rather than resorting to a centralized base station for post-processing. We investigate essential grand challenges of collaborative processing and query evaluation in wireless sensor networks, and aim to improve the accessibility, interactivity and shareability of sensor data.

The first key problem of information processing is how to link users' selective queries with relevant information. It is challenging because both queries and related data can appear anytime anywhere in the network. Furthermore, a complex query may depend on multi-dimensional data collected by different types of sensors, which themselves can be distributed far apart. Thus, how to match those different types of data is the other aspect of this brokerage problem we need to handle. We proposed algorithms for in-network join of multiple data streams in a sensor network, based on the observation that a sensor network can be viewed as a distributed database system. One of our proposed approaches, viz., the Perpendicular Approach, is load-balanced, and results in substantially prolonging the network lifetime. The Perpendicular Approach is further extended to a general double ruling scheme for information brokerage.

The second challenge of information processing comes from the diversity of queries. Some queries request explicit information, e.g., temperature at a particular location. Some may ask for more implicit information, e.g., is there a traffic-free path. Different queries request different processing techniques. Queries for implicit information is especially challenging to be answered, since they usually require global knowledge that is hard to be obtained through sensor's local view. We investigated on a group tracking problem as a specific example of processing implicit queries. We proposed a light-weight contour tracking algorithm to process implicit contour information and its topological features. This algorithm performs a foundation for further information processing of spatial sensor data.

Thirdly, the underlying deployment environment has fundamental effects on high level tasks. Designing protocols for a specific deployment is expensive and time-consuming. Thus, it is highly desirable to have a generic approach to handle sensor fields with complex shapes, and make the design of new protocols transparent to the deployment specifics. We proposed a segmentation algorithm that partitions an irregular sensor field into nicely shaped pieces such that existing algorithms and protocols can be reusable inside each piece. Across the segments, problem dependent structures specify how the segments and data collected in these segments are integrated.

The ultimate goal of information processing is to return useful information to users. Thus, it is essential to provide friendly programming paradigm. We propose a deductive framework for programming and querying sensor networks. In this framework, sensor networks work as collaborative data processing engines and allow users to specify with ease the high-level functionality of an application, while hid from the low-level details. All of the above proposed collaborative processing techniques can be fundamental blocks and integrated into this framework.

Dedicated to my parents.

Contents

List of Tables

List of Figures

Acknowledgements

I am grateful to my advisor, Professor Himanshu Gupta, for giving me great advice and guidance during my graduate studies. Himanshu opened the door of sensor-network research to me and gave me a lot of freedom to explore this field. Besides advice on research, I will also always remember the support and encouragement he gave me on other aspects. He was always there at every frustrating moment I had. Without his help and encouragement, this dissertation would not be possible.

I am grateful to Professor Jie Gao for her great advice. It is my honor to get the opportunities to work with Jie. She shared a lot of nice ideas with students and gave me deep insights into the research areas of sensor networks and geometry. Jie is not just a good advisor, but also a great friend. She always gave me warm cares and helps on study and life.

I would like to thank Professor Samir Das, for his kind advice on my research. I would like to thank Professor Goldie Nejat, for being my committee member and giving me feedback on my dissertation work. I want to thank all my coauthors: Himanshu Gupta, Jie Gao, Samir Das, Joseph Mitchell, Rik Sarkar, Bin Tang, Zongheng Zhou and Xiang Xu. Working with them has always been enjoyable and productive. I also want to thank all my friends at Wireless Networking and Simulation Lab: Jing Cao, Pralhad Deshpande, Shweta Jain, Anand Kashyap, Ritesh Maheshwari, Vishnu Navda and Anand Prabhu Subramanian. They made the group warm and my life at Stony Brook colorful and fun.

Finally I want to thank my parents for their endless love. This dissertation is dedicated to them.

Publications

1. Xianjin Zhu, Rik Sarkar, Jie Gao, Segmenting A Sensor Field: Algorithms and Applications in Network Design. To appear in ACM Transactions on Sensor Networks.
2. Xianjin Zhu, Rik Sarkar, Jie Gao, Joseph S. B. Mitchell, Light-weight Contour Tracking in Wireless Sensor Networks. Proc. of the 27th Annual IEEE Conference on Computer Communications (INFOCOM'08), 1175-1183, April, 2008.
3. Rik Sarkar, Xianjin Zhu, Jie Gao, Leonidas J. Guibas, Joseph S. B. Mitchell, Iso-Contour Queries and Gradient Routing with Guaranteed Delivery in Sensor Networks. Proc. of the 27th Annual IEEE Conference on Computer Communications (INFOCOM'08), 960-967, April, 2008.
4. Xianjin Zhu, Rik Sarkar, Jie Gao, Shape Segmentation and Applications in Sensor Networks. Proc. of the 26th Annual IEEE Conference on Computer Communications (INFOCOM'07), 1838-1846, May, 2007.
5. Xianjin Zhu, Himanshu Gupta, Fault-Tolerant Manycast to Mobile Destinations in Sensor Networks. Proc. of the IEEE International Conference on Communications (ICC'07), 3596-3603, June, 2007.
6. Zongheng Zhou, Himanshu Gupta, Samir Das, Xianjin Zhu, Slotted Scheduled Tag Access in Multi-Reader RFID Systems. Proc. of the IEEE International Conference on Network Protocols (ICNP'07), 61-70, October, 2007.
7. Rik Sakar, Xianjin Zhu, Jie Gao, Hierarchical Spatial Gossip for Multi-Resolution Representations in Sensor Networks. Proc. of the International Conference on Infomation Processing in Sensor Networks (IPSN'07), 420-429, April, 2007.

8. Rik Sakar, Xianjin Zhu, Jie Gao, Double Rulings for Information Brokerage in Sensor Network. The 12th Annual International Conference on Mobile Computing and Networking (MobiCom'06), 286-297, September, 2006.
9. Xianjin Zhu, Bin Tang, Himanshu Gupta, Delay Efficient Data Gathering in Sensor Networks. The International Conference on Mobile Ad-hoc and Sensor Networks (MSN'05), 380-389, December, 2005.

Chapter 1

Introduction

Wireless sensor networks, combining sensing with communication and computation capabilities, enable a level of fine sensing resolution that was never achieved before. Sensor nodes are able to sample the physical world at a high spatial and temporal resolution and generate a massive amount of data. Thus, data-centric is one of the most important features that make sensor networks distinct from other types of networking systems. How to store, process, transmit and retrieve useful information from massive sensor data to aid diverse applications becomes a fundamental problem in sensor networks.

In this dissertation, we aim to explore the potential of sensor networks as collaborative data processing engines. It is expected that in the near future, the functionalities of sensor networks will go beyond simple data collection as in traditional scientific sensing applications, and will reactively impact the physical world and interact with end users, who live in the same physical domain and query the sensor network anytime anywhere. In that context, it requires that sensor nodes collaboratively process information in the network rather than resorting to a centralized base station for post-processing. We investigate on key challenges of collaborative processing and query evaluation, and propose novel algorithms and protocols to tackle those challenges.

1.1 Sensor Network Overview

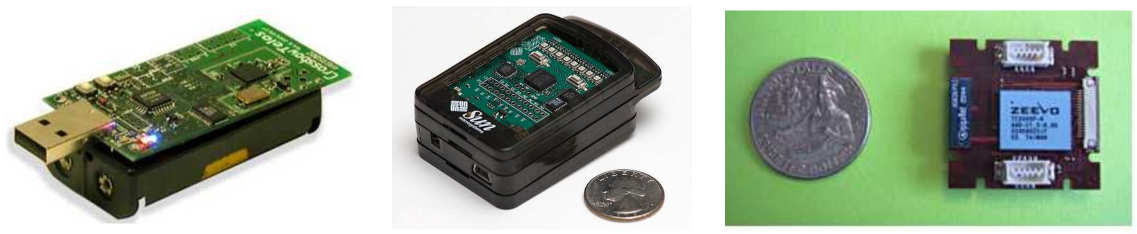
Wireless sensor networks [4–7]are multihop ad hoc networks formed by a large number of resource-constrained sensor nodes. Two sensor nodes can either communicate with each other directly if they are within each other's transmission radius or indirectly using intermediate nodes. The data generated in a sensor network is simply the readings of the sensing devices on the nodes. Devices in sensor networks are usually battery powered and not necessarily rechargeable. Thus, energy efficiency is one of the major concerns in the design of sensor network protocols.

1.1.1 Technology Trend in Sensor Networks

The most typical embedded devices used in current sensor networks are called motes [1–3, 8]. Each mote is equipped with a short-range radio, a low-power CPU, limited processing memory and battery energy, and some sensing devices. Various motes have been developed and put on the shelf. Newer generations of motes will have higher capabilities, including processing capability, memory, etc. while keeping the price low and size small (see Figure 1 and Table 1). The increase of memory (especially flash memories) means that the sensor network with thousands of devices as a whole has huge storage capacity.

With the rapid growth of new embedded hardware and technologies, sensor networks will become heterogeneous and user-friendly. Devices with different capabilities are combined to accomplish sensing tasks in a collaborative manner. More powerful nodes may be intermixed with standard sensing nodes to handle either more demanding sensing modalities (such as images or video) or to provide microserver-class machines with additional local processing and storage. Mobile-handheld devices (e.g., PDA and cell phones) will become important elements of sensor networks and allow end users interact with sensor networks in real-time. In applications such as Nokia's SensorPlanet [9] and Microsoft SensorMap [10], users are able to submit and share data from their own cell phones/PDAs. The integration of sensor networks with handheld or human/vehicle-carried devices also makes mobility one important feature of sensor networks.

The technology trends with increased memory, more powerful processing capability, heterogeneity, mobility and real-time interaction make in-network processing feasible and important.



(i) (ii) (iii)

Figure 1: Various motes. (i) telosB [1]; (ii) SunSpot [2]; (iii) Intel mote prototype [3].

	mica [1]	telosB/tmoteSky [8]	SunSpot [2]
Microcontroller	Atmel Atmega 128L	Texas Instruments MSP430	ARM 920T
RAM (KB)	4	10	512
ROM	128KB	48KB	4MB
Radio	CC1000 916Mhz	CC2420 250kbps 2.4GHz	2.4 GHz 802.15.4

Table 1: Various motes used in wireless sensor networks.

1.1.2 Application Trend in Sensor Networks

The development of wireless sensor networks was originally motivated by military applications such as battlefield surveillance. Its fine granularity sensing capabilities make it widely used in scientific research for dense sampling, environment and habitat monitoring [11–13]. Most of these early sensor network applications ask for simple distributed data collection systems, in which each node samples the environment and sends the signals back to a central base station.

With the advance of hardware and communication technology, sensor networks are enabling novel applications by supplying real-time sensing and situation understanding. It has moved into our daily life, creating a smart environment that interprets and adapts to dynamic situations and aids human activities. A diverse set of applications for sensor networks cross different fields including healthcare, home automation, traffic control, agriculture, environment and many others. The most important feature of these new arising applications is that users of sensor networks are people that stay and roam in the same physical space as the network, rather than scientists operating remotely from the observation site. Embedded users can inject queries into the network anytime anywhere, and expect to receive feedback from the network in real-time. Users and sensor networks are no longer passively connected. They actively interact with each other and impact each other's decisions. The network is not only used to collect data for posterior analysis, but also required to provide real-time actionable information to its users, based on the current state of the world.

1.1.3 Query Engines for Sensor Networks

Most existing query engines regard sensor networks as simple distributed data collection systems and are built upon a many-to-one collection model. Examples of such query engines are TinyDB [14] and Cougar [15]. In those systems, sensor nodes send data towards a centralized base station via a tree-like structure. To minimize energy consumption, it is suggested to perform in-network processing at internal nodes of the tree [15, 16], such as aggregation, filtering, compression, etc.

However, such many-to-one model has fundamental limitations. It scales poorly as sensor networks grow large in size. The centralized servers become bottlenecks and nodes around servers can be depleted much faster than other nodes, which may soonly result in a non-functionable disconnected network. More important, as sensor networks serve a diverse and distributed community of users, the strong dependency on servers hurts the accessibility of sensor data and the interactivity of the network to users.

So, it is worth asking the following fundamental question: "What is a good query engine for a large scale wireless senor network with potential diverse applications?" Observations from technology and application evolution that (1) new technology makes sensor nodes capable of in-network processing and storage; (2) new applications exhibit high variety and require real-time interaction with the network, suggest the following features for a suitable query processing engine for a large scale reactive sensor network:

- **Accessibility:** Enable low-latency access to selective sensor data for end users with diverse interests.

- **Interactivity:** Provide real-time interaction with users, get feedback from users and intelligently react to the physical world.
- **Transparency:** Hide underlying implementation details from users, let users concentrate on high-level functionality of the network.
- **Expressibility:** Support diverse applications with various high-level queries; make information sharable among users.

1.2 Our Approach

We look upon a large scale wireless sensor network as a collaborative integral data processing engine. We propose collaborative information processing and query evaluation techniques based on a logically flat network model, wherein sensor nodes collaboratively extract useful information via local processing and evaluation; users are hidden from low-level implementation details by a deductive programming paradigm and simply specify high-level queries with logic rules. We focus on tackling the fundamental challenges in collaborative processing and query evaluation and explore the potentials of sensor networks as data processing engines with good accessibility, interactivity, transparency and expressibility.

1.2.1 Logically Flat Network Model

The diversity of new applications and the explosive growth of sensor networks motivate us to choose a fairly flat network model for general design instead of the many-to-one collector model. Specifically, in a flat network model, we logically consider all nodes play roughly the same role and have similar computation and communication capabilities. Most nodes at different times perform, at some level, all three of these functions: (1) data acquisition, local processing and event detection, (2) query injection into the network for certain types of data, and (3) information routing and/or aggregation to communicate with other nodes.

The flat network model essentially removes the strong dependency on central servers, thus avoids single point of failures and improves the robustness and scalability. More importantly, it enables seamless real-time interactions with mobile users. On the other hand, it also imposes new challenges on information processing and query evaluation. Without centralized authorities and global coordination, information processing must be performed in network in a distributed and energy-efficient manner. Since each sensor node can only obtain limited local knowledge, it requires intelligent collaborations among thousands of sensor nodes to achieve high-level global objectives through local processing and communications, while keeping in mind the general design criteria such as energy-efficiency and load balance.

Note that we talk about the flat model at the logical data level, regarding it as a virtual abstract of the underlying real deployment. The physical implementation of a sensor network can still be hierarchical and heterogeneous [17]. More powerful nodes can easily simulate the proposed algorithms and act as proxies for sets of mote-level nodes.

1.2.2 Contributions

We investigate on in-network data processing and query evaluation, but our scope goes beyond simple aggregation queries [16, 18, 19]. Under the logically flat network model, we do not rely on any special powerful node or any global knowledge. We push intelligence down to individual sensor node, and let sensor nodes collaborate with each other to achieve global query objectives via local processing.

The theme of our research is to answer the key question of collaborative processing and query evaluation in sensor networks:

"How to extract global information of users' interests from pieces of local knowledge each sensor node has?"

Under this general goal, we address the following specific challenges in this research domain.

1) Link users' interests with data anytime anywhere. There are two correlated subproblems involved. The first question for information processing is how to link users' queries with data, since both queries and related data can appear anytime anywhere in the network. This is one of the central problems of information discovery, dissemination, and brokerage. Besides that, to infer high-level useful information, it is also critical to link multi-dimensional data. In most applications, events of interest are usually implied from the combination of multiple pieces of data collected by different types of sensors, which may be distributed far apart.

We proposed an in-network join scheme which takes a database approach to treat multiple types of sensor values as different database tables and compute join on them based on certain conditions. The join-based approach provides a general way to detect complex event and evaluate high-level queries. This scheme can be further generalized to provide an efficient, load-balanced and locality-sensitive double-ruling scheme for information brokerage.

2) Make implicit information explicit. The diversity of applications poses different requirements and queries on sensor networks. Some applications retrieve more explicit information, e.g. asking for temperature at a particular location. But, there are also other applications that are interested in implicit information, e.g., group behavior of a set of sensors, contours at certain levels, the evolvement of blobs, etc. To retrieve those implicit information, it requires non-trivial techniques.

We proposed a contour tracking algorithm to process such type of implicit level set information. It tracks information of interests and captures the topological changes when the signal field evolves over time. This algorithm provides a foundation for further information processing of spatial sensor data.

3) Decouple network design from specific deployment. The diversity of sensor network deployment comes naturally from the diversity of geographical features of the underlying environment, and has essential influence on network design. In most scenarios, it is infeasible to carefully deploy thousands of sensor nodes in a pre-planned organized way, due to unforeseen obstacles, poorly accessibility, and possible changes in the environment, etc. It is thus desirable to automate the network design process and let the sensor nodes self-organize to a properly functioning network and carry out required tasks.

We proposed to develop a unified approach to handle complex network geometry, in particular, a segmentation algorithm that partitions an irregular sensor field into nicely shaped pieces such that algorithms that assume a uniform and dense sensor distribution can be applied inside each piece. Across the segments, problem dependent structures specify how the segments and data collected in these segments are integrated. This enables the re-use of existing protocols on an irregular network and make the development of new protocols transparent to the specifics of the shape of a sensor field.

4) Provide easy programming paradigm. The ultimate goal of information processing is to return useful information to users, thus it is important to provide friendly interface for users to query/access the network.

We proposed a deductive programming paradigm, wherein the overall collaborative functionality of a sensor network application can be easily represented using deductive (logic) rules. It allows the users to easily specify the high-level functionality of an application, while hiding the low-level details related to distributed computation, resource constraints, energy optimizations, etc. The system translates the high-level specifications to distributed energy-efficient code to run on every individual network node automatically.

Summary. All the above work focus on providing information processing and query evaluation techniques through collaborations among sensor nodes to achieve global objectives (i.e., matching users' interests, reasoning, obtaining topology information) via local processing and communications. The four pieces of work investigate on collaborative processing at different angles, from information brokerage, information-guided network navigation to the impact of network topology and programming paradigm. Each work standalone provides algorithmic foundations for designing future sensor network architecture. All of them together can be packaged into the deductive framework (see Figure 2). In this framework, users' queries can be injected into the network anytime anywhere in the form of logic programs, which will be translated into a set of rules and evaluated by our query evaluation engine (Chapter 5) with the multi-table join algorithm (Section 2.2) in the core. The query evaluation layer sits above a set of network services like network topology discovery using shape segmentation (Chapter 4), signal field topology discovery using contour tracking and contour tree algorithms (Chapter 3) and various routing protocols. The evaluated queries can be further linked with related information and returned to users via an information brokerage scheme like double-ruling (Section 2.3).

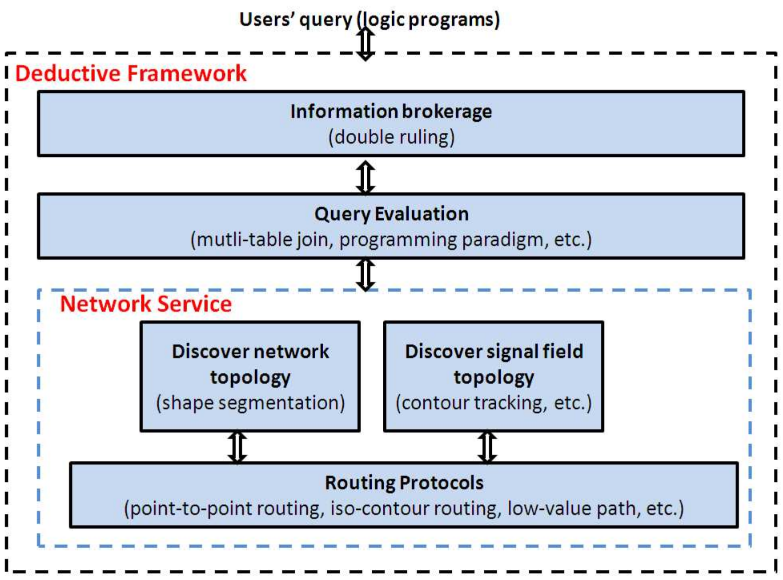

Users' query (logic programs)
Deductive Framework
Information brokerage
(double ruling)
Query Evaluation
(mutli-table join, programming paradigm, etc.)
Network Service
Discover network topology
(shape segmentation)
Discover signal field topology
(contour tracking, etc.)
Routing Protocols
(point-to-point routing, iso-contour routing, low-value path, etc.)

Figure 2: Overview of Dissertation Work.

1.3 Graph Models

Before presenting the main work of this dissertation, I would like to first introduce three graph models used in later chapters.

A sensor network can be modelled as a graph $G = (V,E)$, with V as the set of sensor nodes and there is an edge $e_{ij} \in E$ if node i and node j can directly communicate with each other. The graph can be directed if the links between sensor nodes are asymmetric, i.e., node i can communicate with node j but not true vice versa. Depending on different communication models, the set of edges E can be defined differently.

Unit Disk Graph Model (UDG). In UDG, each node has uniform transmission range of radius 1, two nodes can communicate with each other if and only if their distance is no more than 1. It is showed in [20] that in UDG model, if two edges intersect with each other, there must be a node connecting with all the other three nodes. We used this property in later Chapters to take advantage of broadcast nature of wireless channel.

UDG is simple and useful for theoretical analysis, but does not precisely capture the complex characteristics of radio communication. In reality, the transmission range can become far from a perfect unit disk, especially in a complex environment.

d-Quasi Disk Graph Model (d-QUDG). Quai disk graph model [21, 22] is proposed to represent the irregular radio shape. We use $D(i, j)$ denotes the distance between node i and node j. In d-QUDG:

- if $D(i, j) < d$, edge $e_{ij} \in E$;
- if $D(i, j) > 1$, edge $e_{ij} \notin E$;
- if $d < D(i, j) < 1$, edge e_{ij} exits with certain probability.

Lossy Radio Model. To capture link dynamics, each edge in the graph can be annotated with a weight representing the packet loss probability on this link. Such lossy radio model has been incorporated in the widely used sensor network simulator Tossim [23].

1.4 Organization

In Chapter 2, we present the distributed join and double ruling schemes for information brokerage in senor networks. In Chapter 3, we discuss contour tracking problem to extract implicit contour information. In Chapter 4, we handle underlying complex shaped sensor field with shape segmentation algorithm. In Chapter 5, a deductive framework is proposed to provide easy programming paradigm. We conclude the dissertation in Chapter 6.

Most work presented in this dissertation is done under the supervision of Professor Himanshu Gupta and Professor Jie Gao, and is the product of collaborations with other co-authors. The work on Double Ruling (Section 2.3), Contour Tracking (Chapter 3) and Shape Segmentation (Chapter 4) is joint work with Rik Sarkar. Bin Tang contributed to the implementation of Perpendicular Join Scheme without location information (Section 2.2.5). I would like to thank Xiang Xu for working on building up the test-bed for the Deductive Programming Paradigm work.

Chapter 2

Link Users with Data Anytime Anywhere

2.1 Introduction

Most popular applications of sensor networks, e.g., target tracking, emergency rescue, health-care management, fall into the categories of event detection and real-time sense-and-respond, in which sensor networks provide large-scale intense monitoring over the environment, and/or tracking of interesting targets, as well as delivering the relevant data to the interested parties. Each sensor node typically generates a stream of data items that are readings obtained from the sensing devices on the node. Multiple nodes, with their low-level readings, may collaboratively arrive at a high-level semantic event report, e.g., ‘dangerous contamination’, ‘traffic jam’. Users of the sensor networks may well be embedded in the same physical space and inject queries to the network at any time searching for certain types of data.

These large-scale sense-and-respond applications impose new challenges and requirements on data discovery and delivery protocols.

1. Events of interest may be derived from multiple types of sensor readings, which are spatially/temporarily distributed over the entire sensor network. How to find correlated information from massive data in a distributed efficient way is a challenging problem.
2. On the other hand, data queried by users are often highly selective. Sensor

nodes may detect numerous events of different types at the same time, among which each user is only interested in a much smaller subset. Since queries can be injected into the network at anywhere and events can be detected by any sensor node, how to link users' interests with related events is the other interesting problem we address here.

Matching Multi-dimensional Information. Simple events can be derived from a set of sensor readings of the same type, e.g., abnormally temperature reading higher than a threshold. Complex events may involve multiple different types of data obtained by different sensors, e.g., explosion event is detected only if light, sound, and temperature readings satisfy certain conditions. Since data is spatially distributed at different locations, which can not be predicated in prior, *the first challenge of in-network processing and query evaluation is to find correlated information from multi-dimensional data.*

Most recent works on in-network processing addressed aggregation, selection and compression [14, 16, 18, 19, 24–26] on single type of sensor readings, but few work considered exploring the correlation among multi-dimensional data for high-level event reasoning. The vision of sensor networks as distributed database systems [16, 25, 27] naturally motivates us to consider the *join* operator, which is expressive on presenting complex connections among multi-dimensional data, for general event detection and information matching. Unfortunately, although join has been extensively studied in traditional database systems [28–31], the proposed algorithms typically work in a centralized setting with ample computational resources or in a distributed environment but machines can directly talk to each other and communication cost is not a big issue, there is no much prior work on distributed implementation of join in sensor networks. The difficulty here is due to the fact that each sensor node could generate tuples for any stream table, so a stream table crosses over all nodes in the network and a node does not know where other data (especially the matched data) are. In addition, we must always keep energy-efficiency in mind. Distributed implementation of join must minimize the communication cost incurred, since message communication between sensor nodes is the main consumer of energy [4]. In particular, load-balanced implementation strategies are essential to prolong the network lifetime, because unbalanced strategies are likely to render the network ineffective or inoperable much sooner.

The main contribution of our work is the design of various distributed implementations for join in sensor networks. In particular, we propose the Perpendicular Approach (PA) which is communication-efficient and load-balanced, and in fact, incurs near-optimal (within a constant factor) communication cost for binary joins in grid networks. PA works by using appropriately defined horizontal and vertical paths for data storage and join-computation respectively. The approach is able to efficiently incorporate joins with spatial constraints, and can be generalized to sensor networks without location information [32]. We analyze the communication cost of our approaches, and compare their performance through extensive simulations on *ns2* [33] simulator. We observe that use of PA results in a substantially prolonged network lifetime compared to other approaches. The performance gap is much larger for the more realistic scenario of joins involving spatial constraints.

Matching Information Producer and Consumer. Since both queries and events of interest can appear anytime anywhere in the network, there is a missing link between information consumer and information producer. Users (information consumers) do not know where to find useful data, and sensors detecting certain type of events (information producers) do not know where to deliver those data. Thus, *how to match users' interests with events detected is the second issue we must solve for in-network information processing.* We model the problem as the matching of *information producers*, that perform data acquisition and event detection, with *information consumers* who search for this information. Naturally in a sensor network there can be multiple producers that generate a variety of data types as well as multiple consumers, possibly mobile, that search for relevant information.

Emergency response applications and distributed control systems often impose a high requirement on the access delay at users to ensure event reports or control commands being delivered on time, since information ages fast and stale data is useless. Data queried by users are often highly selective. Furthermore, the arrival of queries may be spatially and temporally distributed. We aim to develop a scheme for large-scale networks that support low-delay queries for multiple users that search selectively for data types discovered and stored in the network.

We proposed a spherical double rulings scheme which stores data replica at a curve instead of one or multiple isolated sensors. The consumer travels along another curve which guarantees to intersect with the producer curve. The double rulings is a natural extension of the flat hashing scheme such as GHTs [34] with improved query locality, i.e., consumers close to producers find the data quickly, and structured aggregate queries, i.e., a consumer following a curve is able to retrieve all the data. Further, by the flexibility of retrieval mechanisms we have better routing robustness and data robustness. We show by simulation that the double rulings scheme provides reduced communication costs and more balanced traffic load on the sensors.

In the following, we first propose efficient schemes for distributed implementation of the join operator in sensor networks. The main contribution of this work is a Perpendicular Approach which bridges relevant information together. Then, we extend the perpendicular approach to a spherical double ruling scheme, and use it to solve the problem of matching information producer and consumer. Note that the Perpendicular Approach is a special case of Double Ruling. Thus, the two level matching problems — multi-dimensional information matching and information producer/cosumer matching are solved in a unified way.

2.2 Join of Multiple Data Streams

In this section, we address efficient and load-balanced in-network implementation of join operator. As mentioned before, the sensor network data corresponding to readings of sensing devices can be modeled as relational data streams. In addition, there may be other data streams in the network corresponding to derived views (such as detected events). Each of the data streams may be generated by an arbitrary set of nodes (perhaps, the entire network), and a node may generate tuples for multiple streams. The node that generates a particular tuple is referred as its *source node*. Each data stream is maintained as a *sliding window* [35–37], by storing only a bounded number of tuples (typically, most recent).

2.2.1 Problem Description and Related Works

Problem Description. Given n data streams $R_1, R_2, R_3, \ldots, R_n$ (not necessarily distinct and a data stream may even correspond to a derived view) in a sensor network, we wish to compute $R_1 \bowtie R_2 \bowtie \ldots R_n$ in a communication-efficient and load-balanced distributed manner. We do not make any assumptions about the join conditions in the join query. However, since the sensor data is highly correlated in the spatial domain [38, 39], we give special consideration to spatial joins (formally defined below) and modify our techniques to efficiently incorporate them. Due to limited memory resources, we constrain a join operation to the join of sliding windows [35–37] of operand streams. Temporal correlation is implicit in the fact that join is computed over the sliding windows. We output the joined result as a data stream across the network just like the operand data streams, for appropriate storage across the network and/or further use as an operand data stream. Since sensor nodes generate streaming data asynchronously and continuously, we evaluate join in an incremental manner, such that the arrival of a new tuple will only generate new results associated with that tuple.

Definition 1 ***Spatial Join.*** *A join between two data streams R_i and R_j is said to be a* spatial join *of range s if the join condition is a* conjunction *of ($|R_i.nodeLocation - R_j.nodeLocation| \leq s$) and other arbitrary predicates. Here,* nodeLocation *is the attribute for the location of the tuple's source node, and $|x - y|$ is the distance between x and y.*

Performance Criteria. Our main performance criteria of a join implementation is the resulting *network lifetime*. In general, network lifetime is defined as the time after which the network is rendered "useless" (ineffective or inoperable) or disconnected, due to failure of enough nodes. However, the precise definition of the network lifetime depends on the specific objective of an application. In either case, the network lifetime is prolonged by conserving overall battery energy and uniform depletion of battery resources across the network. The former is achieved by minimizing communication cost [4] and the latter by a load-balanced implementation. Thus, we focus on design of communication-efficient and load-balanced implementations. In our simulations, we define network lifetime in terms of the approximation ratio of the obtained join results.

Motivation. One of the strong motivations for distributed implementation of join in sensor networks is that the join operation forms the core of (bottom-up) evaluation of deductive rules, as shown in our concurrent work [40]. The above is particularly significant due to the recent interest and suggestion of deductive programming as an appropriate vehicle for declarative programming of sensor networks [40,41] and in general, for declarative networking [42, 43]. For instance, [40, 41] shows that typical sensor network applications such as shortest path tree, vehicle trajectories, localization, etc. can be expressed as simple deductive programs.

Another specific motivation for join implementation is event detection, one of the most prominent applications of sensor networks. An *event* indicates a point in time of interest based on certain conditions over the sensor data. In certain cases, events may simply depend on the local value of a sensor reading. Higher-level events or complex events may be specified using composition operators over the primitive events. In particular, the complex events may be represented as a join, involving spatial and temporal constraints, as illustrated below.

Motivating Example 1. Consider a sensor network deployed in an underground mine to detect explosions. Let us assume that the event of an explosion is characterized by interaction between three phenomena/events viz., sound, light, and temperature, and each phenomenon is detected by respective sensors. A temperature event is said to occur when the temperature sensed at any sensor node reaches (or increases) by a certain threshold. Light and sound events are similarly defined. Each of these events is detected locally, and stored in the respective tables along with the locally computed *duration* of the event.

The *explosion event* is defined to occur when the following conditions are satisfied [44]. (i) The light, sound, and temperature events occur within 10 meters of each other, (ii) The ratio of the durations of sound and light events is at least *c* (some constant depending on the speeds of sound and light), and (iii) The duration of the temperature event is at least 60 seconds. The query that can be run in the network to detect the above explosion event is as follows.

```
SELECT *, event as "EXPLOSION"
FROM Sound, Light, Temperature
WHERE |Sound.location-Light.location| < 10
    AND |Light.location-Temperature.location| < 10
    AND |Sound.location-Temperature.location| < 10
    AND Sound.duration > 60
    AND Sound.duration/Light.duration > c
    AND Temperature.duration > 60
```

The above query may result in an `Explosion` event stream being generated in the network. Note that above every pair of streams has a spatial-join of range 10, and the temporal correlation is implicit in the maintenance of sliding windows.

Motivating Example 2. Consider a sensor network deployed for tracking moving vehicles. Each sensor has some means (possibly, vibration or magnet sensors) of detecting presence of a vehicle in the proximity. Consider the event: A vehicle surrounded (from all four directions) by four other vehicles [45]. If detection of a vehicle by a node results in generation of a corresponding record in a global table T, then detection of the above event requires a 5-way self-join of the table T using an appropriately defined *surrounded* predicate over five `nodeLocation` arguments.

Motivating Example 3. Consider a more complex event defined over the `Explosion` event stream of Example 1: A vehicle surrounded (from all four directions) by four explosion events within a certain time window. Here, the location of an explosion event can be defined as the centroid of the sound, temperature, and light event locations. This example illustrate the use of a derived view stream in defining a involved event query.

Related Works. The vision of sensor network as a database has been proposed by many works [15, 27, 46], and simple query engines such as TinyDB [25] have been built for sensor networks. However, prior research has only addressed limited SQL functionality - single queries involving simple aggregations [14, 16] and/or selections [25] over single tables [47], or local joins [16]. So far, it has been considered that correlations such as median computation or joins should be computed on a single node [16, 25, 48]. In a recent work [49], authors consider a combination of localized and centralized implementation for a join operation wherein one of the operands is a relatively small static table which is used to flood the network. However, the problem of distributed implementation for general join operation has not been addressed yet in the context of sensor network, except in below described recent works [50, 51].

In [50, 51], authors addressed in-network implementation of join over two tables assuming a fixed query source. In particular, in [50], they addressed in-network implementation of the general join operation of two relational tables, and presented an optimal algorithm which incurs provably optimal communication cost under reasonable assumptions. In [51], they considered in-network implementation of joins with range predicates, and developed communication-efficient hash-join and index-join implementations. The problem addressed in this work is more general than the above works in that in our work, we consider join of multiple (two or more) data streams. More importantly, the focus of this work is the development of communication-efficient *and* load-balanced implementations in networks without a fixed/static query source. Note that hashing and indexing techniques such as that used in [51] are not directly applicable for join of more than two data streams, unless we restrict ourselves to very specialized join conditions.

The similar idea as our approach is also used in the literature of information dissemination and retrieval [52–55], wherein queries are interested in a certain type of data. However, all above works restrict to nice formed networks (grid network with direct communication between any pairs of machines [53], random grid with constraint geographical flooding [54] or uniformly densely deployed senor network [55]) rather than a general network. In addition, the problem of information brokerage [55] has different criteria from this multi-dimensional information matching, which leads to specific design and implementations. However, our treatments of these two problems indeed have generic principal and similar core ideas. We will discuss the information brokerage problem in Section 2.3.

In addition to the work done in the context of sensor network databases, there is a large body of work done on efficient query processing in data stream processing systems [56–59]. In particular, [60] approximates sliding window joins over data stream and [61] has designed join algorithms for joining multiple data streams constrained by sliding windows. However, a data stream processing system is not necessarily distributed, and hence, minimizing communication cost is not the focus of the research.

There has also been a lot of work on multi-join query in parallel database systems [28–31]. In such systems, there are powerful communication facilities for passing messages among processors, so communication cost is not an issue. The focus of parallel join is on how to allocate processors so that the total execution time of a set of joins can be minimized. Sensor networks differ significantly from parallel database systems because of severe resource limitations and multihop communication-cost model in sensor networks. Thus communication efficiency is a major concern in sensor networks.

2.2.2 Naive Broadcast and Centroid Approaches

In this section, we present a couple of simple approaches, viz., Naive Broadcast and Centroid (CA), for distributed implementation of $R_1 \bowtie R_2 \bowtie \ldots R_n$ where each R_i is a data stream in the network. Here, we consider only in-network implementations since routing all sensor data to a central server would incur prohibitive communication costs [4, 62]. However, CA is essentially an optimized in-network centralized approach.

Naive Broadcast Approach (BA). The simplest way to implement a join of multiple data streams is to broadcast each generated tuple to the entire network, and store all the sliding windows at each node. Then, the join can be computed locally at any network node. In case of spatial joins, a tuple of R_i needs to be broadcast only within a region of radius $\max_j s_{ij}$, where s_{ij} is the range of the spatial-join between R_i and R_j. Note that a non-spatial (or non-existent) join is a spatial-join of infinite range. This approach is infeasible in most cases due to severe memory constraints in sensor networks. The other way to do naive broadcast is to store tuples locally but compute join by broadcasting tuples within a certain region. In the following, we refer to this approach as BA. In BA, all partial results need to be broadcast again until a final result is generated. In the case of join without spatial constraints, it requires flooding the entire network a lot of times.

Centroid Approach (CA). CA works by first choosing appropriate *storage regions* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ in the network for storing the sliding windows for the streams $R_1, R_2, \ldots, R_n$ respectively. To facilitate efficient computation of join, the regions $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ (not necessarily different) are all located close to each other. Each generated tuple t of each stream R_i is first routed from its source node to its storage region $\mathcal{C}_i$, where it is stored at some node with available memory (see below for details). Thereafter, the tuple t and the resulting intermediate tuples are routed through the regions $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{i-1}, \mathcal{C}_{i+1}, \ldots, \mathcal{C}_n$ (in some order) to compute the join result.

Storage Regions, Routing, and Storage. Let ρ_i be the rate of generation of tuples of data stream R_i, and let $\mathcal{R}_i$ be the set of nodes (possibly, the entire network) generating the tuples of R_i. Let us define the centroid $\mathcal{C}$ as the location in the network that minimizes the value $\sum_{i=1}^{n} \rho_i \overline{d}(\mathcal{R}_i, \mathcal{C})$, where $\overline{d}(\mathcal{R}_i, \mathcal{C})$ is the average number of hops between a node in $\mathcal{R}_i$ and $\mathcal{C}$. Now, it can be shown (we skip the simple proof here) that choosing the storage regions closely around the centroid $\mathcal{C}$ minimizes the total communication cost (number of hops traversed) of CA for dense networks. However, nodes around the centroid $\mathcal{C}$ are congested and could be depleted faster than other nodes. When sufficient nodes in the storage region fail, a new storage region is selected and all nodes informed.

The location of the storage regions can be either broadcast to the entire network initially or maintained at the node closest to the network center. The above allows each generated tuple to be routed to the required storage regions using geographical routing. In sensor networks without location information, we need to construct and maintain routing paths from each node to the storage regions. In either case, when a new arrival tuple t of stream R_i reaches the node I closest to the center of $\mathcal{C}_i$, the node I searches for a close-by node in $\mathcal{C}_i$ with available memory. Such a node can be found by broadcasting an appropriate request message to nodes in $\mathcal{C}_i$, gathering responses from nodes that have available memory, and picking the closest node among them. After storage, the tuple t (along with other intermediate results) is routed to other storage regions (in some order) for computation of join.

2.2.3 Perpendicular Approach (PA) in Grid Network

In the following, we describe the Perpendicular Approach (PA), which is load-balanced and communication-efficient. We start by describing PA in grid networks, and then, generalize it to general topologies with and without location information.

Definition 2 ***2D Grid Network.*** *A two-dimensional (2D)* $m \times m$ grid network *is a network formed by placing a node of unit transmission radius at each location* (p,q) *(*$1 \leq p \leq m$ *and* $1 \leq q \leq m$*) in a two-dimensional coordinate system.*

Join of Two Streams in Grid Networks. In a 2D coordinate space, every horizontal line (i.e., a line parallel to x-axis) intersects every vertical line (i.e., a line parallel to y-axis). Thus, if each generated tuple is stored at all nodes of some horizontal line, then the set of nodes on any vertical line will collectively contain all sliding windows. Here, we arbitrarily choose horizontal lines for storage and vertical lines for join-computation; however, their roles can be easily swapped.

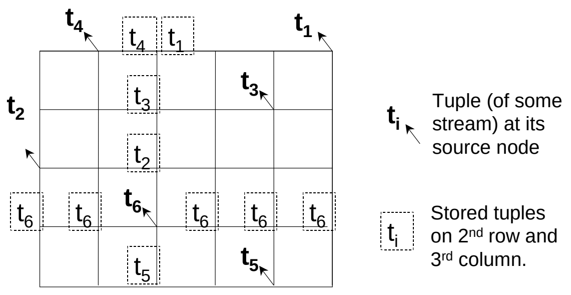

t4
t4
t1
t1
t3
t2
t3
t2
t6
t6
t6
t6
t6
t6
t6
t5
t5
ti
Tuple (of some stream) at its source node
ti
Stored tuples on 2nd row and 3rd column.

Figure 3: Storage/Replication of PA in 2D Grid Networks.

Based on the above observation, PA consists of *two phases*, viz., storage and join-computation. Consider a grid network with data streams R_1 and R_2, and a tuple t (of either data stream) generated at coordinates (p,q).

- *Storage Phase*: In the storage phase, the tuple t is stored (replicated) along the q^{th} *horizontal line*, i.e., at all nodes whose y-coordinate is q. This ensures that set of nodes on *each* vertical line collectively contain the entire sliding windows for R_1 and R_2. See Figure 3.
- *Join-computation Phase*: In the join-computation phase, we route t along the p^{th} vertical line to compute the result tuples due to t (i.e., $t \bowtie R_2$ or $t \bowtie R_1$ depending of whether t is in R_1 or R_2). The result tuples are computed by locally joining t with matching tuples of R_1 or R_2 stored at nodes on the p^{th} vertical line.

To maintain time-based sliding windows [35], each stored tuple t is kept in the local memory of node I until $\tau_w + \tau_s + \tau_j$ time after its arrival at I, where τ_w is the interval of the sliding window, and τ_s and τ_j are the upper bounds on the time to complete the storage and join-computation phases respectively. To correctly handle simultaneously generated tuples across the network, we start the join-computation phase for a tuple only after the completion of its storage phase. Thus, we introduce a delay of τ_s between the start of two phases. The correctness of above approach follows from the more general claim in Theorem 5.

Spatial Joins. If $R_1 \bowtie R_2$ is a spatial-join of range s, then a tuple t (of R_1 or R_2) generated at (p,q) is stored at only those nodes on the q^{th} horizontal line that are within a range of s from (p,q). Similarly, in the join-computation phase, the tuple t is routed only to nodes within a range of s from (p,q) on the p^{th} vertical line.

Communication Cost within Constant Factor of Optimal. We now show that PA incurs communication cost within a constant factor of optimal (in addition to being perfectly load-balanced) for uniformly generated data streams in a grid network.

Theorem 3 *Consider an $m \times m$ grid network with the data streams R_1 and R_2 being generated uniformly over the entire network. The total communication cost (total number of tuple-hops traversed) incurred by PA to compute the join of R_1 and R_2 is at most eight times the minimum communication cost needed.*

Proof. For simplicity, we assume a general (non-spatial) join; however, the proof easily generalizes to spatial joins. First, it is straightforward to see that the total communication cost incurred by PA is at most $2m$ units for each generated tuple.

Note that in an $m \times m$ grid network, there are at least $m^2/2$ disjoint pairs (I_1, I_2) of nodes such that the distance (in hops) between I_1 and I_2 is $m/2$. The above is true since there are at least $m/2$ such disjoint pairs of nodes on each vertical and horizontal line, and there are m disjoint vertical or horizontal lines. Now, consider a pair of nodes (I_1, I_2) in the above set of disjoint pairs. Each tuple t of R_1 generated by I_1 must join with each tuple t' of R_2 generated by nodes I_2. Since, the distance between I_1 and I_2 is m/2, the communication cost incurred in joining t and t' is at least $m/2$. Thus, if each network node generates one tuple each for R_1 and R_2, then the minimum communication cost incurred in computing the join of these $2m^2$ generated tuples is at least $(m/2)(m^2/2)2 = m^3/2$. If the generation of operand tuple is uniform, the minimum communication cost required is $m/4$ per generated tuple, which is one-eighth of that incurred by PA. $\square$

Multiple Streams in Grid Networks. Consider a grid network with data streams $R_1, R_2, \ldots, R_n$. PA can be generalized to handle more than two data streams as follows. First, the storage strategy remains the same as before, i.e., each tuple t generated at (p, q) is still stored along the q^{th} horizontal line. However, in the join-computation phase, we need to traverse the vertical line in a more involved manner. Below we describe two schemes, viz., one-pass and multiple-pass, for traversing the vertical line in the join-computation phase. We start with a definition.

Definition 4 ***Partial Result.*** *Let $R_1, R_2, \ldots, R_n$ be given data streams and let t be a tuple of R_j. A tuple T is called a* partial result *(of $R_1 \bowtie R_2 \ldots \bowtie R_n$) for t if*

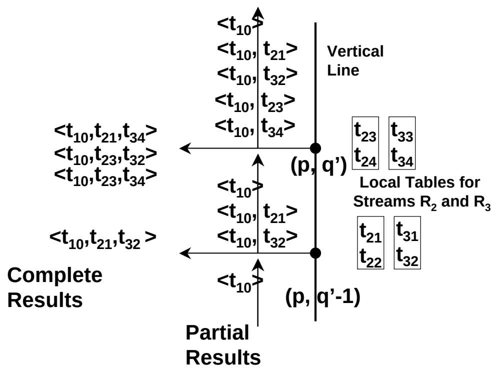


Figure 4: One-Pass Join Computation. Here, $t_{i*} \in R_i$, and we assume the join conditions to be such that t_{10} matches only with $t_{21}, t_{23}, t_{32}, t_{34}$. Also, there is no join condition between R_2 and R_3.

$T \in (t \bowtie R_{i_1} \bowtie R_{i_2} \ldots R_{i_k})$ where $k < n-1$ and $i_l \neq j$ for any l. If $k = 0$, then $T = t$, which is also considered a partial result. If $k = n-1$, then T is called a complete result.

One-Pass Scheme. Consider a tuple t (of some data stream) generated at a node (p,q). In the one-pass scheme, the tuple t is first unicast to one end (i.e., $(p,0)$), and then, is propagated through all the nodes on the p^{th} vertical line by routing it to the other end. At each intermediate node (p,q'), certain partial and complete results (as defined above) are created by joining the incoming partial results from $(p,q'-1)$ with the operand tuples stored at (p,q'). The computed partial results along with the incoming partial results are all forwarded to the next node $(p,q'+1)$. See Figure 4. Certain incoming tuples may join with the operand tuples stored at (p,q') to yield complete results, which are then output and not forwarded. The partial results generated at the last node (other end) are discarded.

Theorem 5 *Given data streams $R_1, R_2, \ldots, R_n$ in a sensor network, the Perpendicular Approach (with one-pass join-computation scheme) correctly computes $R_1 \bowtie R_2 \bowtie \ldots R_n$, in response to distributed (and possibly, simultaneous) generation of tuples. We assume bounded τ_s and τ_j, the time for completion of storage and join-computation phases respectively, and no message losses.*

Proof. By description of the scheme and the definition of complete results, the one-pass scheme outputs only those tuples that belong to the final join result. To show that every tuple of final join result is eventually output, consider an arbitrary tuple T in the final result. Let T be the result of matching of $\{t_1, t_2, \ldots, t_n\}$, a set of n tuples $t_i \in R_i$ one from each data stream. Let t_l (for some $l \leq n$) be the tuple among t_i's whose storage phase was completed the last. Now, we claim that the tuple T must be output during the one-pass join-computation phase of t_l. Let t_l be generated at node (p,q). When the join-computation phase of t_l starts, the storage phase of each t_i ($1 \leq i \leq n$) has been completed by definition of l and the fact that the join-computation phase of t_l starts after the completion of its storage phase. Thus, during the join-computation phase of t_l, each of the tuples t_i is available at some node on the p^{th} vertical line,[1] and the tuple t_l encounters (in some arbitrary order) each one of these t_i tuples. Thus, the tuple T is eventually output. □

Multiple-Pass Scheme. In the multiple-pass scheme, the join-computation phase takes place in a certain order of data streams. Each iteration of the multiple-pass scheme is essentially a one-pass scheme involving join of a data stream with partial results generated in the previous iteration. More formally, let the predetermined join-ordering of data streams be $R_{i_1}, R_{i_2}, \ldots, R_{i_{n-1}}$ (not including the stream of the new tuple t). In the first iteration, the tuple t is propagated through the vertical line (from one end to another) to join with R_{i_1}. In general, in the k^{th} iteration, the partial results obtained from the previous $(k-1)^{th}$ iteration are propagated through the vertical line to join with R_{i_k}. Thus, the partial results generated in the k^{th} iteration constitute $t \bowtie R_{i_1} \bowtie \ldots R_{i_k}$.

2.2.4 Perpendicular Approach in General Networks

We now generalize the PA approach to general network topologies. The main challenge in generalizing PA is to define appropriate notions of horizontal and vertical paths such that each horizontal path intersects each vertical path. In addition, we do not want to involve too many nodes in a particular path, since that would waste scarce resources in network. Due to network topology being dynamic and limited resources at each node, we do not wish to *construct* and maintain generic horizontal

[1] Availability of t_i at a node follows from the fact that t_i and t_l matched to form T (and hence, must have been generated with τ_w of each other), and a tuple t_i expires at node I only after $\tau_w + \tau_s + \tau_j$ time of its arrival at I. Here, τ_w is the interval of the time-based sliding window.

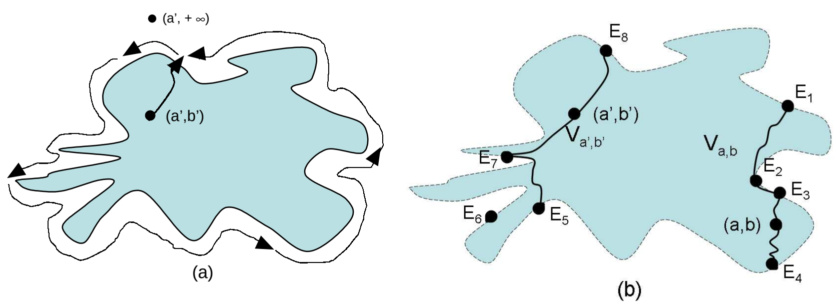


Figure 5: Illustration of vertical paths in an arbitrary sensor network topology, with and without markings. (a) The path from source node (a',b') to destination $(a',Y_{amx}+1)$ constructed by GPSR. Since the destination is out of the network field, after reaching the boundary, the path will travel the entire boundary until it returns to the same node and there are no other faces to go around. (b) Vertical paths for nodes (a,b) and (a',b'). Here, the markings on the given boundary nodes is as follows: E_1 and E_8 are marked `Highest`, E_4 and E_5 are marked `Lowest`, and the rest are marked `Middle`. So the vertical path $V_{a',b'}$ stops at node E_8 and E_5.

and vertical paths for each network node. Therefore, we build paths on-the-fly. Ideally, we would like the horizontal and vertical paths to be the set of nodes encountered during routing to source-node-specific destinations. Moreover, we would like the communication and data replication costs to be near-perfectly balanced across the network. The presence of topological holes and arbitrary boundaries of the network make the implementation in general topologies particularly challenging. Keeping the above challenges in mind, we first introduce location-based routing in the context of path construction, then we define the vertical and horizontal paths respectively.

Location-Based Routing. In sensor networks, nodes are typically referred to by their geographic locations (instead of IDs), and each node is aware of its location (using GPS or localization techniques [63]). Thus, in this section, we consider sensor networks with location information, and design techniques which use location-based routing (described below) for routing between nodes/locations. We do generalize our techniques to networks without location information in Section 2.2.5.

In *location-based routing* protocols, the destination is specified by its geographic location. Due to severe memory constraints, the location-based routing protocols in sensor network are reactive (on-demand), and determine the next hop on the fly. One simple location-based routing protocol is the greedy approach [64] wherein each node forwards the packet to the neighbor closest to the destination.

However, greedy approach can get *stuck* at nodes that have no neighbor closer to the destination than itself. In contrast, face-routing [64] protocol routes the packet through a sequence of faces (in an extracted planar subgraph of the network) that intersect the line segment connecting the source and the destination. For efficiency, face-routing is combined with the greedy approach – yielding the well-known GPSR [64] protocol.

Vertical Paths in General Networks. For a node at a location (p,q) in the network, we denote its *vertical path* as $V_{p,q}$ and define it as the concatenation of the paths traversed (or set of nodes encountered) when a packet is routed using GPSR protocol from (p,q) to $(p,Y_{min}-1)$ and from (p,q) to $(p,Y_{max}+1)$. Here, Y_{max} and Y_{min} are the largest and smallest y-coordinate values in the entire network. Since there is no node at location $(p,Y_{max}+1)$ or $(p,Y_{min}-1)$, $V_{p,q}$ includes the entire network boundary (since the external face intersects the line segment connecting the source and destination) (see Figure 5(a)). In general, the boundary nodes are part of every vertical path. To choose minimal segments on the boundary, we mark the boundary nodes and modify the GPSR protocol as described below.

Definition 6 ***Markings on Boundary Nodes.*** *A node (p,q) on the boundary is marked* `Highest` *(*`Lowest`*) if there is no network node I such that I has an edge intersecting the line $x=p$ and has a y-coordinate greater (less) than q. Otherwise, the node (p,q) is marked* `Middle`*. See Figure 5 (b).*

Redefining Vertical Paths. Based on the above markings, we construct a vertical path $V_{p,q}$ such that: (i) It is a continuous path connecting *some* `Highest` node to *some* `Lowest` node; this is to ensure intersection with every horizontal path (defined later). (ii) It deviates as little as possible from $x=p$; this is to efficiently incorporate spatial joins (as discussed later). (iii) It includes as few nodes as possible, and is different for different (p,q); this is for efficiency and load-balance. Keeping in mind the above considerations, we still define $V_{p,q}$ as the set of nodes traversed when routing (p,q) to $(p,Y_{max}+1)$ and $(p,Y_{min}-1)$, but modify the behavior of GPSR on the boundary as follows.

On way to $(p,Y_{max}+1)$, if GPSR reaches a boundary node I: If I is marked `Highest`, then GPSR stops and the vertical path is completed; else GPSR is directed to (i) a non-boundary neighbor of I with a higher y-coordinate than I, or (ii) (if

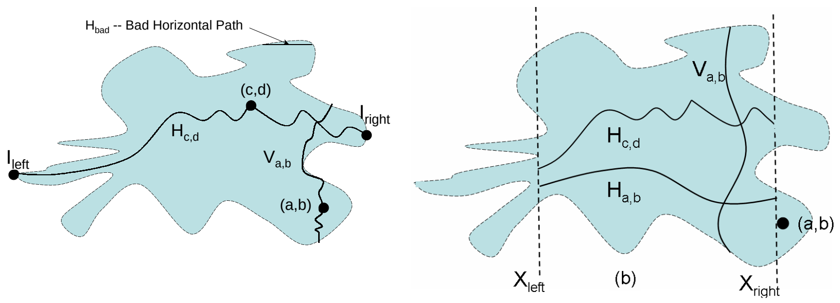


Figure 6: Illustration of horizontal paths in an arbitrary sensor network topology. (a)Horizontal path for node (c,d). Here, I_{left} and I_{right} are the leftmost and rightmost nodes in the network. The path H_{bad} shows that defining horizontal paths in a similar way as vertical paths does not ensure intersection of vertical and horizontal paths. (b) Ears of the network is cut to solve congestion around I_{left} and I_{right}. Horizontal paths $H(a,b)$ and $H(c,d)$ stop when reaching lines X_{left} and X_{right}.

no such non-boundary neighbor exists) the left or right boundary neighbor of I whichever is on the shorter path to some node marked `Highest`. For instance, in Figure 5 (a), on the path from (a,b) to $(a,Y_{max}+1)$, at E_3 GPSR is directed to the left boundary neighbor E_2, while at E_2 GPSR is directed to a non-boundary neighbor C. GPSR is similarly modified on reaching a boundary node on way to $(p,Y_{min}-1)$. The boundary-node markings and information (left or right boundary neighbor for directing GPSR) can be computed periodically in a centralized manner at some node.

Definition 7 ***Vertical Path.*** *Vertical path $V_{p,q}$ for a node at a location (p,q) is defined as the concatenation of the paths connecting* some `Highest` *node to* some `Lowest` *node (or set of nodes encountered) when a packet is routed using GPSR protocol from (p,q) to $(p,Y_{min}-1)$ and from (p,q) to $(p,Y_{max}+1)$. Here, Y_{max} and Y_{min} are the largest and smallest y-coordinate values in the entire network.*

Horizontal Paths in General Networks. Defining horizontal paths precisely in the same manner as vertical paths will not ensure intersection of a horizontal path with each vertical path. For instance, see H_{bad} and $V_{(a,b)}$ in Figure 6 (a). Hence, we define horizontal paths as in Definition 8. Theorem 9 proves pairwise intersection of such defined vertical and horizontal paths.

Definition 8 ***Horizontal Path.*** *Let I_{left} and I_{right} be the leftmost (i.e., the node with the smallest x-coordinate) and rightmost nodes respectively in the entire network.*

We denote the horizontal path *for a node at location (p,q) as $H_{p,q}$ and define it as the concatenation of paths traversed by the GPSR protocol when a packet is routed from (p,q) to I_{right} and from (p,q) to I_{left}.*

Theorem 9 *Consider two arbitrary nodes (p,q) and (r,s) in a connected sensor network. The paths $V_{p,q}$ and $H_{r,s}$ intersect, i.e, the paths contain a common node or a pair of edges (one from each path) that cross each other.*

Proof. By definition $V_{p,q}$ is a continuous path connecting a pair of nodes $H = (h_x, h_y)$ to $L = (l_x, l_y)$ such that H is marked `Highest` and L is marked `Lowest`. It can be shown that H and L lie on different sides of $H_{r,s}$(or one of them lies *on* $H_{r,s}$), since $H_{r,s}$ is a continuous path from the leftmost node I_{left} to the rightmost node I_{right} in the network. Thus, $V_{p,q}$ intersects $H_{r,s}$. □

Overall Approach. Using the above notions of horizontal and vertical paths, overall PA works as follows. Each tuple t (of any data stream) generated at a node (p,q) is first stored on every node of $H_{p,q}$. In the join-computation phase, the tuple t is routed through all the nodes on $V_{p,q}$. Note that due to the wireless broadcast advantage, the tuple t is automatically received at at least one node of each edge that crosses $V_{p,q}$.[2] The above observation allows us to actually reduce storage cost by storing each tuple only on a subset of (instead of every) nodes on the horizontal path during the storage process, without affecting the correctness. We can use either one-pass or multiple-pass scheme for the join-computation phase.

Storage vs. Communication Cost Tradeoff. Instead of storing tuples at every nodes on the horizontal path, we can store tuples at every other node. More generally, to reduce data replication further, we could store a tuple on every k^{th} node on the horizontal path for an appropriately chosen parameter k (Figure 7). If $k \geq 2$, we need to broadcast the tuple t to the $\lfloor k/2 \rfloor + 1$-hop neighborhood of every node on the vertical path during the join-computation phase, to ensure correctness.

Theorem 10 *If tuples are stored at every kth node on the horizontal path, the new arrival tuple needs to be broadcast $\lfloor k/2 \rfloor + 1$-hop neighborhood of every node on the vertical path during the join-computation phase, to ensure correctness.*

[2]This is due to the property of unit-disk graph model stated in Section 1.3.

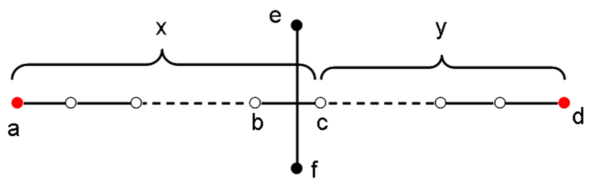


Figure 7: Reduce storage by storing tuples every kth node. Tuples are stored at red nodes but not hollow nodes. The worst case happen when the vertical path intersects with the $(\lfloor k/2 \rfloor + 1)$-th edge of the horizontal path.

Proof. Suppose the intersected edges are E_{ef} on the vertical path and E_{bc} on the horizontal path P_{ad}, and only node a and d store a particular tuple (see Figure 7). The farthest distance from e (or f) to (a or d) is $\lfloor k/2 \rfloor + 1$, when E_{bc} is the $(\lfloor k/2 \rfloor + 1)$-th edge of the horizontal path. So $\lfloor k/2 \rfloor + 1$ is sufficient to meet a tuple. $\square$

Dynamic Topologies and Fault Tolerance. PA is immune to changes in the topologies (node failures or additions), since the vertical and horizontal paths are constructed dynamically (on the fly) using the GPSR protocol. In addition, the markings on the boundary nodes need not be up to date for correctness, because if GPSR reaches a node with no markings (due to obsolete information) we can always resort to normal GPSR. Finally, the approach is inherently fault-tolerant to node/link failures, since in an irregular topology a vertical path $V_{p,q}$ is likely to intersect a horizontal path $H_{r,s}$ at multiple nodes/edges. Essentially, the join result is likely to contain duplicate result tuples due to the above phenomenon – making the approach fault-tolerant.

Traffic Congestion Problem. The horizontal paths may result traffic congestion in the region around the I_{left} and I_{right} nodes.[3] We can solve the above congestion problem by excluding the "ears" of the network (Figure 6(b)). In particular, we periodically determine two values X_{left} and X_{right} such that (i) the number of nodes between X_{right} and X_{left} is large, and (ii) the number of nodes with an edge intersecting with each of the lines $x = X_{right}$ and $x = X_{left}$ is also large. For a given node (p,q), if $X_{right} \le p$ or $p \le X_{left}$ (i.e., if (p,q) is in the ear), then we choose a node (p',q') such that $X_{left} \le p' \le X_{right}$. Else, let $(p',q') = (p,q)$. Now, we define the horizontal path $H_{p,q}$ as the concatenation of the following paths: (i) Path traversed

[3]It is for this reason that we defined vertical paths differently than horizontal paths. Otherwise, defining vertical paths in a manner similar to horizontal paths does ensure correctness (intersection).

when routing from (p',q') to (X_{left},q') until $x = X_{left}$ is reached, and (ii) Path traversed when routing from (p',q') to (X_{right},q') until $x = X_{right}$ is reached. However, with above definition of a horizontal path, intersection of paths (Theorem 9) cannot be guaranteed for the above notion of horizontal/vertical paths. Nevertheless, in our simulations over random dense networks, we observed that as per the above definitions, each horizontal path still intersected with each vertical path.

Extensions. The proposed PA can be easily incorporated with spatial join, which is one of the most common cases in sensor networks. We further discuss range-join and other complex queries combing joins with selections and aggregations.

Incorporating Spatial Joins. For spatial joins, we need to store and propagate each tuple along only parts of the paths. For a data stream R_i, let $s_i = \max_j s_{ij}$ where s_{ij} is the range of the spatial-join between R_i and R_j. Note that a non-spatial join is a spatial-join of infinite range. Now, let d_x be such that the x-coordinate of any node on any vertical path $V_{p,q}$ is most $(p+d_x)$ and at least $(p-d_x)$. In other words, d_x is the maximum *deviation* of any vertical path from its vertical. Similarly, let d_y be the maximum deviation of any horizontal path from its horizontal line. Then, for spatial joins, the vertical path $V_{p,q}$ used for a tuple of R_i at (p,q) is the concatenation of paths traversed when routing from (p,q) to $(p,q+s_i+d_y)$ and from (p,q) to $(p,q-s_i-d_y)$. Similarly, the horizontal path $H_{p,q}$ used is the concatenation of paths traversed when routing from (p,q) to $(p+s_i+d_x,q)$ and (p,q) to $(p-s_i-d_x,q)$.

Note that the value d_x (d_y) used for $H_{p,q}$ ($V_{p,q}$) above should only be such that the deviation of a vertical (horizontal) path of any node within a range of s_i from (p,q) is at most d_x (d_y). Thus, for $H_{p,q}$ or $V_{p,q}$, we need to care about path-deviations of only those nodes that are within a range of s_i from (p,q). Thus, the values d_x and d_y depend only on the local network density, and hence, can be gathered periodically from nodes within a range of s_i. Such gathering of deviations can be achieved by each node broadcasting its path-deviations to nodes within a range of s_i.

Incorporating Range Joins. Range-joins (Definition 11) can be regarded as a generalization of spatial-join and converted into spatial join by hashing a joined attribute to a geographical location. The hash function we choose must satisfy locality sensitive property [65]. A hash function $h(u) = (x,y)$ satisfies locality sensitive property, if $|u-v| < s$ then $d(h(u),h(v)) < g(s)$ where u and v are the attribute values, d is the distance function, and g is some function. Based on the above property, we can map a tuple with attribute value u to location $h(u)$. For a data stream R_i, let $s_i = \max_j s_{ij}$ where s_{ij} is the range on attributes of the range-join between R_i and R_j, we use $g(s_i)$ as the range of the spatial constraint and replicate data (compute join) only on a segment of the horizontal (vertical) path as the way in the spatial join.

Definition 11 ***Range Join*** *A join between two data streams R_i and R_j is said to be a* range join *of range s if the join condition is a* conjunction *of ($|R_i.att - R_j.att| \leq s$) and other arbitrary predicates. Here,* att *is the join-attribute for the operand tables.*

<u>Combining with Selections/Aggregations.</u> It is straightforward to combine our join scheme with other query schemes, since the derived join results can be treated in the same way as sensed streaming data and regarded as inputs for other operators. To optimize communication cost, we can generate query evaluation plans that push down selection/aggragtions, so that tuples are filtered/aggregated before join. Essentially, it reduces the generating rate of operand streams of the join operator, and would not affect other parts of our join schemes.

2.2.5 Perpendicular Approach Without Location Information

Our PA is built on top of location-based routing protocol, and hence, assumes that each node is aware of its geographic location. However, in certain applications, location information is either not accurate enough or not even available. For such sensor networks, we define perpendicular *regions*, viz., k-dominating set (defined below) and k-hop neighborhood for some carefully chosen k, for each node I, and use them for storage and join-computation. Since we need to traverse the k-dominating set, we use connected k-dominating sets instead.

Definition 12 ***k-Dominating Set; Clusterheads; Connected k-Dominating Set; Gateways.*** *In a given graph G, a* k-dominating set (k-DS) *S is a subset of vertices of G such that each vertex in G is within the k hops of some node in S. We refer to each node in the set S as a* clusterhead*. A subset of vertices C is called a* connected k-dominating set (k-CDS) *if C is a k-DS and the subgraph induced by C in G is connected. A k-CDS C can be thought of as composed of a k-DS S of clusterheads and a set $C - S$ of* gateways *used to connect S.*

Constructing Connected k-Dominating Sets (k-CDS). The authors in [66] proposed a distributed and localized algorithm for constructing a k-CDS, which is suitable for our purposes. Since the connectivity of k-CDS is critical to our approach, we use reliable messaging (using messages acknowledgments and retransmissions) with [66]'s approach. In addition, we use the approach with multiple times with different sets of node IDs to construct multiple k-CDS. We refer the reader to [66] for details of their approach. To achieve load-balance, we construct multiple such k-CDS by using different random IDs for each node. With each node I, we also associate an arbitrary (preferably, the closest) k-CDS, and maintain a path connecting I to the associated k-CDS.

Tuple Storage and Join-Computation Phases. Note that each k-DS intersects (i.e., has a common node) with a k-hop neighborhood of any node. Thus, the k-DS and k-hop neighborhoods can be looked upon as "perpendicular" to each other. We connected the k-DS to allow easy propagation of tuples (for storage or join computation) over the clusterheads. In our discussion, we use the k-DS for storage of tuple and the k-hop neighborhoods for the join-computation phase. In particular, each new tuple generated at I is routed over the k-CDS associated with I and stored at the clusterheads. For join-computation, the tuple is joined (using a multiple pass scheme) with the tuples stored in the k-hop neighborhood $N_k(I)$ of I. Note that $N_k(I)$ is guaranteed to contain complete sliding windows of each data stream, since it intersects with every k-DS.

Choice of Parameters. The parameters k needs to be carefully chosen to optimize performance. In particular, if we use the k-DS for storage, then larger k entails lesser degree of replication and larger communication cost and delay. Opposite is the case when k-hop neighborhoods are used for storage. Also, k should be chosen such that a k-hop neighborhood is large enough to store all the sliding windows. After having chosen k, we construct multiple number of such k-CDS to ensure that each node is a clusterhead in at least one of the k-CDS.

Dynamic Topology. The constructed k-CDS are preprocessed data structures, and hence, need to be maintained in response of changes in topologies (node failures or additions). In our discussion, we assume that a failing node informs its neighbors about its impending failure. Such an assumption is reasonable for failures due to battery depletion. The above assumption can be easily relaxed by requiring each node to send periodic beacons. Since, each k-CDS can be maintained independently, we consider maintenance of a single k-CDS C.

Node Additions. When a new node I joins the network, it gathers $(k+1)$-hop neighborhood information. If there is no clusterhead in the k-hop neighborhood, then I selects itself as a new clusterhead. In either case, I connects itself to C (using new gateway nodes) using the gathered $(k+1)$-hop information. Here, we have assumed that the node I is connected to at least one network node.

Gateway Node Failures. Before a gateway node I_g dies, it gathers l-hop neighborhood information (for some l) and constructs a Steiner tree (in a centralized manner) connecting the neighbors of I_g that are in C. The nodes in the constructed Steiner tree are added to the set C, and notified of their membership in the k-CDS C.

Clusterhead Failure. The situation is more complicated when a clusterhead I_h fails. Here, we select some of the neighbors of I_h as new clusterheads, connect the selected new clusterheads with new gateways (if needed), and add all of these new nodes to C. Let B be the set of neighbors of I_h that are in the k-CDS C, and $\bar{B}$ be the set of neighbors of I_h that are not in C. We start off by selecting each element of B as a clusterhead, and designate each element of $\bar{B}$ as a *temporary clusterhead*. Next, we determine which nodes in $\bar{B}$ should be selected as clusterheads. First, all neighbors of I_h (i.e., $B \cup \bar{B}$) broadcast a `probe` message in their k-hop neighborhoods. Consider a node I that had only I_h as its clusterhead. If I does *not* receives a `probe` message from any node in B, then it sends a `make-permanent` message to the lowest-ID temporary clusterhead that I received a `probe` message from. A temporary clusterhead selects itself as a clusterhead (i.e., adds itself to C) on receiving a `make-permanent` message from any node. Finally, we add additional gateway nodes to connect the newly added clusterheads, by computing a Steiner tree (in a centralized manner) connecting them. Such a Steiner tree can be constructed by the failing clusterhead I_h (before failure) by gathering certain neighborhood information.

2.2.6 Performance Evaluation

2.2.6.1 Communication Cost Analysis

In this section, we analyze the total communication cost incurred in various approaches, which will be used to develop a heuristic for the join-ordering problem. Here, we define the communication cost incurred as the sum of the total number of hops traversed by each operand tuple. We start with considering the approaches in sensor networks with location information.

Definition 13 ***Selectivity Factor.*** *The* selectivity factor σ_P *of a join condition P between two data streams R_i and R_j is defined as the fraction of tuple pairs (one each from R_i and R_j) that satisfy the join condition P. More formally, $\sigma_P = |R_i \bowtie_P R_j|/(|R_i||R_j|)$.*

Communication Cost in One-pass PA. In PA, the total communication cost is due to storage and join-computation. For a newly generated tuple, the communication cost incurred in PA (in either one-pass or multiple pass) for storage is just the hop-length of the horizontal path. The communication cost incurred during the join-computation phase of the one-pass PA for sensor networks with location information can be computed as follows. Consider a vertical path of L nodes. Let us assume that the tuples of each sliding window are uniformly distributed along the vertical path. Consider a newly generated tuple t_1 of data stream R_1 (we choose R_1 for simplicity of presentation). In the one-pass scheme, t_1 traverses along the vertical path from one end (first node) to another end (L^{th} node). Consider the l^{th} node, i.e., the node that is l hops away from the first node on the vertical path. Below, we derive an expression for $N_l^2(\bar{n})$, the number of *new* partial results generated at the l^{th} node due to t_1, t_2 (some tuple of R_2; we choose R_2 for simplicity), and a set of $\bar{n}-2$ data streams other than R_1 and R_2. Here, $\bar{n} < n$ since we are counting only partial results. Let R'_i denote the part of the sliding window for R_i stored between the first and the l^{th} nodes. Note that $|R'_i| = |R_i| l/L$ if we assume uniform distribution of tuples across the vertical path. Then, the expression for $N_l^2(\bar{n})$ can be written as

$$N_l^2(\bar{n}) = \sum_{S \subset_{\bar{n}-2} \{3,\ldots,n\}} |t_1 \bowtie t_2 \bowtie R'_{i_1} \bowtie R'_{i_2} \ldots R'_{i_{\bar{n}-2}}|,$$

where the summation is taken over all subsets of size $\bar{n}-2$ from $\{3,\ldots,n\}$ and $S = \{i_1, i_2, \ldots, i_{\bar{n}-2}\}$ is an instance of such a subset. Now, let σ_{uv} denote the selectivity factor of the join condition between R_u and R_v. Then, we get

$$N_l^2(\bar{n}) = \sum_{S \subset_{\bar{n}-2}\{3,\ldots,n\}} \Big(\prod_{u,v \in (S \cup \{1,2\})} \sigma_{uv} \Big) \prod_{u \in S} (|R_u| l/L).$$

Now, the total number of partial results generated at l^{th} node is $\sum_{\bar{n}=2}^{n-1} \sum_{i=2}^{n} N_l^i(\bar{n}) |R_i|/l$. Recall that each of the partial results of size $\bar{n}$ traverses the remaining part of the vertical line, and hence, incurs a communication cost of $\bar{n}(L-l)$. If L_h is the hop-length of the horizontal path (and hence, the communication cost incurred for storage), the total communication cost (OP_PA_Cost) incurred by PA one-pass scheme due to a tuple t_1 of R_1 can be given by:

$$\text{OP\_PA\_Cost} = L_h + \sum_{l=1}^{L} \sum_{\bar{n}=2}^{n-1} \sum_{i=2}^{n} \bar{n}(L-l) N_l^i(\bar{n}) |R_i|/l. \quad (\mathbf{1})$$

The above equation also applies to sensor networks without location information with L_h being the size of the region (k-CDS or k-hop neighborhood) used for storage and L being the size of the other region used for join-computation.

Communication Cost in Multiple-Pass PA. Computation of communication cost in the case of multiple-pass PA depends on the join ordering. For simplicity, let us assume that the order of the join is $R_2, R_3, \ldots, R_n$. Consider a newly generated tuple t_1 of R_1. We start with considering networks with location information. In the i^{th} iteration of the multiple-pass scheme, the partial results corresponding to the tuples in $t_1 \bowtie R_2 \bowtie R_3 \ldots R_i$ traverse the entire vertical path to find matches from R_{i+1}, the next data stream in the join ordering. Before that, each of these generated partial results also traverses $L/2$ hops to get to the first node (to get ready for the next iteration). Since the first iteration incurs a communication cost of L (hop-length of the vertical path), the total communication cost (MP_PA_Cost) incurred by PA multiple-pass scheme for the join ordering $R_2, R_3, \ldots, R_n$ in response to a generated tuple t_1 of R_1 is:

$$\text{MP\_PA\_Cost} = 1.5L\Big(\sum_{i=2}^{n-1} i(|R_2||R_3|\ldots|R_i|) \prod_{1 \le i_1, i_2 \le i} \sigma_{i_1 i_2}\Big) + L_h + L. \quad (\mathbf{2})$$

Recall that L_h (hop-length of the vertical path) is the communication cost incurred during the storage phase. As before, the above equation also applies to sensor networks without location information.

Communication Cost in Centroid Approach (CA). The above analysis for multiple-pass scheme can also be used to compute the communication cost incurred by CA. If r is the memory available at each node, then $|R_i|/r$ is the number of nodes in the storage region $\mathcal{C}_i$ storing the sliding window for R_i. Let D_i is the average distance (in hops) between a network node and the storage region $\mathcal{C}_i$, and d be the average distance (in hops) between two storage regions. Then, the total communication cost (CA_Cost) incurred by CA in the join-computation phase for the join-ordering $R_2, R_3, \ldots, R_n$ in response to a generated tuple t_1 of R_1 is:

$$\begin{aligned}\text{CA\_Cost} = (\sum_{i=2}^{n-1}(d+|R_{i+1}|/r)(i)(|R_2|\ldots|R_i|)\prod_{1\le i_1,i_2\le i}\sigma_{i_1 i_2})\\ +D_1+2|R_1|/r+(d+|R_2|/r). \quad (\mathbf{3})\end{aligned}$$

Above, $2|R_1|/r$ is the communication cost incurred in searching for a node with available memory in $\mathcal{C}_1$, and $(d+|R_2|/r)$ is the cost of routing to and broadcasting t_1 in the storage region $\mathcal{C}_2$.

Communication Cost in Naive Broadcast Approach (BA). BA can be looked upon as a special case of PA, wherein the horizontal path only has the origin of the tuple, and the vertical path has all nodes in a circular region (the entire network in join without spatial constraints). Thus, the same formula of PA can be directly applied here. The total communication cost of multi-pass BA is showed as follows.

$$\text{BA\_Cost} = N(\sum_{i=2}^{n-1} i(|R_2||R_3|\ldots|R_i|)\prod_{1\le i_1,i_2\le i}\sigma_{i_1 i_2})+N. \quad (\mathbf{4})$$

Join Ordering Problem. The *join-ordering* problem of finding an optimal ordering of data streams that minimizes the overall communication cost (Equation 2 for multiple-pass PA or Equation 3 for CA) can be shown to be NP-hard using a reduction from maximum clique [67]. Note that the above join-ordering problem is equivalent to finding the optimal left-deep tree for evaluation of the given join query, and that the overall communication cost incurred in our multiple-pass PA and CA is proportional to the sum of the sizes of the intermediate results. Thus, we could directly use the techniques developed in [67] to determine a communication-efficient join-ordering of data streams. In particular, we use the greedy heuristic of [67] which works by first selecting the data stream that minimizes the communication cost incurred in the last iteration of the join-computation phase, as the last stream in the ordering. After picking the last data stream, the best choice for last-but-one data stream in the ordering is selected, and so on. The above greedy heuristic essentially works on the premise that the communication cost incurred in the last iteration dominates the overall cost. As typical sensor network queries are long running, we assume that all the catalogue information needed (estimated sizes, locations of the operand relations and join selectivity factor) can be gathered by initial sampling of the operand tables.

2.2.6.2 Simulation Results

We present our simulation results that compare the performance of various approaches. We simulate our algorithms on *ns2* [33], a general purpose network simulator capable of simulating wireless ad hoc networking protocols. Since our techniques are targeted for large sensor networks (hundreds of nodes), it was infeasible to simulate our techniques on real sensor networks or real sensor data (since the largest available data we could find online is for 30-40 nodes).

Parameter Values and Settings. We generated random sensor networks by randomly placing 1000 nodes in an area of 3000×3000 meters. We fix the transmission radius of each node to be 250 meters to ensure a connected network graph, and consider the case of join of 4 data streams. Each stream is generated uniformly across the network at the rate of 150 tuples per unit time. We compute the join based on a sliding window of size 150 tuples (or one unit time) for each data stream. The default memory capacity at each node is 30 tuples, but we vary it in one set of experiments. Note that *the absolute value of the sliding window size or memory capacity per node is immaterial for purposes of performance comparison*; thus, we only vary the ratio of sliding window size to the memory capacity by varying the latter. We set the battery energy, transmission power, receiving power of each node to 120J, 0.28W, and 0.14W respectively. By default, we store a tuple on every *other* node (of the horizontal path) in the storage phase, and do a one-hop broadcast from each node on the vertical path; we consider different replication factors in one set of experiments. We use a uniform selectivity factor (1/2 for spatial joins and 1/10 for non-spatial join) for all pairs of streams. For the given selectivity factors and sizes of sliding windows, the communication-cost Equations 1 and 2 suggest that the multiple-pass will be more efficient than the one-pass scheme. Thus, we use multiple-pass scheme for PA. Note that beyond the choice of one-pass vs. multiple-pass, *the absolute value of selectivity factors does not have any effect on the relative performance of the approaches*. In one set of experiments, we use non-uniform selectivity factors, and compare the performance of one-pass versus multiple-pass with different join-orderings.

Performance Metrics. We use the following performance metrics (over time) to measure the performance of our approaches: total battery energy dissipated (i.e., total communication cost), number of battery-depleted nodes, and "approximation ratio" of the output results. The *approximation ratio* of an approach at a given time is defined as the ratio of (i) the number of result tuples output by the approach, to (ii) the total number of tuples in the actual join result (computed independently in a centralized way), due to the input tuples generated in the last T units of time. In our graph plots, we choose T to be 10 units. The approximation ratio metric signifies the *current* state of the network (based on last 10 units of time) and incorporates almost all aspects of the performance of an approach. Thus, we use approximation ratio as our main performance criterion. *Network lifetime* can be defined as the amount of time for which the approximation ratio remains above a certain threshold; we consider 80% threshold in our discussions. Low approximation ratio could be due to node failures, message collisions, non-availability of sliding window tuples due to limited memory and/or network partitioning. We expect load-balanced and efficient PA to have a much higher network lifetime than CA.

Approaches. For the given parameter values, the Naive Broadcast approach is infeasible; e.g., for default values each node can only store 57.3% (= $(30 * 3000^2/(600 * \pi(500)^2))$ of the entire sliding windows and thus, the approximation ratio can be at most 57.3%. Thus, we implement the *Local Storage (LS)* approach, which stores each tuple only locally (at its source node) and uses multiple passes (as in the multiple-pass PA scheme) for join computation. In each pass, each partial result is broadcast upto the range of the spatial join or to the entire network for the case of non-spatial join. LS approach is load-balanced, but incurs more communication cost than PA. Below, we compare CA, PA, and LS approach for various parameter values and settings. For the PA scheme, we include all the overhead cost, except for the minimal (two messages per boundary node) one-time cost of computing the boundary nodes and markings. Also, duplicates are an inherent fault-tolerant feature of PA, and are not eliminated. Finally, as discussed in Section 2.2.3, the result tuples are output across the network. Collecting results at a central *node* will have similar problems as in the Centralized Approach discussed in Section 2.2.2; hashing of result tuples or shipping them to a central server connected to all nodes will have similar cost for all schemes and is thus ignored.

Spatial Join of Range 500 Meters. We start with considering performance of various approaches for the case of a spatial join of range 500 meters. As mentioned before, we use an *additional* (beyond the selectivity due to the spatial constraint) selectivity factor of 1/2 for all pairs of streams. We plot our simulation results in Figure 8. We see that the rate of energy dissipation in PA is less than in CA or LS. The rate of energy dissipation tapers off in each approach after some time, due to decrease in the number of active nodes. We notice that in PA the nodes start failing much later than in CA or LS, due to the communication-efficient and load-balanced operation of PA. Finally, we can see in Figure 8(c), that the approximation ratio of PA stays close to 100% for a long time. Essentially, when all nodes are alive, the fault-tolerance of the approach makes up for the few lost messages. The message collisions were observed to be rare due to "non-convergent" communication pattern and low rate of tuple generation. If we use the approximation ratio threshold (for network lifetime) of 80%, then the network lifetime of PA is about 3 times longer than that of LS and about 12 to 15 times longer than that of CA.

Varying Memory Capacities. In Figure 9, we compare performance of various approaches for different values of memory capacities (10, 60, and 90 tuples per node). We observe that PA continues to outperform both CA and LS by a large factor (3 to 10) in terms of the network lifetime. Note that LS approach doesn't change with change in memory capacity. We observe that the performance of PA is same for memory capacities of 30 or more, and the performance of CA improves with

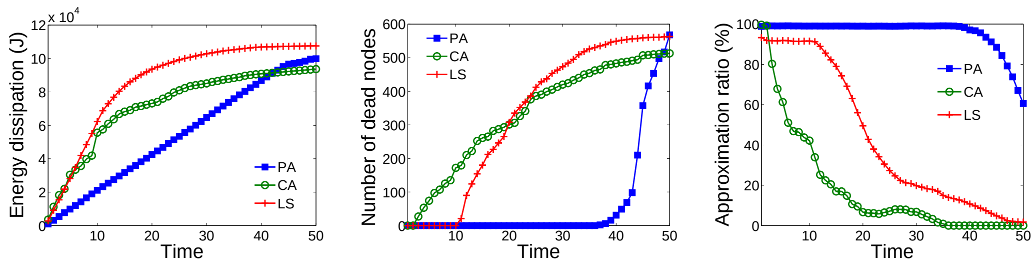
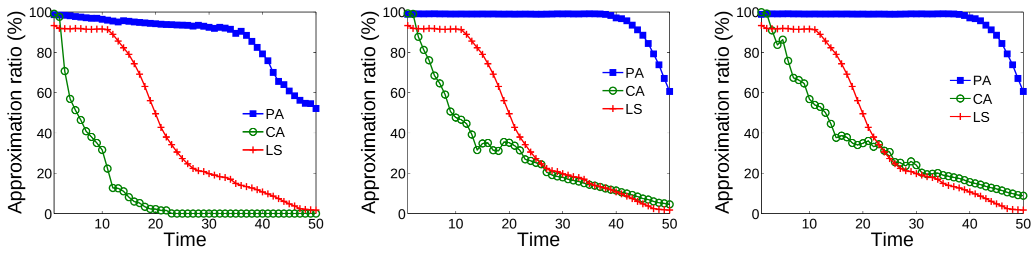


Figure 8: Performance of various approaches for a spatial join of range 500 meters with memory capacity of 30 tuples/node. (a) Total energy dissipated, (b) Number of node failures, and (c) Approximation ratio.

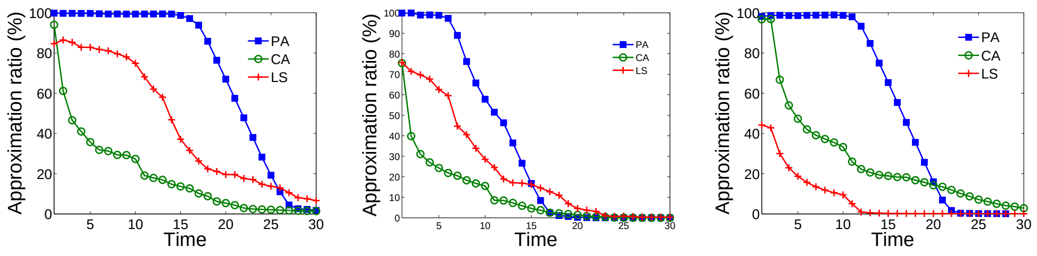


(a) 10 tuples per node. (b) 60 tuples per node. (c) 90 tuples per node.

Figure 9: Approximation ratios over time for the spatial join of range 500 meters with different memory capacities.

increase in memory capacity.

<u>Different Spatial-Join Ranges.</u> In Figure 10(a)-(b), we consider other ranges of spatial join, viz., 750 and 1000 meters. Since the transmission radius is 250 meters, considering lower range value is too perfect for PA, and a value of 1500 or higher will almost cover the entire network (and hence, equivalent to a non-spatial join). Here, we plot only the approximation ratio, since it incorporates all the performance



(a) Range of 750 m. (b) Range of 1000 m. (c) Non-spatial join.

Figure 10: Approximation ratios over time for different spatial-join ranges, and non-spatial join.

metrics. For the range of 750 meters, the network lifetime of PA is about 3 times longer than LS and about 20 times longer than CA. For the range of 1000 meters, both CA and LS have an effective network lifetime of zero, while that of PA is about 10. The low approximation ratios of CA or LS (even in the initial phases) is due to a large number of message collisions in the join computation phase, which requires repeated broadcast (within the storage region for CA or entire network for LS) for each computed partial result. Note that even though CA does not incorporate spatial joins, performance of CA worsens with increase in spatial-join range due to the increase in the overall selectivity factor.

Non-Spatial Join. As mentioned before, for non-spatial join, we use a selectivity-factor of 1/10 for each pair of streams. Since the overall selectivity-factor for the non-spatial join is perhaps (they aren't easily comparable) less than the spatial join of range 1000, we see that CA and PA perform better for the latter. See Figure 10 (c). Moreover, we see that LS performs very poorly; it has an approximation ratio of at most 50%.

Varying Replication Factor (k) and Memory Capacity. In this set of experiments, we vary the replication factor k, which signifies how often we store each tuple on a horizontal path. As mentioned in Section 2.2.4, if we store a tuple at every k^{th} node on the horizontal path, then we need to do a $\lfloor k/2 \rfloor + 1$-hop broadcast from each node on the vertical path. However, in most cases, the last hop broadcast is not required due to the inherent fault-tolerance of the approach and the random network topology. Thus, we use 1-hop broadcast for $k = 2$, and for $k = 3$ or 4 we do the last-hop broadcast with a 20% probability. Figure 11(a) plots the approximation ratio of PA for varying memory capacity. Here, we plot the approximation ratio during one unit of initial time, when all nodes are alive. As expected, we see that the approximation ratio decreases with decrease in memory capacity or replication factor. In Figure 11(b), we plot the communication cost for varying k and memory capacities per node. Increase in k should result in more energy dissipation. However, we notice that $k = 1$ incurs a much higher communication cost than $k = 2$ or 3, due to a large number of duplicate partial results generated when $k = 1$. From the given plots in Figure 11, we can conclude that to achieve an approximation ratio of at least 80%, we should use $k = 2$ (with one-hop broadcast) for memory capacity of 20 or higher. For lower memory capacities, higher values of k are needed. Note

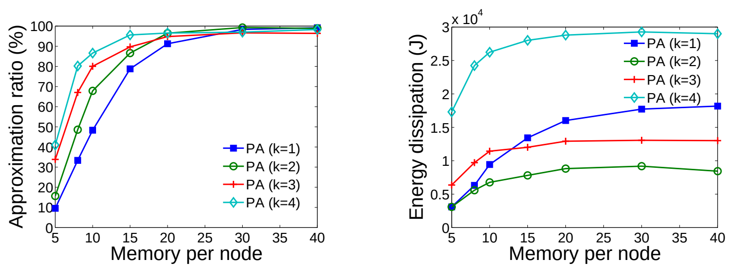
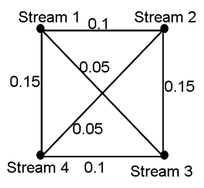


Figure 11: Varying replication factor k and memory capacity for non-spatial join. (a) Approximation ratio and (b) Energy dissipation, in one (initial) time unit.



Initial Stream	Greedy Ordering	Sequential Ordering
1	[3,2,4]	[2,3,4]
2	[4,1,3]	[3,4,1]
3	[1,2,4]	[4,1,2]
4	[2,1,3]	[1,2,3]

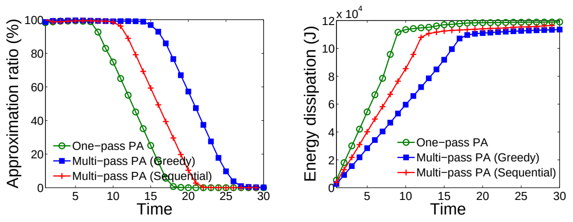


Figure 12: Non-uniform selectivity factors and join ordering for non-spatial join. (a) The join graph depicting various selectivity factors, (b) Greedy and sequential join orders, (c) Approximation ratio, and (d) Energy dissipation.

that $k = 1$ is never a good choice.

Effect of Join Ordering. We now depict the effect of join-ordering on the performance of PA. In this set of experiments, we choose non-uniform selectivity factors as shown in Figure 12(a). We compare the performance of one-pass PA and multiple-pass PA. For the multiple-pass PA, we use two different join orderings, viz., greedy (as described in Section 2.2.6.1) and sequential (where each new tuple iteratively picks the next data stream in sequence). See Figure 12(b). We see in Figure 12 (c)-(d) that the multiple-pass PA with greedy join ordering performs the best, followed by the multiple-pass PA with sequential join ordering.

Summary of Simulation Results. In our simulations, we have compared PA with other approaches for a wide range of network parameters. In general, we observed that PA resulted in a much longer network lifetime for computation of join, due to its communication-efficiency and load-balance. For spatial joins, LS outperformed CA, while for non-spatial joins LS performed very poorly. In general, the performance of CA and PA improve with increase in memory capacity. In the full version of the paper [32], we present more extensive experiments including evaluating our techniques in an irregular topology.

2.3 Double Rulings for Information Brokerage

In this section we address the problem of *information brokerage* which, specifies how data is collected and stored as well as how queries are routed to discover relevant data. We model the problem as the matching of *information producers* that perform data acquisition and event detection, with *information consumers* who search for this information. Naturally in a sensor network there can be multiple producers that generate a variety of data types as well as multiple consumers, possibly mobile, that search for relevant information. We aim to develop a scheme for large-scale networks that support low-delay queries for multiple users that search selectively for data types discovered and stored in the network. I present the key ideas of double rulings in this section and leave all details in our publication [55].

2.3.1 Background

Early work on information discovery and routing follow two basic approaches: data-centric routing [68] and data-centric storage [69]. Data-centric routing takes a reactive approach, as in directed diffusion [68] and TinyDB [14]. Little collaborative preprocessing is performed. Thus the discovery of the desired information usually relies on flooding the network. This approach targets at infrequent queries for streaming data type so that the cost of flooding can be justified and amortized by the following long-term data delivery. For queries from multiple consumers for the same data source, the performance deteriorates as data sources might be re-discovered separately by multiple consumers. The delay incurred by information discovery may also be too high for real-time queries in emergency response or delivery of control demands.

Data-centric storage is proposed for large-scale networks with many simultaneously detected events that are not necessarily desirable for all users [34, 69], as in the applications considered here. A producer leaves data on rendezvous nodes for consumers to retrieve. Thus data across space and time can be aggregated at rendezvous nodes. Prior work includes geographical hash tables (GHTs) [34] where data is hashed by its data type to geographical locations. The node closest to the hashed location is identified as the rendezvous node. The consumer applies the same hash function and retrieves data from the same rendezvous node. Data and query delivery to the rendezvous node is implemented by geographical routing such as GPSR [64]. GHTs have greatly reduced the communication cost and energy consumption by avoiding network-wide flooding for information discovery. Its simplicity is also attractive. There are a few weaknesses with GHTs though. First, the data retrieval scheme is not distance-sensitive. Even when the consumer is close to the producer, it may have to go to a far away rendezvous node. Second, the rendezvous node for popular data queried by many consumers imposes a communication bottleneck. This artifact in traffic patterns may eventually hurt the network lifetime. Third, the rendezvous node is a single point of failure. Structured replications on mirror nodes can be adopted to improve the system robustness but at a high cost of communication. Fourth, the property that data is randomly scattered in the network is good for load balancing, but bad for structured data organization and subsequently bad for queries that require cross-type data aggregations. Improvement of the flat hashing by hierarchical hashing has been investigated with hash locations aware of data correlation, i.e., similar data is stored close by, or query locality, i.e., nearby consumers should discover producers more quickly [70–72].

2.3.1.1 Double rulings

Our approach is to develop what is called *double rulings* scheme, an extension of the basic GHTs hashing. The idea is to choose the rendezvous nodes along a continuous curve, instead of one or multiple isolated sensor nodes, as in the case of GHTs [34]. The motivation is two-fold. Data delivery from data source to a rendezvous node is implemented by multi-hop routing. Thus it is natural to leave information hints along the trail that the data travels on, at no extra communication cost. Furthermore, data hint replication on multiple nodes provides more flexibility for a consumer to discover relevant data — it is easier to encounter a 1D curve than a 0D node.

The core idea of a basic *double-ruling* scheme is exactly the same as the Perpendicular Approach we proposed in Section 2.2. In the context of information brokerage, it works as follows: data or pointers are stored at nodes that follow a *replication curve* while a data request travels along a *retrieval curve*. Any retrieval curve intersects the replication curve for the desired data. Thus successful retrieval can be guaranteed. For an easy familiar case, assume the network is a two-dimensional grid embedded in the plane with nodes located at all the lattice points. The information storage curves follow the horizontal lines. The information retrieval curves follow the vertical lines. To be differentiated with the double rulings we will propose, we call this simple double rulings scheme the rectilinear double rulings. Notice that the data retrieval curves are independent of the location of the data sources. In fact, a consumer traveling along the vertical line through itself is guaranteed to hit *all* horizontal storage lines, and thus is able to find *all* the data stored in the network. This double-ruling scheme is also distance sensitive — if the producer and consumer are actually near each other, they must also be near each other along the path connecting them using the horizontal and vertical lines. By replicating data on more nodes that are not in close proximity with data sources or hashed locations as in GHTs, double rulings scheme enables better fault tolerance against geographically concentrated node failures.

Despite all these good properties, the rectilinear double rulings idea is so far restricted on networks with nice graph structures, e.g., those that resemble grids [54, 73, 74], due to its rich geometric flavor. The recent work by Fang *et al.* [75] also studied double rulings for routing purpose in a general sensor field with non-trivial topology. We also note that rumor routing [76] can be considered as a probabilistic double rulings scheme. Information producer takes a walk (either a random walk or a straight trajectory) and leaves data pointers on the trail. A consumer travels along another walk hoping to encounter one of the data pointers. Any two walks have a probability to intersect. The consumer sends out enough retrieval walks to have a sufficiently high probability to meet with one of the event curves. Essentially the challenge of designing good double rulings is to find data replication and retrieval paths that intersect, are not too long each (not too many replications), and are evenly spread out across the network. In our join work [32], the Perpendicular Approach is extended to general topologies by routing along appropriately defined horizontal and vertical paths. It can be directly used for information brokerage. In this work we further investigate double rulings schemes with a focus on the flexibility of retrieval mechanisms.

2.3.1.2 Our contribution

We propose a simple double rulings scheme that actually has GHTs as a subcase. Same as in GHTs, a data item is hashed by its data type (also called key in GHTs) to a geographical location. However, instead of traveling along the geographical greedy path to the rendezvous node, the producer travels along a circle that goes through itself and the rendezvous node and replicates data or data pointers on the way. We show that this simple modification to GHTs suddenly allows a large variety of retrieval mechanisms. The consumer does not necessarily travel to the hashed location to retrieve the data. It only needs to hit the replication curve. And we show that there are many such retrieval curves. Thus the consumer has great flexibility to design its retrieval strategy subject to the current network load and energy level. Among these retrieval schemes, several have special properties:

- **Distance-sensitive retrieval:** if the consumer is of distance d from the producer, the consumer can discover the data with a cost of $O(d)$, although neither has the knowledge of each other's location or the bound on d. This is an attractive feature in many applications, as information will be most useful, thus queried more frequently, in the spatiotemporal locale where it was collected.
- **Aggregated data retrieval:** in GHTs, if a consumer is interested in multiple data types, such as detections of both vehicles and animals, the consumer has to visit multiple rendezvous nodes for these data types to collect all the data. In our double rulings scheme, we show there is a simple rule based on which one can design a curve (actually many such curves) that will surely intersect with all replication curves of desired data types. Thus the consumer travels along a simple curve and gather all the information.
- **Double rulings retrieval:** the most powerful retrieval mechanism is to travel along any double ruling curve (among many such curves) that will intersect all replication curves. Thus a user can discover all the information discovered and stored in the network. This has further applications in data collection by data mules.

Our double rulings scheme improves the weaknesses of GHTs, with modestly increased replication. As explained above, it supports distance-sensitive retrieval and structured data retrieval. In addition, the double rulings scheme has substantially improved load balancing and robustness to node failures. With the flexibility in retrieval curves, the rendezvous node is no longer a bottleneck since retrieval curves may not necessarily visit it. We show that the data storage admits a local recovery scheme. If the sensors in a certain region are destroyed, then all the relevant data are stored on the boundary and thus can be locally recovered. Compared with structured replication in GHTs or hierarchical hashing that aims to improve data robustness, the double rulings scheme imposes much lower communication cost for replication, since the replicas are organized along a closed curve that are easy to visit.

We name this new double rulings scheme *spherical double rulings*, to be differentiated from the simple double rulings scheme with vertical/horizontal lines (which is denoted as *rectilinear double rulings*). As it may not be apparent, the spherical double rulings philosophically generalizes rectilinear double rulings and contains it as a subcase. The key insight about the difference between spherical and rectilinear double rulings and why the spherical double rulings provides more nice features will be discussed in subsection 2.3.3.1, after the description of our design.

2.3.2 Spherical double rulings

In this section we will use a continuous domain for the intuition and easy explanation. In a discrete network, a continuous double ruling curve can be easily implemented by a path in the network in a greedy fashion [77]. The implementation details and evaluations are presented in [55]. As in GHTs, we assume that the sensor nodes know their geographical locations and a few parameters of the sensor field such as the diameter and the boundary.

2.3.2.1 Projective mapping

For an easy explanation, we use projective geometry to map sensor nodes onto a sphere. There are several ways of projecting points on a sphere one-to-one to points in the plane. One commonly used mapping is stereographic projection [78]. Specifically, we put a sphere with radius r tangent to the plane at the origin. Denote this tangent point as the south pole and its antipodal point as the north pole. A point

h^* on the plane is mapped to the intersection of the line through h^* and the north pole with the sphere. See Figure 13 (i). This provides a one-to-one mapping of the projective plane $\mathbb{P}^2$ to the sphere, with the north pole mapped to the point of infinity. More details on projective geometry can be found at [79]. Stereographic projection preserves circularity. Any circle on the sphere, including great circles, is mapped to a circle in the plane. It is also a conformal mapping, i.e., one for which local (infinitesimal) angles on a sphere are mapped to the same angles in the projection. It does not preserve distances or area, however. The distortion around the north pole can be high.

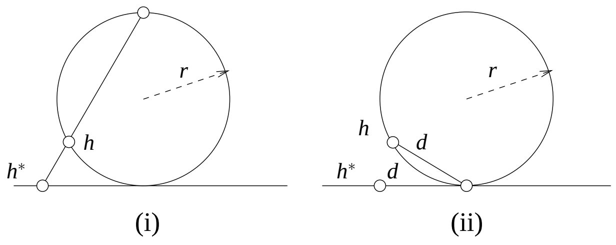


Figure 13: Spherical projection (i) stereographic projection; (ii) equal area projection.

Let the sphere be defined by the equation $(\mathbf{x}-\mathbf{p})\cdot(\mathbf{x}-\mathbf{p}) = r^2$ where $\mathbf{p}$ is the center of the sphere, and r is its radius. The straight line from a point $\mathbf{q}$ to the north pole of the sphere (denoted by $\mathbf{n}$), is given by $\mathbf{l}(t) = \mathbf{q}+t\mathbf{v}$, where t is the parameter and $\mathbf{v} = \mathbf{n}-\mathbf{q}$. Then the intersections of the straight line with the sphere are defined by the roots of the quadratic equation

$$t^2(\mathbf{v}\cdot\mathbf{v}) + t(2\mathbf{v}\cdot(\mathbf{q}-\mathbf{p})) + ((\mathbf{q}-\mathbf{p})\cdot(\mathbf{q}-\mathbf{p}) - r^2) = 0.$$

One root corresponds to the north pole $\mathbf{n}$, and the other is the projection. Thus, given the sphere, and a point in the plane, we can compute the mapping of the point on the sphere.

Conversely, given a point $\mathbf{h}$ on the sphere, its projection on the plane will lie on the straight line $\mathbf{l}'(t) = \mathbf{h}+t\mathbf{w}$, where $\mathbf{w} = \mathbf{h}-\mathbf{n}$. We define the plane by $(\mathbf{x}-\mathbf{o})\cdot\mathbf{z} = 0$ where $\mathbf{o}$ is the origin, and $\mathbf{z}$ is the unit vector perpendicular to the plane. Then the projection of $\mathbf{h}$ on the plane is given by

$$\mathbf{h}^* = \rho(\mathbf{h}) = \mathbf{h} + \frac{(\mathbf{o}-\mathbf{h})\cdot\mathbf{z}}{\mathbf{w}\cdot\mathbf{z}}\mathbf{w}.$$

The stereographic projection maps an infinite plane onto a sphere. For a sensor network field, the area in which the sensor nodes lie correspond to a finite region of the plane. Let this region be called **S**. Thus, any point in **S** maps to a point $\mathbf{h} = (x, y, z)$ on the sphere where $z \leq k$ for some $0 < k < 2r$. The radius r can be adjusted for a suitable value of k in this range. The distance from the origin to the point $\mathbf{h}^* = \rho(\mathbf{h})$ is given by $2r\sqrt{z/(2r-z)}$. Also, the distance from the origin to the point $(x, y, 0)$ is given by $\sqrt{z(2r-z)}$.

With the knowledge of the sensor field, we can place the sphere at the center of the sensor field. Suppose the furthest sensor node is of distance D from the origin (the south pole of the sphere). Then the parameter k, i.e., the z-value of the highest projection on the sphere, is at most $2r \cdot \frac{D^2}{4r^2+D^2}$. In [55], we show that for a finite region, we can choose r such that the mapping gives a constant distortion on the distances. Specifically, we choose r as $D/(2\sqrt{\varepsilon})$, $\varepsilon > 0$. $k = 2r\varepsilon/(1+\varepsilon)$. Recall that circles on the sphere map to circles in the plane. The next Lemma shows that the lengths on the circle on the sphere is not too much different from the lengths on the circle in the plane. The proof can be found in the full version paper [55].

Theorem 14 *Consider any two points p_1 and p_2 on the sphere with their projections on the plane, $\rho(p_1)$ and $\rho(p_2)$. If the distance from p_1 to p_2 along a circle is d, and the distance between $\rho(p_1)$ and $\rho(p_2)$ along the projection of the circle is ℓ, then we have*

$$\frac{\ell}{d} \leq \frac{2r}{2r-k} = 1+\varepsilon.$$

When $\varepsilon = 1$, all the points map to the bottom half sphere. We usually take ε as a constant larger than 1. For any two points in **S**, their distance in **S** along a circle is within a constant factor of the distance between their mappings on the sphere along the corresponding circle.

We use $d(\cdot,\cdot)$ to represent the geodesic shortest distance between two points on the sphere and $|\cdot|$ to represent the Euclidean distance in the plane. Thus we have,

Corollary 15 $|p^*q^*| \leq (1+\varepsilon)d(p,q)$.

Proof. The shortest distance between p, q on the sphere must be along a circle. The distance between p^* and q^* along the projected circle in the plane is bounded by $(1+\varepsilon)d(p,q)$. Further, the Euclidean distance of two points is always smaller than the distance along any circle. Thus $|p^*q^*| \leq (1+\varepsilon)d(p,q)$. □

With this mapping specified, we will explain our replication and retrieval schemes on a sphere. The above theorems imply that we can focus on the distances on the sphere. The real distances travelled in the sensor field are bounded by at most a constant multiplicative factor[4].

2.3.2.2 Data replication

For points on a sphere, there is an intuitive double ruling scheme — any two great circles of the sphere must intersect. Thus we can use great circles as the double rulings to replace the horizontal and vertical lines in rectilinear double rulings. There is one difference however. In rectilinear double rulings, the replication curves and the retrieval curves purely depend on the locations of producers and consumers. Through each node, there is a unique horizontal line and a unique vertical line. A point on a sphere, however, stays on infinitely many great circles. This property implies that the producer and consumer curves can have a lot of flexibilities, as we will see in the following.

We design a double rulings scheme that actually includes GHTs as a special case. Each data type is hashed to a geographical location h^* as in GHTs. When a producer routes towards the hashed location, instead of following the geographical greedy route as in GHTs, it follows the great circle defined by its own location p and the hashed location h, denoted by $C(p,h)$. Data from different producers with the same data type will be routed to the same hashed location where information aggregation can be performed. All the great circles with type $C(*,h)$ pass through the hashed location h, as well as the antipodal point $\overline{h}$. Thus there are actually two rendezvous nodes, h and $\overline{h}$, located far away in the network that have all the information of the same data type. Notice that the hashed location h depends only on the data type. Thus the location $\overline{h}$ can be derived by a simple geometric computation. See Figure 14 for an example.

By the properties of stereographic mapping, a great circle is mapped to a circle in the plane. In particular, the image of any great circle of the sphere encloses the

[4]This is subject to the assumption that the projective curve is within the sensor field and the sensors are dense enough such that the hop count of the path is proportional to its Euclidean length.

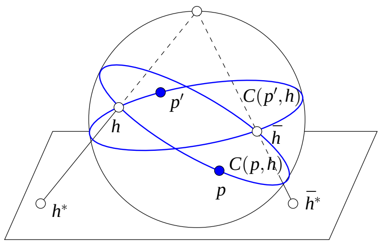


Figure 14: A point in the plane h^* is projected to a point h on the sphere. The great circles for two producers p, p' are drawn in blue.

tangent point of the sphere and the projected plane. These circles, i.e., replication curves, may have different sizes and centers. Figure 15 shows the actual routes followed by multiple producers.

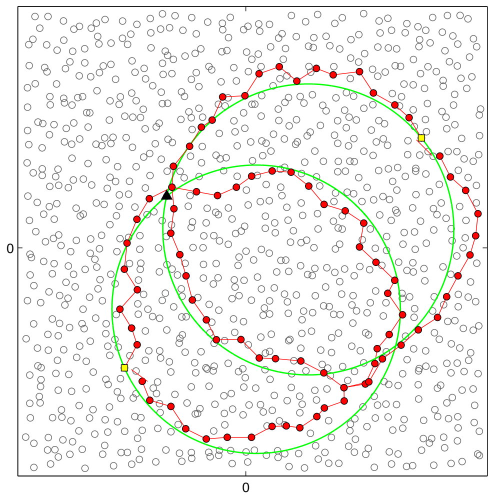


Figure 15: Replication curves of multiple producers with the same data type. The hashed location is denoted by the dark triangle. Both the virtual replication circles and the actual routing paths are shown.

The hash function picks two geographical locations h^* and $\bar{h}^*$. The rendezvous nodes are selected as those closest to these locations and can be discovered by greedy forwarding in a similar way as in GHTs [34]. We abuse the notation a little bit and use h^* and $\bar{h}^*$ to represent the hashed rendezvous nodes as well. The data is always replicated at the hashed rendezvous nodes h^* and $\bar{h}^*$. Data of the same type from multiple producers is aggregated at the rendezvous nodes h^* and $\overline{h}^*$. Dependent on the storage requirement, other nodes on the replication curve either store the real data or simply a pointer to where the real data is stored.

2.3.2.3 Data retrieval

With this new routing strategy from producers to hashed locations, the retrieval scheme for the consumer q can be more flexible than that in GHTs. Observe that the mapping described in section 2.3.2.1 leaves an empty region near the north pole of the sphere that projects to points outside the network, and it is possible that a curve chosen by the consumer on the sphere intersects the replication curve in this region. However, circular curves on the sphere will have two intersections on the producer curve, and we select our retrieval curves in ways that ensure that atleast one of the intersections projects to a point inside the network. We present a number of such retrieval rules as well as their properties.

GHT retrieval

Obviously the same retrieval rule as in GHTs can still be used, with two rendezvous nodes though.

Definition 16 ***GHTs retrieval rule****: the same as in GHTs, the consumer can route to the hashed location h or $\overline{h}$, whichever is closer, to retrieve all the data of the same type.*

This retrieval scheme, as in GHTs, suffers from two disadvantages. It is not distance sensitive. Even when the consumer is actually close to producer, the hashed location might be far. On the other hand, popular data items will create communication bottleneck around the rendezvous nodes that hold them. With the simple modification of the replication curve, we show in this section several retrieval schemes that are distance sensitive and also alleviate traffic hot spot for popular data. Besides, it is attractive to have the flexibility of different data retrieval schemes, simply for load balancing and routing robustness.

Distance-sensitive retrieval

Assume that the distance between a producer and a consumer on the sphere is d, we would like to have a retrieval scheme where the distance traveled by the consumer is $O(d)$. Such a retrieval scheme is named *distance-sensitive*. Notice that the consumer does not know where the producer is and vice versa. The goal of the retrieval scheme is to travel along a curve that hits the replication curve as quickly as possible.

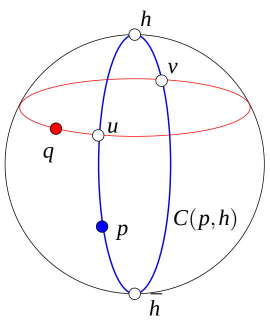


Figure 16: The consumer follows the circle with fixed distance to the hashed location h to retrieve all the data with the same data type.

If we rotate the sphere so that the hashed location h is at the north pole, then the replication curve is exactly a longitude curve. The distance-sensitive retrieval scheme follows the latitude curve searching for a replication curve. We denote by $L(q,h)$ this latitude curve. It is not necessarily a great circle. There are two intersections, u,v, between the retrieval curve and the replication curve, as shown in Figure 16. Now we claim that the closer one, in this case, u, is of distance at most $d\cdot\pi/2$ from the consumer along the latitude curve $L(q,h)$. Obviously, the minimum distance from a point q to a set of points $C(p,h)$ is always smaller than the distance from q to one point in this set, for example p. The following lemma says that the distance between q and u along the latitude curve $L(q,h)$ is at most a factor of $\pi/2$ of this shortest distance. The proof of the lemma appears in [55].

Lemma 17 *Take a longitude curve C through the north pole h and a latitude curve $L(q,h)$ through a point q. Assume that u is the closer intersection of C and $L(q,h)$ to q. Denote by k' the distance between q and u along $L(q,h)$ and k the shortest distance from q to C on the sphere. Then $k'/k \leq \pi/2$.*

The consumer, however, does not know which direction to go to on $L(q,h)$ to find the closer intersection u. This can be easily solved by a doubling trick, where the consumer chooses a direction randomly and travels a distance 2^i, with i initially set as 0. If the consumer has not encountered an intersection with $C(p,h)$, it turns around, increases i by 1 and travels a distance 2^i along the opposite direction from q. The process stops when the consumer discovers the closer intersection. Suppose at this point we have a parameter i, then $d\pi/2 \geq k' > 2^{i-2}$, where k' is the distance from q to u along $L(q,h)$ and $d = d(p,q)$, the shortest distance between p,q on the sphere. The total distance traveled by the consumer is bounded by

$$2 \cdot \sum_{j=0}^{i-1} 2^j + k' \leq 9k' - 2 \leq 9\pi d/2 - 2.$$

In summary, we have

Definition 18 ***Distance-sensitive retrieval rule:*** *the consumer travels along the circle on the sphere with equal distance to the hashed location h, and uses a doubling trick to discover the closer intersection with the replication curve. The distance traveled by the consumer is at most $O(d)$, if the distance between producer and consumer is d on the sphere.*

The bound on the consumer cost is for the worst case scenario. We show by simulation later that the performance is pretty good if we just choose a random direction. We note that here we focus on the continuous replication and retrieval curves. In a discrete network, the curves are realized by routing paths. When two continuous curves intersect, the corresponding routing paths may either have a common node, or have a pair of crossing links. We remark that under a unit disk graph model, if there are two crossing links, then one node must have links to all the other three nodes. With wireless broadcasting, all the nodes in the neighborhood can hear the message and are able to respond if they have the data. In practice, a consumer can also explicitly check the neighbors along the retrieval path or a producer explicitly store pointers on the neighbors along the replication path.

Aggregated data retrieval

The data replication scheme enables a number of interesting retrieval schemes for aggregated data. If the consumer travels along the latitude curve $L(q,h)$ with h as the north pole, it actually can discover all the data with the same data type. In fact, *any* closed curve that separate the hashed location h from its antipodal point $\bar{h}$ will intersect all the replication curves with the same data type. Thus a consumer is given great flexibility in choosing the retrieval curve according to the current network traffic load and energy consumption level. We formalize the data retrieval rule for aggregated data of several data types $\{T_i\}$, $i = 1, \cdots, m$.

Definition 19 ***Aggregated data retrieval rule:*** *the consumer searching for all the data with data type $\{T_i\}$, $i = 1, \cdots, m$, can follow a data retrieval curve that, for each data type T_i,*

- *either goes through the hashed location $h = h(T_i)$ or $\bar{h}$, where the aggregates are computed and stored;*
- *or is a closed curve that separates h from $\bar{h}$, collects all the relevant data and computes the aggregates.*

We remark that the above retrieval rule does not specify a unique retrieval curve but allow infinitely many possibilities. In fact, this is one of the advantages provided by this double rulings scheme. The design of a retrieval curve satisfying this rule can be performed at each consumer node. All the information needed is the data type and their hashed locations. Thus multiple consumers searching for the same data type may choose, by their own decisions, different routes. This flexibility of data retrieval rule enables load balanced traffic patterns and routing robustness.

Double rulings retrieval

The double rulings property enables a full power retrieval scheme. A consumer q following any great circle will definitely cross all the producer curves.

Definition 20 ***Full power data retrieval rule:*** *the consumer travels along any great circle and is able to retrieve* all *the data stored in the network.*

Locality-aware data recovery

The idea of replicating on a 1-dimensional curve, rather than a 0-dimensional point, greatly enhances the system robustness to failures. In GHTs, geographical routing with the combination of greedy forwarding and perimeter routing is used to deliver data to the hashed node. A planar subgraph, such as the relative neighborhood graph or the Gabriel graph, is subtracted from the connectivity graph. When greedy forwarding can not find a neighbor closer to the destination, perimeter routing is adopted to traverse the face in the planar graph. Specifically, the hashed

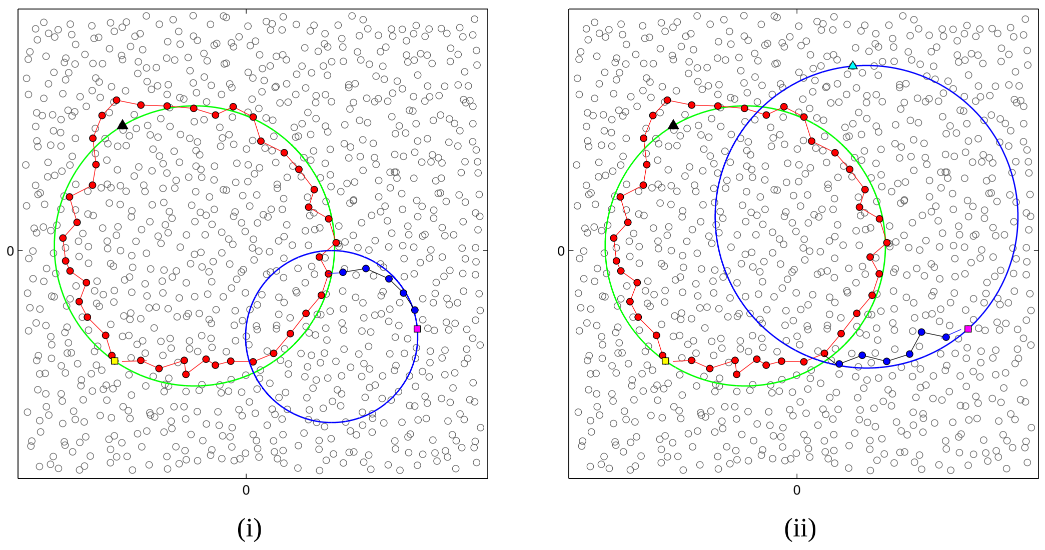


Figure 17: (i) Consumer latitude curve. (ii) Consumer great circle curve. Dark triangle denotes the hashed location; the red paths denote producer replication curves; dashed blue paths denote retrieval curves; yellow square denotes one producer and magenta square denotes one consumer.

geographical location, most likely, does not have a sensor node right there. Thus perimeter routing will be adopted to tour around the face that encloses the hashed location. The basic GHTs scheme relieves data loss at the home node, the one closest to the hashed location, by replicating the data around these perimeter nodes. However all the perimeter nodes are still in geographical proximity thus a ‘block error’ that destroys the sensors in a nearby region may destroy all the replicas. Structured replication can be used to improve the system robustness and relieve the traffic bottleneck at the home rendezvous node, in cases when too many events with the same key are detected in the network. Producers only put data at a nearby mirror node, while consumers may need to access multiple mirror nodes until they get what they want. The mirror nodes are chosen in a hierarchical way by using quad-tree structure. For the 1-level replication, the sensor field is partitioned into 4 equal size quadrants. The hashed location falls in one of them. 3 mirror nodes are chosen as those with the same relative locations inside other quadrants. More replication can be made in a recursive way. Such structured replication is costly since the mirror nodes are chosen to be geographically sparse. Replication along a curve improves the robustness without paying extra communication cost. Further we show that our replication rule supports local recovery when a group of nodes die.

In our spherical double rulings scheme, the hashed node is no long the single point of failure. If the nodes in the neighborhood of the hashed node h are destroyed, the nodes on the boundary of the destroyed region contain all the relevant information and can be used to recover the aggregates. This is possible as long as the destroyed region does not include both the hashed location h and its antipodal point $\bar{h}$. Since h and $\bar{h}$ are geographically fairly apart in the network, a local disaster is not likely to cover such a large region. All replication curves for this data type will leave data replicas on a curve connecting h and $\bar{h}$, thus intersect the boundary of the destroyed region. So all the data replicated inside the destroyed region have their corresponding replica on some boundary nodes. These boundary nodes can be detected by a local greedy sweeping as in [80], or by using a topological method as in [81].

2.3.3 Discussions

In this work we propose a simple replication mechanism that supports flexible retrieval mechanisms. Here, we compare the spherical and basic double rulings schemes and discuss future works along this directions.

2.3.3.1 Spherical and rectilinear double rulings

After the detailed description on spherical double rulings and the various nice features it provides for data retrieval and recovery, we will give some insights on the key difference between spherical and rectilinear double rulings and why replacing vertical/horizontal lines with circles suddenly makes our lives so much easier. The following discussion is on a philosophical and mathematically abstract level and the goal is to help the readers better understand the essence of our design.

Let us begin with some superficial differences between spherical and rectilinear double rulings. Through each point in a plane, there is only a unique horizontal and vertical line. Data replica are left on the horizontal line through the producer, which depends completely on the producer location. Different producers (not on the same horizontal line) with the same data type store data on parallel horizontal lines. Thus data items with the same type do not encounter each other and can not be aggregated in-network. The nice thing in spherical double rulings is to introduce a hash location that brings the producer curves with the same data type together at the hashed node. This allows data aggregation at the hashed location, consistent aggregated data query, etc.

From a projective geometry's point of view, however, two horizontal lines do intersect — at the point of infinity. As the projective plane is simply a sphere with the point of infinity mapped to the north pole by stereographic mapping, we can consider the rectilinear double rulings a special case of spherical double ruling — with all data types hashed to the point of infinity! Obviously there is no reason that all data types be hashed to the same node (point of infinity, i.e., the north pole of the sphere). In addition hashing to the point of infinity is not feasible in a finite sensor field. The spherical double ruling scheme essentially makes two natural modifications to rectilinear double rulings to do it right: we distribute hash locations grouped by data types and bring all hashed locations back to be within a finite sensor field.

However, which scheme is the best fit depends on our needs. For the implementation of join, we believe rectilinear double rulings (i.e., Perpendicular Approach) is sufficient because retrieval flexibility is not its major concern and more important, PA can easily incorporate spatial constraints.

2.3.3.2 Future Works

Double rulings with mobile nodes

Information collection and delivery can explicitly use mobile nodes, such as data mules [82–86]. This is motivated by the observation that nodes around static sinks suffer from unbalanced traffic and energy consumption. Furthermore, controlled mobility helps to get around fundamental capacity problems imposed by insufficient sensor density. In an extreme case, such as a disconnected network, mobile nodes have to be involved to deliver information between two disconnected components. However, designing the moving trajectory for data mule is challenging. One obvious metric is to have the data mule travel a short distance. Finding the shortest path that visits all the communication ranges of the nodes with data is a traveling sales man problem and is NP-hard [87].

We observe that data mules can be naturally combined with double ruling approaches to shorten the traveling distance of the data mule, with a modest in-network storage and aggregation. A mobile node physically traveling along a consumer curve is able to retrieve all the data in the network. This substantially decreases the distance traveled by the data mule. If the network is uniformly deployed in a squared region of n nodes. The shortest traveling salesman path is roughly $O(n)$ (visiting each node), but the double ruling curve has length roughly $O(\sqrt{n})$.

Advanced hashing

An additional variance that is not discussed is the choice of data-centric hash functions. The choice of hash functions is orthogonal to the double rulings scheme. In [55] we had used a uniform random hash function. Advanced hashing schemes, such as the one used in DIFS [88] or any distance-sensitive hashing schemes [65] that preserve data proximity can be directly incorporated. For example, we may prefer to hash similar data types nearby that may facilitate efficient aggregated data retrieval and rang join. The discussion of advanced hashing mechanisms and their interaction with double rulings will be interesting future work.

Sensor field with irregular shape

The spherical double rulings scheme is designed for a nicely distributed sensor field. In the case that sensors are deployed in an irregular shape with holes, the double ruling curves may accumulate on the hole boundaries. We can either resort to rectilinear double rulings as in the join work [32], or we can define double ruling curves in a virtual coordinate system that adhere to the underlying network geometry [75, 89]. For example, in the virtual coordinate system defined by the medial axis of the sensor field [89], there are natural double rulings curves, those that are parallel to the medial axis and those perpendicular to the medial axis.

Another approach to apply double rulings mechanism in a sensor field with complex geometry is to partition the sensor field into nicely shaped components and construct double ruling curves for each piece as in the GLIDER-based scheme [75]. We can also use our shape segmentation scheme presented in Chapter 4 to partition the field into nice shaped pieces and apply double rulings inside each piece.

Chapter 3

Make Implicit Information Explicit

3.1 Introduction

Most physical measurements exhibit strong spatial and temporal correlations, since physical phenomena are predominantly governed by the law of diffusion. Thus, spatially distributed sensor readings represents a time varying signal field. Studying the topology of such signal field and make its implicit but rich features explicit becomes fundamental for further information processing. In this section, we address the problem of tracking contours represented by binary sensors, and we focus on light-weight maintenance of changing contours and their topologies over time. This abstracted problem is motivated by a variety of tracking and monitoring applications.

Contour tracking scenarios. Consider an application scenario in which the sensors are used to detect and track chemical pollution. Each sensor measures the chemical intensity in its vicinity. As chemical contamination often comes from some pollution source, and the propagation of contaminants is typically by water current, wind, or diffusion, the pollution map exhibits strong spatial correlation and is often modeled and represented by a smooth signal field. The contaminated regions, having sensor readings above a danger threshold, naturally form a number of (possibly nested) blobs. Over time, the blobs may morph, merge, or split, indicating the pollution movement and/or the effectiveness of pollution treatment.

In another example, a group of targets moving in a field may alert the monitoring acoustic sensors nearby. Target movements in nature, such as human, vehicle, animal movements, have a tendency to be clustered. A group target can be monitored by tracking the contour of acoustic readings above a certain threshold. Contour changes reveal important information, e.g., the formation of a team or gathering, the dispersion of vehicles, or certain animal activities.

Contour tracking can also be applied to monitoring the health of the network itself. A well-behaving network should avoid traffic congestion and unbalanced energy depletion. Each node can locally determine its "health level" based on its traffic load and energy reserves. A contour tracking protocol identifies the congested regions and low-energy regions, providing the user a global view of network health.

Note that, in all of the above application scenarios, simply detecting nodes around the boundary of the evolving blobs is not sufficient. There can be a large number of nodes within a thick band identifying themselves on the boundary based on local readings, but none of them has a clear idea of the global picture of the entire signal filed — the number of disconnected pieces, the merging/splitting of them, the nesting relationships, etc. Those topological features of these contours are often of special interest to users. When monitoring contaminants, a user may query for a low-risk path through the geographical domain of the sensor field; if completely surrounded by hazardous chemicals, then special care (e.g., a rescue helicopter) may be needed. In tracking group targets, we may want to ensure that the targets do not surround a certain landmark. In monitoring network health, we want to make sure the network remains connected.

The challenges of contour tracking. We proposed a distributed algorithm that maintains, on the fly, the contours and their topological changes, while minimizing the use of network resources. We focus on the maintenance of contours of a single level-set (as in the scenarios above); the algorithm can be applied to the multi-level contours case and, potentially, to learning topological features of the sensor field.

We first survey previous work on tracking and contour detection and then explain the challenges of light-weight contour tracking.

Most prior work on tracking by distributed sensors focused on tracking of individual targets (e.g, vehicles, humans, animals) moving in the field (see [90–95]

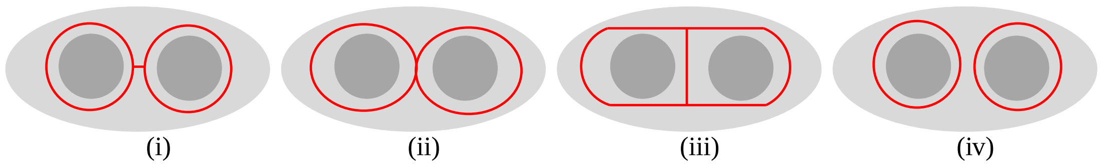


Figure 18: A field with two blobs. Figure(i)-(iii) show three valid contour networks of the gray band – all of them are deformation retract (intuitively, they all capture the fact that the gray band is connected and has two holes) but have different local features. Figure (iv) shows an invalid contour network. It is hard for an individual sensor to figure out the global topology.

and references therein). Tracking of a continuously deforming blob or of groups of targets is not as well studied. One could apply existing target tracking algorithms to each individual in the group, but this is not only *inefficient*, since only the sensors near the boundary of a possibly large blob need be involved, but also *insufficient*, since important topological changes (e.g., splitting and merging of blobs) are not easily available. Liu *et al.* [96] studied the problem of tracking a half-plane shadow by using geometric duality, exploiting the continuity of the contour and identifying the "frontier" sensors that may be included in the shadow in the future. Their approach, however, requires node locations, centralized pre-processing, and non-local communications.

In the static scenario, when the targets do not move or when considering a snapshot of chemical spreading, the problem reduces to detecting the boundary or the "holes" in the network, where sensors have abnormal readings. There has been a lot of work on boundary detection [80, 81, 97–102]. In the dynamic setting, one can periodically run a hole detection algorithm to discover contours. However, a major issue is choosing the update frequency — frequent updates waste network resources and infrequent updates miss critical changes. The update frequency is often dictated by the highest frequency of changes in the contours. Further, periodic update schemes requires global coordination and good network synchronization.

The problem we want to tackle here is to construct and maintain a *contour network* that abstracts the global topology of the contour components. This is very tricky since the topology of the contours represents a global feature of the signal field, and thus, an individual sensor cannot easily tell the computed contour network is valid or not (see an example in Figure 18). Our goal is to devise an algorithm in which each node maintains some local states, yet collectively they accomplish the global task.

Our contribution. We propose a light-weight and distributed algorithm to track the contours as they evolve over time. We construct and maintain a *contour network*, which tightly surrounds the contours and captures precisely the important topological features, e.g., how many connected components and how the contour components are connected and nested. See Figure 19 for an example.

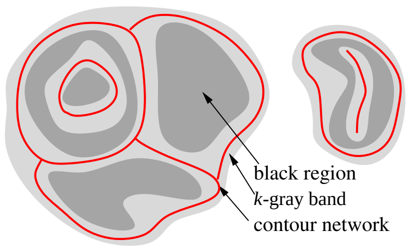


Figure 19: An example of the BLACK regions, k-gray band, and the contour network.

As the contours evolve, the basic idea is to freeze the valid segments in the old contour network, and only repair the contour network where it is broken. We propose a local algorithm such that, within only a local neighborhood of the broken contour, a node without the global knowledge can still repair the contour network and maintain the topological properties. Our algorithm has the following characteristics.

- the topology of the contours is captured precisely;
- the communication cost is "output-sensitive", being proportional to the magnitude of the changes to the contours, with the algorithm adapting automatically to the frequency and intensity of changes in the input data; and,
- the algorithm requires only local communication and does not require node location information.

The light-weight contour tracking algorithm serves as a fundamental network monitoring module, providing the basic input for further processing and representation of the signal field, e.g., for contour aggregation and simplification [103, 104]. It also allows efficient use of system resources, since the nodes not in the vicinity of the contour can stay on low duty-cycle and thus reduce energy consumption.

We present in Section 3.2 a distributed and practical implementation of the algorithm. Augmented with an additional process, we provide theoretical guarantee on the contour network property. We include the proofs in Section 3.3.

3.2 Contour Tracking Algorithm

3.2.1 Problem Setup

We consider a set of sensor nodes densely deployed in an environment of interest. We abstract the problem by assuming a continuous signal field σ covering the entire domain. Each sensor i has a reading, which is a discrete sample of the signal field at the location of i, measuring the value of a physical phenomenon being monitored. We consider the sensors are static, but the signal field evolves over time. For description simplicity, we focus on single-level contour tracking; i.e., there is a certain predicate that specifies the range of sensor readings of interest to us, and we define for each sensor a 0/1 variable called the “contour value” of the sensor.

Definition 21 ***Contour value****: The contour value $v(i)$ of node i is set to 1 if the reading of node i is within the range of tracked contour levels, and 0 otherwise. The set of contour values within the 1-hop neighborhood of node i (including i itself) is denoted by $V(i)$.*

A practical concern of the contour tracking algorithm is to deal with the issue of robustness of using the discrete values to approximate the continuous signal field. In particular, our algorithm only keeps track of contours of “sufficient significance”, which is formulated in terms of colors:

Definition 22 ***Color****: The color $c(i)$ of node i is defined as:*

- BLACK*: $0 \notin V(i)$, i and all of its neighbors have value 1.*
- WHITE*: $1 \notin V(i)$, i and all of its neighbors have value 0.*
- GRAY*: $0 \in V(i)$ and $1 \in V(i)$, a node that is neither* BLACK *nor* WHITE.

Thus, to enhance the robustness of the system we keep track of the connected components of BLACK nodes, called *black regions*. This introduces two benefits. First, a collection of sensors with “salt-and-pepper” type of contour values do not have significant contours to be tracked by our algorithm. Thus, we are more robust to noises in sensor measurement: A single value 1 in a group of sensors with value 0 does not trigger contour creation; rather, only when there is sufficient evidence witnessed by a node and all of its neighbors having value 1 does it indicate the node is BLACK and there is a contour worthy of tracking. Second, a BLACK node cannot be adjacent to a WHITE node, by definition. There is a GRAY band that separates the BLACK regions from WHITE regions. We are interested in learning and maintaining the shape and topology of the BLACK regions; symmetrically, we can track the WHITE region by reversing the contour values. Our algorithm outputs a *contour network* inside the GRAY band and tightly surrounding the BLACK regions as they evolve over time.

Definition 23 ***Contour network***: *A network of* GRAY *nodes within k hops from* BLACK *regions. Nodes of the contour network are called* RED *nodes.*

All of the contour tracking operations are performed on the GRAY nodes within *k* hops from BLACK regions, denoted as *k-gray band*. *k* here is a parameter that characterizes the tightness of the contour network in approximating the boundaries of the BLACK regions. See an example in Figure 19. We set *k* to 2 in our simulations.

In this section we will describe a practical algorithm for contour tracking in a distributed network. An augmented version of the algorithm, described in the next section, can be proved to maintain a contour network with the same topology as the *k*-gray band.

3.2.2 State Transitions

Our contour tracking algorithm is abstractly thought of as an automaton running at every sensor node. The state transition of the automata is based on the states of *k*-hop neighbor nodes, and does not require location information. The color of node *i* is the “state” of the automaton running at node *i* (see Figure 20). Information stored at each node is minimal, including its node ID, contour value, color, and RED neighbors on the contour network.

Node *i* notifies all of its 1-hop neighbors about the change of its contour value $v(i)$; and notifies its *k*-hop neighbors about the change of its color/state. All transitions happen when $v(i)$ changes or node *i* receives notifications from neighbors. At

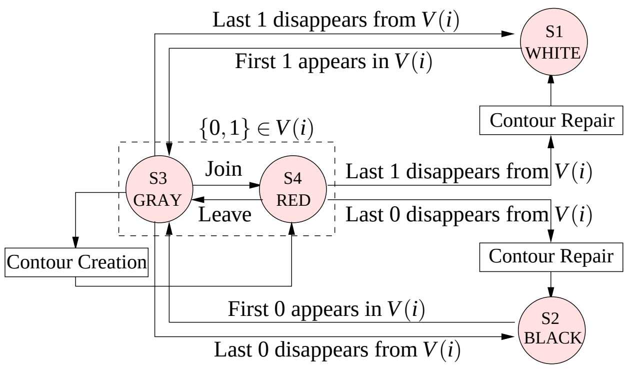

Last 1 disappears from $V(i)$
First 1 appears in $V(i)$
S1 WHITE
Contour Repair
$\{0,1\} \in V(i)$
S3 GRAY
Join
Leave
S4 RED
Last 1 disappears from $V(i)$
Last 0 disappears from $V(i)$
Contour Creation
Contour Repair
First 0 appears in $V(i)$
Last 0 disappears from $V(i)$
S2 BLACK

Figure 20: State transitions of the automata running at each node.

states S3 (GRAY) or S4 (RED), node *i* tracks its neighborhood to decide whether to stay at current state or change to BLACK/WHITE. If node *i* finds at least one neighbor with different contour value from itself, *i* remains in current state; otherwise it transits to S1 or S2 depending on $v(i)$. To do this, a node maintains two counters recording the number of neighbors with contour value 1 and 0 respectively, and decrease/increase the corresponding value as informed by its neighbors. When a RED node *i* turns to BLACK/WHITE, *i* must leave the contour network, which may result in a broken contour. Therefore, the transition from RED to BLACK/WHITE triggers contour repair. On the other hand, if new GRAY nodes find that they are close to BLACK nodes but cannot see RED nodes nearby (within its *k*-hop neighborhood), which is possibly a sign of the appearance of a new BLACK region, those GRAY nodes would start to collaboratively construct a new contour. The operations of contour creation and contour repair are atomic operations that are not interrupted by other transitions. Nodes are locked when they enter into either of these two phases until the repair/creation is finished. In the following, we discuss the details of contour repair.

3.2.3 Contour Repair

There are different cases that require contour repair — a single contour moves, expands, shrinks, splits, or multiple contours merge to one. Because every node can only see the changes within its local neighborhood, a sensor node has absolutely no idea of the global changes of the contour topology. Therefore, the major challenge in contour repair is to reconnect the broken contours in a distributed fashion, such that the resulting contour network is still topologically valid. In the following, we first describe the scenario of repairing a single contour cycle, then move to the cases of simultaneous repair of multiple contours, contour merging and splitting.

3.2.3.1 Repair of a single contour cycle

When a set of RED nodes leave the contour cycle (in the case of Figure 21 (i) these nodes change to BLACK), this leaves part of the contour cycle invalid and a few nodes with a missing RED neighbor. We call these nodes as *open* RED *nodes*, such as *a* and *b*.

Definition 24 ***Open* RED *Nodes & Closed* RED *Nodes***: *When a* RED *node loses (one or more) of its current* RED *neighbors, it becomes open. Otherwise, it stays as closed* RED *node. A* RED *node may also become open if triggered by others. An open* RED *node actively repairs the broken contour.*

When a RED node first becomes open, it is responsible for repairing the broken cycle. It initiates a repair message within the *k*-gray band, looking for other (open or closed) RED nodes to connect to. However, without location information a sensor node lacks the sense of directions. If we allow the repair message to propagate freely in the *k*-gray band, the resulting cycle may not be valid (as shown in Figure 21(i)). Therefore, it is important to block traffic traveling in the wrong direction, implied by the closed RED neighbors of the open RED node.

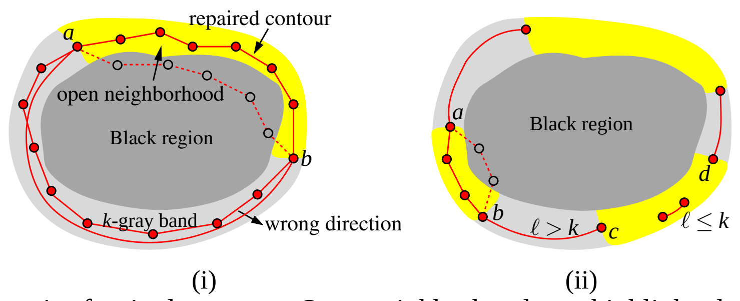


Figure 21: Repair of a single contour. Open neighborhoods are highlighted. (i) *a* and *b* are two open RED nodes. The repair message only travels within the open neighborhood. (ii) A single contour cycle is broken into multiple chains.

Definition 25 ***Blocked neighborhood and open neighborhood***: *a* RED *node is defined to be a* block node *if it is closed and is at least k-hops away from an open*

RED *node in a connected component of a contour cycle. The union of the k-hop neighborhoods of block nodes is the* blocked neighborhood *B. The rest of the k-gray nodes is called the* open neighborhood $O = \mathcal{G} \setminus B$. *The open neighborhood has a number of disconnected components, denoted as* O_i.

Clearly each connected open neighborhood O_i has at least one open RED node. The repair message only travels within the open neighborhood, as shown in Figure 21 (i). Since the repair operation is confined inside each O_i there is no interference between the repair efforts in different connected components.

Notice that for the two open nodes a, b, without the knowledge of each other, each would attempt to repair the contour. We suppress the repair messages by the ID of the initiator. In particular, the repair message will not be forwarded when it either (i) enters the blocked neighborhood, or (ii) arrives at some nodes who have received repair messages initiated by other open RED nodes with ID *lower* than its initiator. As the repair message is forwarded, an aggregation tree rooted at a is also cached on the nodes who forward the message. This aggregation tree helps the open RED nodes gather information about RED nodes encountered on the way. In particular, when a node stops forwarding, it returns with the ID of the RED nodes it learned so far. At an internal node of the aggregation tree, a node propagates up the aggregated information when all of its children have reported their information. If an open RED node learned through the information gathered that there is another open RED node with smaller ID, then it retires. The node with smaller ID, called the *repair node* for this open neighborhood, is responsible for the repair and selects a path connecting to the other open RED node. The GRAY nodes on this path are invited to join the cycle and turn to RED.

In general, a RED cycle may be broken into multiple chains (Figure 21(ii)). The repair is done in a similar way. All of the nodes with at least k hops away from an open RED node block their k-hop neighborhood. If a left-over contour chain is shorter than k, it is hard to distinguish the correct repair direction from the wrong direction at open RED nodes. Those short chains become un-repairable.

For un-repairable chains the open RED nodes find they are on chains with length $\ell < k$. They remove themselves from the contour network and turn back to GRAY. Eventually the entire short chain disappears. Remaining repairable chains still participate in cycle repair (Figure 21 (ii)). When a RED cycle is broken into all short chains, it is good time to discard all un-repairable chains and reconstruct a new cycle from scratch using cycle creation algorithm, to be explained later.

3.2.3.2 Simultaneous repair, merging and splitting of BLACK regions

When multiple BLACK regions are close, their repair processes may interfere with each other. An open RED node may see multiple open RED nodes. This typically happens when the topology of the BLACK regions changes, i.e., two BLACK regions merge together or one BLACK region splits into several, the contour network will need to capture the new topology.

When a RED node a first becomes open, again it initiates a repair message which is propagated in the k-gray band as before (with the k-th hop closed RED neighbor of a blocking the message propagation). We focus on the open neighborhood O_i containing a. If there are multiple open RED nodes in O_i, again the repair messages from all but the one with lowest ID are suppressed. The repair message travels to every node in an open neighborhood O_i and is only stopped if it hits the BLACK regions, or the boundaries of the k-gray band, or blocked by the k-hop node of some block nodes. Thus a learns about these block nodes, which are grouped into *bounding segments*.

Definition 26 ***Bounding segments.*** *We take the* RED *nodes within k-hop of an open neighborhood O_i and denote them the* bounding RED nodes *of O_i. Each connected component of the bounding nodes is called a* bounding segment.

A bounding segment may or may not have an open RED node. See Figure 22 for two examples. Now the repair node a simply connects through shortest paths to each and every bounding segment.

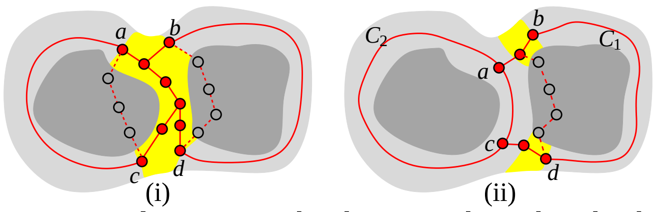


Figure 22: (i) The repair node a connects by shortest paths to the other bounding segments in its open neighborhood. (ii) The open RED nodes b, d connect to their respective bounding segment (in this case, a segment with only closed RED nodes).

In a special case, an open RED node may not see any bounding RED node in a different segment than itself, the contour repair operation gets stuck. Figure 23

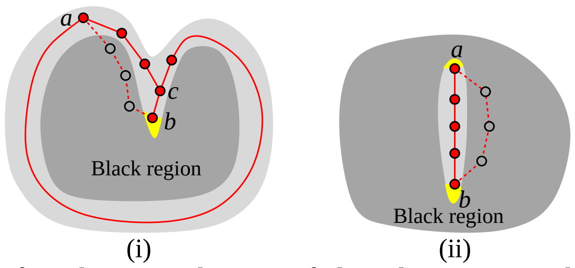


Figure 23: (i) Repair from *b* gets stuck since it fails to discover any other bounding segment to connect to. Node *b* becomes closed and triggers the next node to be open. (ii) Both open RED nodes *a*, *b* fail to repair and become closed. The adjacent node is triggered to be open.

shows two such scenarios. The open neighborhood of node *b* in Figure 23 (i) has only one bounding segment containing itself. In this case the repair operation at *b* will terminate and node *b* stays as it is. The next node adjacent to *b*, in this case, node *c*, will now become an open node and attempts to repair the contour. Figure 23 (ii) shows a case when both *a* and *b* can not discover any RED nodes other than those connected to them. This will eventually leave a RED segment as part of the contour network.

It is possible that some nodes between nearby red chains change states and become *k*-gray. In such a case the red chains may need to be connected or merged together to reflect desired homotopy. As such, a simple check for presence of red nodes within the *k*-neighborhood does not always suffice. So, a newly turned *k*-gray node does the following. It looks at its connected *k*-gray neighborhood (by initiating a *k*-hop flood and aggregation), and verifies that the red nodes in this neighborhood form a tree. If they do not, then all these red nodes are eliminated and become open. Open red nodes start contour repair as described above.

3.2.3.3 Contour creation and disappearance

Initially, as a BLACK region appears and grows, the creation of a new contour is triggered at some BLACK nodes that have a GRAY neighbor but cannot see RED nodes in its *k*-hop neighborhood, because this indicates the appearance of a new BLACK region not tightly surrounded. The BLACK node turns its GRAY neighbor to an open RED node.

Now a GRAY node *i* enters *contour creation* phase. It is possible that multiple GRAY nodes try to create a new contour for the same BLACK region. To avoid every GRAY node repeating the same thing, the GRAY nodes who want to start a contour will participate in a local leader selection procedure. The leader selection algorithm selects a node as a leader if no other node within its *k*-neighborhood becomes a leader. This can be done with any clustering algorithm or by local message suppression. After that, only leaders participate in the creation. The distance between any two leaders is at least *k* hop apart.

The leaders become open RED nodes. But we consider the *k*-hop neighborhood of a leader to be blocked to a different leader node. Now the leaders use the contour repair algorithm to find other leaders to connect to. The repair messages from leaders will meet either other bounding RED nodes or nodes with repair messages from other leaders/repair nodes. In both cases, the leader with smaller ID will be selected to build a path to connect to the desired party. GRAY nodes on the path will turn to RED too and together with the leaders form a red chain with length at least *k*. With the same contour repair protocol these chains will eventually be connected to a contour network.

The contour network disappears as the corresponding black regions disappear, because open RED nodes that are supposed to repair the cycle will detect that they are not within *k*-hop of any BLACK node. Those RED nodes would remove themselves from the contour automatically, and the contour disappears eventually.

3.2.3.4 Summary of the repair algorithm

The general principal of contour repair is that the valid segments of old contour network is still usable and repair only happens where the contour is broken. The repair node connects through a *spanning tree* to all other bounding segments of its open neighborhood.

We also emphasize a few issues in the implementation for algorithm robustness: (1) The contour network in an open neighborhood O_i is repaired by the open RED node with the minimum ID in the bounding segments. Thus even in a distributed setting there is always consistency agreed upon who makes the decision. (2) When a node is involved in a repair procedure, it is temporary locked and does not participate in other repair procedures. When the repair operation terminates, the repair messages and the nodes who forwarded them will be refreshed. (3) The repair effort is confined within the open neighborhood, whose size is proportional to the contour changes.

In most common scenarios, when there are no small black regions that newly show up in the k-gray band, it can be proved that the proposed repair algorithm generates a contour network that is a deformation retract of the k-gray band. Intuitively this means that the contour network is a proper "thinning" of the k-gray band, capturing all of the topological information of the band, e.g., how many holes there are and how they are connected/nested. We handle the small holes in Section 3.3 and give a rigorous proof of this homotopy equivalence property.

3.3 Maintenance of Topological Features

The previous section focuses on the practicality and implementation details for contour tracking. The main contribution in this section is to provide a theoretical understanding of contour tracking. In the proofs we consider a continuous setting in which there are BLACK regions and WHITE regions in R^2, separated by a GRAY band. The k-gray band $\mathcal{G}$ contains the GRAY points within distance k from BLACK regions. The contour network is denoted by G; an algorithm invariant is that $G \subset \mathcal{G}$. Our main theoretical result is an algorithm (an elaboration of that of the previous section, handling small holes inside $\mathcal{G}$) that, upon stabilization (when nodes no longer change state) computes a contour network with precisely the same topology as the k-gray band.

Rigorously, we show that the contour network G is a deformation retract of the k-gray band $\mathcal{G}$. A continuous map $\pi : \mathcal{G} \times [0,1] \rightarrow \mathcal{G}$ is defined as a *deformation retraction* if, for every x in $\mathcal{G}$, a in G, and $t \in [0,1]$, $\pi(x,0) = x$, $\pi(x,1) \in G$, $\pi(a,1) = a$. A deformation retraction is thus a homotopy between the identity map on $\mathcal{G}$ and a retract of $\mathcal{G}$ onto G. G is called a deformation retract of $\mathcal{G}$. See [105]. Intuitively, a deformation retraction shrinks a space to a 1-dimensional graph, while keeping the same topology of cycles and connected components.

3.3.1 Contour initialization

We first study in a static setting how to construct a contour network that is topologically equivalent to the k-gray band even with small holes. This algorithm will also be used as a subroutine in the contour repair algorithm, but is confined to the local neighborhood of the contour changes. The contour construction algorithm is motivated by an earlier boundary detection algorithm [81], but is much simplified.

Without loss of generality we assume that $\mathcal{G}$ is connected and has h holes. We compute the *shortest path map* for an arbitrary root, r, which summarizes the shortest path(s) from r to every point in $\mathcal{G}$. The union of points with two or more shortest paths of different homotopy types[1] are called the *cut locus*, C. It is known that C is a forest (let C_i denote the ith tree of C) with m interior vertices (called *SPM vertices*), $h+m$ branches, connecting the h holes and the m vertices to the region boundary. The removal of the cut locus leaves $\mathcal{G}$ simply connected and any shortest path cannot cross a cut branch or go through an SPM vertex [106]. The contour network is computed by putting back shortest paths of different homotopy types, which, together with a part of the cut locus, form the contour network G (Figure 24).

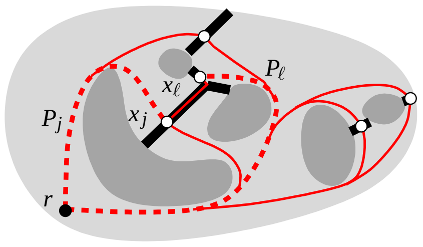


Figure 24: The contour network G of $\mathcal{G}$. G consists of representative shortest paths of different homotopy types (in red) and the cut locus (in thick black lines). We show two examples of the shortest paths P_j and P_ℓ in dashed curves.

In particular, once we remove the cut locus, the resulting field $\mathcal{G} \setminus C$ has no holes, and its outer boundary consists of the boundaries of holes and segments of cut branches, in an alternating fashion. On each such connected cut segment, we pick a point x_j (the *representative point*) and connect r to x_j along a shortest path

[1]Two paths with the same start and end points of different homotopy types get around the holes in different ways; thus, one cannot be continuously deformed to the other without jumping over some holes.

(*representative path*). For all representative points on one connected component of the cut locus C_i, their representative paths have different homotopy types – for any two such paths P_j, P_ℓ, as we continuously deform their endpoints x_j to x_ℓ within C_i, one cannot continuous deform P_j to P_ℓ. Notice that there can be multiple representative paths with the same endpoint x_j as long as they have different homotopy types. Let C'_i denote the subtree of C_i that connects the representative points on C_i. Now, the contour network $G = (\cup_j P_j) \cup (\cup_i C'_i)$ is the collection of representative paths and the subtrees connecting the representative points in each cut locus component.

3.3.1.1 G is a deformation retract of $\mathcal{G}$

Lemma 27 *G is a planar graph in $\mathcal{G}$, and each face contains exactly one hole of $\mathcal{G}$.*

Proof. G is a planar graph, since any shortest path cannot cross a cut branch or go through an SPM vertex [106]. We remove all of the paths P_j and the cut locus C. We are left with a set of connected components; each component is *simple*, with its boundary consisting of two paths P_j, P_ℓ, a portion of the boundary of *one* hole, and some cut branches. Now we put back $C_i \setminus C'_i$ and obtain the faces of G. Notice that the pieces adjacent to the same hole boundary are combined as one face in G. Thus, each face contains exactly one hole. □

Theorem 28 *G is a deformation retract of $\mathcal{G}$.*

Proof. We will describe a continuous map $\pi : \mathcal{G} \times [0,1] \to G$ that shrinks $\mathcal{G}$ to G. In particular, we remove G and are left with a number of connected components $\mathcal{G}_i$, each containing a hole H_i, corresponding to one face F_i of G.

We first describe how to deform $\mathcal{G}_i$ to F_i. We use a continuous function f to map the boundary of the hole to F_i. Now, we extend this function f to a map π_i such that the point $x \in \partial H_i$ is deformed through the shortest path from x to its corresponding point $f(x)$ on F_i. All of the points on this shortest path are also mapped to $f(x)$ accordingly. It can be easily verified that this is a continuous map and is a deformation retraction from $\mathcal{G}_i$ to F_i. The union of these maps inside each G_i is the deformation retract π. □

3.3.1.2 Implementation in a discrete network

This contour construction algorithm can be implemented in a discrete network as follows. We start from an arbitrary node i and flood the k-gray band. The k-gray band has h holes, which can be detected by discovering the *cut nodes,* i.e., the nodes with two or more shortest paths to i of different homotopy type. Denote by C_i a connected component of the cut nodes. These cut nodes are detected by checking whether two neighboring nodes have their least common ancestor (LCA) "far" away (on the other side of the hole) and their shortest paths "far" apart as well. Such a pair is called a *cut pair*. See Figure 25 for an example. By using appropriate parameters to define "far", holes above a given size can be detected by the recognition of these cut nodes [81].

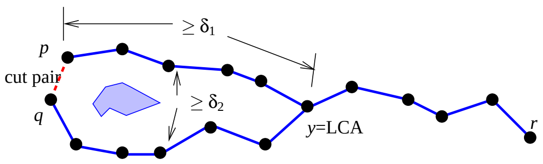


Figure 25: Definition of a cut pair (p,q).

To find the representative paths, we will first remove the edges between all cut pairs. The cut nodes are left in different connected components. Now we will take one node from each connected component, denoted by x_j, and include in G the shortest path from r to x_j. If any subset of these x_j's belong to the same connected component C_i, they are connected by a tree within C_i. This tree is also included in G.

3.3.2 Augmented contour tracking algorithm

We consider the snapshot of two different signal fields. For the first snapshot, we have a topologically valid contour network G for its k-gray region, $\mathcal{G}$. When $\mathcal{G}$ changes to $\mathcal{G}'$, any point that has changed its color (BLACK, WHITE, GRAY) will erase the contour network in its radius-k neighborhood. The part of G unerased contains some broken segments.

The augmented repair algorithm does just one additional operation on top of the repair algorithm in the previous section. In an open neighborhood O_i, recall that the repair node connects through shortest paths to all other bounding segments of O_i. Now, we also include new holes that possibly pop up in O_i. This is by including in addition the "local contour network" in O_i, constructed by the contour initialization algorithm described above.

We argue that after the repair the resulting network G' is topologically equivalent to $\mathcal{G}'$. Network G' has two parts – the old contour network $G_o \subseteq G$ and the newly repaired part G_n.

Theorem 29 *The contour network G' is a deformation retract of the k-gray region $\mathcal{G}'$, after the contour repair is done.*

Proof. Consider first the old contour network, G, and the k-gray region $\mathcal{G}$. We remove G from $\mathcal{G}$. We are left with a number of disconnected components, Γ_j. Each one of them is an annulus (band), surrounded by a hole boundary and a face F_i of the old contour network G. By assumption, we have a deformation retract π_j from Γ_j to F_i. Define the *width* of $\mathcal{G}$ as the maximum radius ball centered at a point $p \in \mathcal{G}$ such that the removal of this ball does not change the topology of $\mathcal{G}$. The width of Γ_j is at most k – otherwise, there will be a BLACK node with GRAY neighbors but do not have RED nodes within distance k. Thus, this node will trigger contour creation.

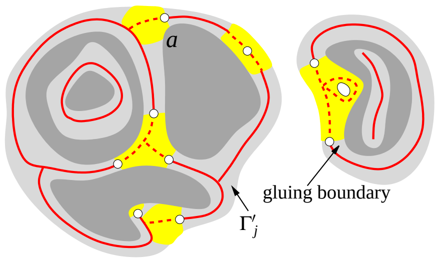


Figure 26: The repaired network G' and the repair regions (highlighted). Node *a* has a closed bounding segment and an open bounding segment (adjacent to itself).

Consider an open neighborhood O_i; it has some bounding segments, some with an open RED end (called *open* bounding segments), some without (called *closed* bounding segments). We define the repair region $R_i \supseteq O_i$ that includes all of the nodes p such that $\pi(p, 1)$ maps to a closed bounding segment of O_i. Intuitively, we are extending the open neighborhood O_i until it hits the closed bounding segments.

Now consider the new contour network G'. We remove the repair regions R_i and the old contour network G_o from $\mathcal{G}'$. This leaves a number of disconnected components Γ'_j, $\Gamma'_j \subseteq \Gamma_j$. If $\Gamma'_j = \Gamma_j$, then we define the deformation retraction π' in Γ'_j to be the same as π. If $\Gamma'_j \subset \Gamma_j$, i.e., part of the contour on F_i has been removed; thus, Γ'_j has the shape of a deformed band. This is because the removal of any point on F_i and its radius k neighborhood will "break" the annulus Γ_j, since the width of Γ_j is at most k. Now Γ'_j is bounded by part of a hole boundary H'_j and part of the face $F'_i \subseteq F_i$, and two "gluing boundaries" adjacent to some repair regions. We map H'_j to F'_j with a continuous function.

We now consider a repair region R_i. We first assume that each open neighborhood O_i is simply connected. Then the repair operation connects with shortest paths from the repair node $r_i \in O_i$ to the bounding segments of O_i. The repaired contour is completely inside R_i. This partitions the repair region into pieces such that each piece is bounded by a contour network segment F'_i, a hole boundary segment H'_i, and gluing boundaries adjacent to some Γ'_j's. We map H'_i to F'_i with a continuous function.

If the open neighborhood has holes, then the repair operation will also include the "local contour network" for O_i, which will partition O_i into disconnected pieces, each face containing exactly one hole with the outer face homotopy equivalent to the outer boundary ∂O_i. This does not interfere with the shortest paths to connect to the bounding segments. Again, the union of any additional shortest paths with the local contour network is still a planar graph.

What this says is that we are able to obtain a continuous mapping of each hole boundary in $\mathcal{G}'$ to a face boundary of G'. With the same argument as in Theorem 28, the homotopy equivalence is established. $\square$

In fact, the proof in the previous theorem states that we can start from a contour network, remove any subset of it, and use the contour repair algorithm to successful repair it. Since we freeze the nodes involved in an open neighborhood under repair, later value changes will be handled after the atomic repair process is finished. The repair operations in different non-overlapping open neighborhoods do not interfere with each other and are handled simultaneously and independently. Thus as long as the signal field stabilizes, the contour network will capture the same topology of the k-gray band. In a dynamic environment, as long as the computation efficiency can catch up with the data change rate, a valid contour network can be maintained. The topological equivalence result implies the following properties listed below with proof omitted from this version.

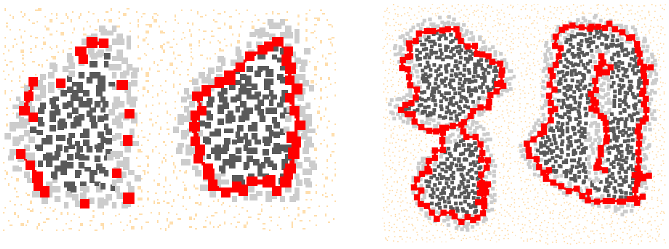
Corollary 30 *The following properties are true when the contour network G stabilizes (i.e., no point switches its state):*

1. *A continuous curve connecting one* BLACK *and one* WHITE *point will have to cross G. A continuous curve connecting two* BLACK *points, from different connected components will have to cross G.*
2. *The contour network is planar with a planar embedding.*
3. *Since all of the repair work happens in the open neighborhood, the communication cost for contour repair is proportional to the amount of contour changes.*

3.4 Simulations

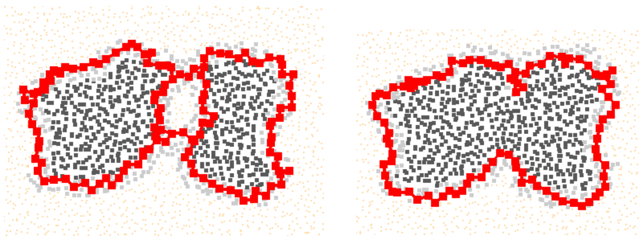
We implemented the contour tracking algorithm described in Section 3.2 in a simulator written in C++, since the algorithm covers all cases of contour evolvement, and works well in practice. We simulated on a network with 4000 nodes distributed in a field of size 500×500 units. Each node has transmission radius of 15 units. We set the parameter $k = 2$ by default, and vary k in one set of experiments to discuss its impact on the performance of the algorithm. The simulator takes a data field with arbitrary shape as inputs; in particular, we experimented with both simulated and real data (e.g., Figure32). We simulate dynamic changes among a sequence of stabilized states of a contour field. Between two states, sensor nodes can change their contour values in an arbitrary order. Video clips of some simulated examples can be found at [107].

Snapshots of Contours. We first show a set of snapshots at intermediate stages of the algorithm. Figure 27 shows the process of contour creation when a new black region appears. Initially, a few nodes within the k-gray band elect themselves as leaders and start contour creation. Since contour creation is done in a distributed manner, when new leaders appear, other leaders may already connect to red chains (Figure 27 (i), left). A complete cycle after creation is showed in Figure 27 (i) (right). In some cases, a new black region may be so close to an existing black region that their k-gray bands already merge together. Then, the new contour directly attaches to the existing contour, which guarantees homotopy equivalence (Figure 27(ii)). Contour creation is also triggered when gray/white areas are born inside black regions. The merging and splitting of contours are symmetric



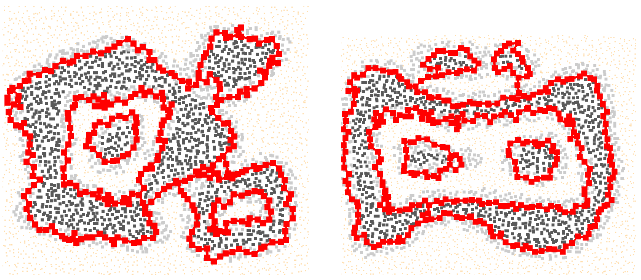
(i) (ii)

Figure 27: Contour creation: (i) Left: new leaders appear, and existing leaders connect to red chains. Right: a contour cycle is created. (ii) Left: a new cycle directly attaches to a red cycle nearby. Right: a red chain is constructed when a gray area appears inside a black region.



(i) (ii)

Figure 28: Merging and splitting. (i) Two black regions move closer. Their gray bands meet each other and (multiple) “bridges” are built up. (ii) Black regions themselves merge together.



(i) (ii)

Figure 29: Snapshots of nested contour network.

processes. Figure 28 (i) and (ii) show what happens when two originally distant black regions (e.g., the two regions in Figure 27(ii)) move closer and closer. When their k-gray bands just touch, a bridge is automatically constructed to connect those two contours together. In Figure 28 (i), two bridges appear, which exactly capture the white hole between them. If two black regions move towards each other further and eventually merge together, the contours also merge into a larger one (Figure 28(ii)). If we look at these snapshots in a reverse order, they exactly represent a typical process of contour splitting. More interesting snapshots are shown in Figure 29.

Irregular Network Fields with Holes. Our contour tracking algorithm is naturally resilient to boundaries and holes with arbitrary shapes. In Figure 30, we show examples of contour networks when a black region attaches to boundaries. The collection of contour pieces correctly separates the black regions from white.

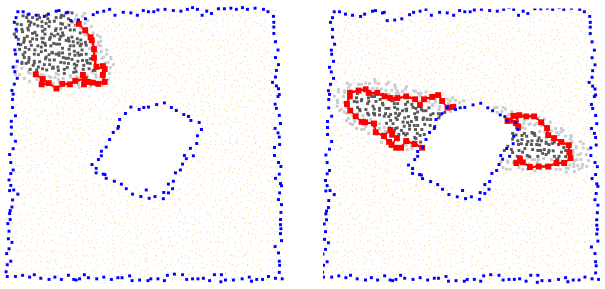


Figure 30: A contour initially sits at the boundary and successfully passes through a hole in the middle of the network field.

Multi-level Contours. Multi-level contours can be easily supported by applying the single-level contour tracking algorithm at each level independently. In Figure 32 (i), we show the multi-level contours corresponding to the elevation data of a small area in Maryland. We take 5 discrete ranges as contour levels of interest. Some sensor nodes may be on multiple contours at the same time if those contours are close to each other. The algorithm correctly maintains the topology of multi-level contours.

Impact of Parameter k. The parameter k controls the tightness of the contour network to the enclosed black regions. Figure 31 shows different contour networks with different choices of k for the same snapshot of the contour field. When $k = 1$, the k-gray bands are very narrow and have not met each other; thus, those two black regions are still enclosed by two separated red cycles. With $k = 3$, the k-gray

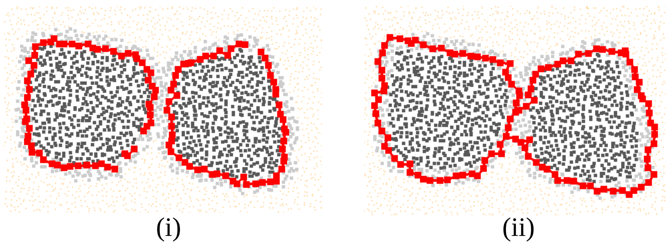


Figure 31: Tightness of the contour network: (i) $k = 1$. (ii) $k = 3$.

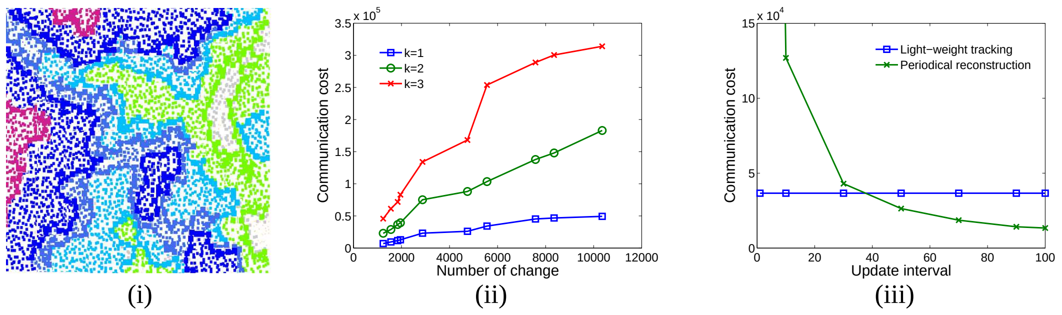


Figure 32: (i) Multi-level contours on elevation data. Colors represent elevation: purple is the highest and green is the lowest. (ii) Communication cost vs. the number of changes. (iii) Communication cost of periodic construction vs. update interval, compared with our tracking algorithm.

bands overlap and red cycles attach to each other. The average distance in terms of hop counts from each red node to a closest black node is about 1.0, 1.4, 1.7, for $k = 1,2,3$, respectively; i.e., as k increases, the contour network becomes “loser”.

The parameter k also affects the communication cost and the completeness of the contour network. Larger k incurs more transmissions (see Figure 32(ii)); on the other hand, if k is too small, the contour network is easily broken into pieces because a narrow k-gray band may not be a connected piece. Thus, there is a trade-off between communication cost, tightness and completeness. In our simulations, we find that $k = 2$ is suitable for most cases.

Communication Cost. We evaluate the efficiency of our algorithm in terms of communication cost, measured by the number of transmissions incurred during the transition from one stabilized state to the next. We use 10 different contour fields as inputs, and run simulations on each filed for 5 rounds.

Figure 32 (ii) shows that the communication cost is approximately linear in the number of changes, since all operations in our algorithm are executed locally.

We further compare the performance of our algorithm with a periodic contour reconstruction scheme (see Figure 32 (iii)). We can run any boundary detection algorithm to reconstruct contours periodically. In simulations, we chose to use our contour creation algorithm, since it is essentially a light-weight boundary detection algorithm, which captures the rough boundaries of the network field, and much cheaper than other accurate boundary detection algorithms. The update interval is defined based on the number of changes and is varied 1 to 100. If we run the periodical reconstruction scheme at every change, it will incur much higher cost, about 30 times the cost of our tracking algorithm, which is out of the range of Figure 32 (iii). With larger intervals, the cost of the periodic reconstruction scheme is reduced, but more critical changes are missed. When the update interval is set to about every 40 changes, it achieves a comparable communication cost with our tracking algorithm, but sacrifices in tracking quality.

3.5 Contour Tree Aided Network Navigation

To further explore the usage of contour tracking in real-time interaction with users, we propose to build up a compact data structure — distributed contour tree, which helps the users navigate the signal field. The details are presented in [108].

Consider a scenario in which sensors and users (such as rescuers or patrol officers) are embedded in the same physical space. Users carry hand-held devices and are able to directly communicate with nearby sensors. In particular, the users want to obtain directions to places that require attention or service, which are indicated by the sensor data being within a specified range. A naive solution is to flood from the query node the entire network, with all the nodes within the range responding to the query. This, however, can be quite energy expensive as many nodes not involved in the query will still be checked. Instead, we would like to quickly get the iso-contours of the limits of the query range and only report those nodes back to the user. These iso-contours bound the regions of interest, and may involve much fewer nodes than all of the nodes within the query range. To abstract the problem, we want to support the following routing and navigation function with a low communication cost:

Iso-contour query: from a query node q, find the iso-contours at value x. And its *variations such as counting/reporting iso-contour components at given value/range.*

The most intuitive solution is to use gradient routing, by exploiting the natural continuity of the signal field. Starting from the query node q, the query message can be greedily guided either downhill or uphill, depending on the comparison of the value at q and the target value x. This greedy descent routing is simple and requires only local knowledge. Thus it has been explored in a number of settings for low-cost data-centric routing [109–113]. Greedy descending/ascending can typically lead the query message to one iso-contour, unless the query message reaches a local minimum or local maximum, in which case the query gets stuck. Indeed, using simple gradient descent for an iso-contour query has a serious defect: the signal field may have multiple peaks and valleys, and greedy descending discovers at most one iso-contour, and is not able to discover all of the iso-contours due to the existence of local optima.

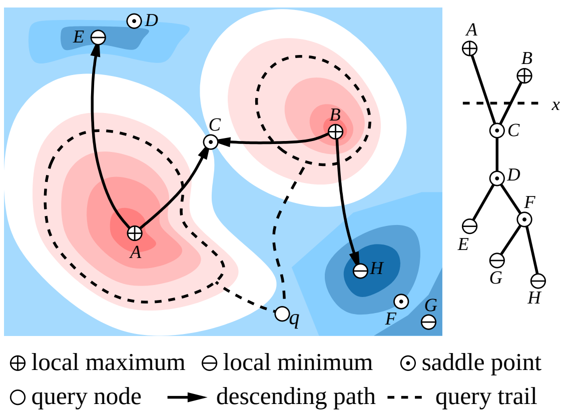

D
E
C
B
A
H
q
F
G
A
B
x
C
D
E
F
G
H
⊕ local maximum ⊖ local minimum ⊙ saddle point
○ query node → descending path - - - query trail

Figure 33: The level sets of a signal field and the contour tree spanning all the critical points (in the right). The figure also shows *some* descending paths connecting the critical points.

Figure 33 shows an example of a potential field by drawing its level sets. Red colors mean hot and blue colors mean cold. We also show all the local maxima, minima and saddle points. A greedy gradient routing from a query node q looking for a desired level contour will follow the local gradient and climb up the mountain. Once the query reaches the desired level it can locally trace out one contour, e.g, the contour on the left peak in the figure. However with only local information the query does not know whether there are other peaks and if so where they are.

The difficulty here is that the greedy gradient routing is completely local, while iso-contours reflect the global topology of the signal field. This is a general problem in navigation with a potential field, as has also been studied in robotics: with only information about the local potential one lacks the big picture of the signal field which is important for guaranteed success. In particular, the collection of critical points (local maxima, minima and saddle points) represents the global topology of the signal field. Thus, in order to make the local greedy descend algorithm always work, one needs to augment it with a compact representation of the critical points and their relationships.

Our contribution. We propose to investigate distributed algorithms to pre-process the iso-contour structures, say with our contour tracking algorithm, of the signal field by what is called the *contour tree* [114], using which a gradient routing scheme can successfully discover *all* iso-contours. In short, a contour tree is a tree on all the critical points of the signal filed and captures the topology of the iso-contours. It is a special case of the *Reeb graph* in Morse theory [115]. Take Figure 33 as an example, the right figure shows the topological contour tree consisting of eight vertices, corresponding to two local maxima, three local minima, and three saddle points. A contour tree captures how the connected components of the iso-contours merge/split as we increase/decrease the isovalue.

We propose an algorithm for the construction of the contour tree in a fully distributed manner. The basic idea is similar to the centralized construction [114, 116–118]. But we need to account for numerous robustness issues due to local noise and degeneracies, and lack of global coordination. We use distributed sweeps [119], initiated at local maxima and minima to identify the saddle points and nodes on the saddle contour. Next an information dissemination phase following the contour tree structure distributes necessary information for gradient routing. The preprocessing involves all together four rounds of sweeps of the signal field and has a linear message complexity.

With such distributed contour tree structure, we design iso-contour routing and low-value routing with guaranteed success. We note here that in this work we only consider a static signal field, because the problem for a static signal field is already quite challenging. In practice, as the signal evolves over time we can periodically execute the contour tree construction phase. The maintenance of the contour tree for a time-varying signal field will be future work. In the following, we briefly describe the contour tree construction algorithms and two routing schemes built on top of that. We leave details in the full version paper [108].

3.5.1 Contour Tree Construction

A contour tree is essentially a combination of a merge tree and a split tree (see Figure 34). At a merge saddle (e.g., p in Figure 34(ii)), two contour components merge into one; and at a split saddle (e.g., s in Figure 34(iii)), one contour components splits into two. The basic idea of the contour tree construction algorithm is that we build up a merge tree and a split tree by sweeping top down and bottom up separately, then combine them together. Without loss of generality, we explain the details with the sweep top down here.

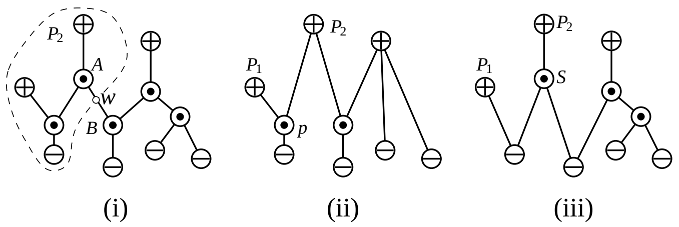


Figure 34: (i) A contour tree and the interior of $C(w)$ shown in the bounded region; (ii) merge tree; (iii) split tree.

A node has its *higher neighbors* as the subset of neighbors with value strictly higher than itself, and its *lower neighbors* as the subset of neighbors with value strictly smaller than itself. Each sweep is initiated and labeled by a critical node (a maximum, minimum or a saddle node). A node identifies itself as a local maximum if it discovers that all its 1-hop neighbors have value no greater than itself. It then initiates a sweep top down. The sweep algorithm runs in a distributed fashion on all the nodes. A node has two possible states, *swept* and *not swept*. Each local maximum node initializes itself as a swept node. When a node has all of its higher neighbors in the swept state, it changes itself to be swept. The nodes who participate in the sweep do not need to be synchronized. They decide on their own state and advance the sweep frontier with only local knowledge.

In the sweep initiated by a local maximum p, the sweep message carries the tuple $(p, \mathcal{F}(p))$, i.e., the node ID and value of p. Each node being swept will keep this information, as well as from which nodes it received this information. We define a *descending path* as a path in which each node has a value no greater than its precedent. During the sweep the information about a local maximum p is propagated along descending paths from p. In additional each node swept learns an ascending pointer which eventually leads to the local maximum. If a node gets two sweep messages from different local maxima, this indicates that two contour components start to merge. Thus a saddle should be identified.

Definition 31 *We define a node to be a* merge saddle node *if it is the one with highest iso-value with two descending paths from different critical points (other merge saddles or local maxima), i.e., it receives two sweep messages from different critical points.*

Since the nodes advance the sweep frontier in a distributed fashion, it may happen that two nodes at the same time both receive the sweep messages from two peaks. In a distributed setting we need to worry about two issues: (i) two nodes u, v (or more) may become potential merge saddles $S(P_1, P_2)$ for the same two peaks. In this case only the real saddle node (the one with highest isovalue) should survive. (ii) it may happen, if the sweep frontier does not proceed in the same speed, that the lower saddle may be discovered before the higher saddle. We resolve the ambiguities by the traversal of the contour component at a potential saddle. The details are in [108]. Similarly, we can identify split saddles by sweeping bottom up.

After identifying all the nodes on the critical contours, we use gradient descending and ascending paths to discover the contour tree. Starting from a merge saddle $p = M(P_1, P_2)$, we follow gradient ascending paths towards P_1, P_2 respectively. If the ascending path towards P_1 reaches P_1 before it hits any other critical contour level, then p will consider P_1 its parent in the contour tree. If the ascending path towards P_2 hits a split saddle contour S, then p will consider S as its other parent in the contour tree. Similarly p also sends a descending path and identify its child in the contour tree. The operations for a split saddle, maximum/minimum are very similar. Thus, the contour tree will be detected precisely as the combination of the merge tree and the split tree.

Information stored at each node. The invariant we maintain on a node p is: (1) the maximum/minimum value, $I^+(w), I^-(w)$, inside the *interior* of its contour component $C(w)$; (2) the maximum/minimum value, $E^+(w), E^-(w)$, inside the *exterior* of $C(w)$. For example, in Figure 34 (i). A node w on an arc AB has a contour component $C(w)$ in between $C(A)$ and $C(B)$. The contour component $C(w)$ decomposes the entire signal field into two components, the *interior* and the *exterior*, corresponding to the two subtrees when edge AB is removed. The interior contains the critical point A, which is reachable from $C(w)$ via a gradient ascending path. The exterior contains the critical point B, which is reachable by a gradient descending path. In addition, w also keeps the node ID of the local maxima/minima it stores. This information is to guarantee that when we send a query message either uphill or downhill, we know for certain that there exist some contours for which we are looking. For the consideration of network load balancing, each node also keeps information about the contours that split off/merge together at their *ascending merge saddle* or *descending split saddle*. Take saddle A in Figure 34 (i) as an example. The contour component $C(A)$ is the union of two contours $C_1(A)$ and $C_2(A)$ that just merge together. Thus we keep at each node $u \in C(p)$, with p on the arc AP_2 (with A as a descending split saddle), (1) the maximum/minimum values of the interior/exterior of both $C_1(p)$ and $C_2(p)$; (2) gradient descending pointers leading to $C_1(p)$ or $C_2(p)$ or both. This information helps us decide before we reach a saddle contour, whether it is worth visiting one or two of the contour components that split off of it and if so, how to get there.

To summarize, each node only keeps a small constant amount of information. The information can be easily obtained by another two-round sweeps.

3.5.2 Gradient Routing with Guaranteed Success

The gradient algorithm uses only the information stored at a node and its immediate neighbors. Starting at q we first check whether x is beyond the range of the signal field, in which case we do not travel even one step and immediately return $\emptyset$. Effectively, this is by checking whether $I^+(q) < x$ and $E^+(q) < x$, or $I^-(q) > x$ and $E^-(q) > x$. If not, we know that there must be some non-empty iso-contours at level x and we use a greedy gradient algorithm to find them. At the query node q,

- If $I^+(q) \geq x \geq I^-(q)$, then q initiates a query message to follow the gradient uphill.

- If $E^+(q) \geq x \geq E^-(q)$, then q initiates a query message to follow the gradient downhill.

We first explain the ascending query message from q. If a query message hits a node w with isovalue x, it will then start a traversal along the contour component $C(w)$. This is done by the same algorithm as explained earlier. At the same time, we also need to check at w whether it is worth getting even higher up — it is possible that at the interior of $C(w)$ there are still contours of value x. Again this is done by checking a higher neighbor of w, say v, whether $I^+(v) \geq x \geq I^-(v)$.

For an ascending query message at a node w, suppose w stays on an arc with p being an ascending merge saddle. Then we will check for two parents of p, denoted by p_1, p_2, whether we will need to ascend on one peak or both of them. Luckily this information has been disseminated for all the nodes on this arc. Thus w will check the value range within the interior of $C_1(p), C_2(p)$ respectively. If the query value x falls in the range, w will initiate an ascending query message for it. See the red query in Figure 35 as an example of two query messages, one for each peak.

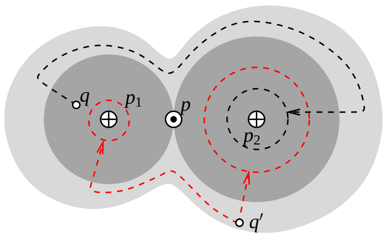


Figure 35: Examples of two queries.

For an ascending query message towards say peak p_2, if w has ascending pointers to p_2, this query message is simply delivered by gradient ascent routing, as the query from q' shown in Figure 35. If not, then the query message will follow a contour at a random value (below $\mathcal{F}(p)$ and above $\mathcal{F}(w)$) and follow the index-decreasing path, in order to cross the ridge and discover some ascending paths to p_1.

For a descending query message towards a merge saddle p from one peak p_1, this query message will hit the saddle contour $C(p)$, from which it learns the value range outside $C(p)$ and the value range inside $C_2(p)$. Again dependent on the query value x, the query message may split into two, one for the exterior of $C(p)$ and continuing to go downhill, and one for the interior of $C_2(p)$ and going uphill. See the black query in Figure 35 for an example. The query looking for peak p_2, again, may not have an ascending pointer immediately. Then the query will take a random contour in between $C(p)$ and the lower critical contour, follow the index-decreasing direction until it discovers the ascending pointer to p_2. In both contour-following routing, we may go a random number of hops further after ascending pointers are discovered, in order to avoid always using the nodes on the ridge.

This two scenarios basically cover all the details of gradient routing. All other cases are symmetric and omitted here. The main idea is to send the query message along the contour tree, possibly splitting at internal branches, and discover all components of the iso-contour of interest. To summarize,

- The gradient routing algorithm is completely local and distributed and successfully finds *all* contour components at a given query level.
- Every step of the routing algorithm is *justified*, we send a query message only when we are sure there is something to be found. So no message will end up in vain.
- The routing scheme does not have to go through the saddles or follow critical contours, thus does not overload those nodes.

We note that this iso-contour query is the most basic query of a family of queries on iso-contours. Other iso-contour queries include: reporting the number of contours at value x, in particular, is there a single contour component? Range-limited queries (count/report contours within a value range)? These can be handled with either the iso-contour query as a subroutine, or by using a similar gradient routing algorithm. We omit the details here as the extension is relatively straightforward.

3.5.3 Restricted range routing

In this section we consider the problem of finding paths within a restricted range of values in the network.

Routing request - nodes(s,t), range$[a,b]$: Find a path $\mathcal{P}$ between nodes s and t such that at every node x on $\mathcal{P}$, $a \leq \mathcal{F}(x) \leq b$, abbreviated as $a \leq \mathcal{F}(\mathcal{P}) \leq b$.

Keeping in mind that every node p its contour component $C(p)$ are mapped to the same point in the contour tree by the retraction $\mathcal{R}$, we state the following theorem:

Theorem 32 *A suitable restricted range path exists in the network if and only if a corresponding path exists in the contour tree.*

Proof. The proof is simple when s and t are on the same edge of the contour tree. In the following, we prove the case where they are mapped to different edges, and $\mathcal{F}(s), \mathcal{F}(t) \in [a,b]$.

A path in the contour tree implies a path in the network. This is easy, since a path in the tree can be realized as a set of ascending/descending paths in the network. Simply following the path in the tree from s to t will construct the required path in the network.

A path in the network implies a path in the tree. We prove the case only for $\mathcal{F}(\mathcal{P}) \leq b$, the other inequality follows symetrically. Suppose for contradiction that there is a suitable path $\mathcal{P}$ in the network, but no such path exists in the contour tree. We represent the maximum height on the path by $\max(\mathcal{P})$. Let $\mathcal{P}'$ be the unique path in the tree between images of s and t in the tree. Then by our hypothesis, $\max(\mathcal{P}) \leq b < \max(\mathcal{P}')$. Since the contour tree is produced by a retraction $\mathcal{R}$, and $\forall r, \mathcal{F}(\mathcal{R}(r)) = \mathcal{F}(r)$, the path $\mathcal{P}$ is mapped to the tree as a connected graph $\mathcal{R}(\mathcal{P})$ with $\max(\mathcal{R}(\mathcal{P})) < \max(\mathcal{P}')$ and spanning s and t. Thus, $\mathcal{R}(\mathcal{P})$ must contain $\mathcal{P}'$. We can now remove all extraneous edges leaving $\mathcal{R}(\mathcal{P}) = \mathcal{P}'$, with $\max(\mathcal{P}') = \max(\mathcal{R}(\mathcal{P})) = \max((\mathcal{P}))$ contradicting the hypothesis. □

Lemma 33 *In any subtree T' of a contour tree there is a node whose removal divides the subtree into* 3 *connected components each containing at most* $2/3$ *the number of nodes in T'.*

Theorem 34 *A contour tree admits a labeling scheme of label size at most $O(\log n)$, such that the max value on the path between any two nodes in the tree can be derived from their labels.*

3.6 Discussions

We study the problem of contour tracking with binary sensors and propose a light-weight distributed algorithm that locally repairs broken contours as they deform, while guaranteeing that the maintained contours capture the global topological features. We focus on information processing and topology maintenance aspects of the problem. Furthermore, we presented the distributed construction of a contour tree and its application in iso-contour queries by gradient routing with guaranteed delivery. For future work, we plan to explore further the applications of the contour tracking algorithm in processing dynamically changing spatial sensor data. One direction is to combine it with our concurrent work in contour tree [108] to construct a distributed dynamic contour tree for guided navigation.

Chapter 4

Decouple Design from Deployment

We studied the topology of a signal field in Chapter 3, then study the topology of the network field and its impact on information processing in this chapter.

4.1 Introduction

In most scenarios, it is infeasible to carefully deploy thousands of sensor nodes in a pre-planned organized way, due to unforeseen obstacles, poor accessibility, and possible changes in the environment, etc. Sensor nodes are typically randomly thrown into the domain to be monitored (e.g., dropped from an aircraft), and start with no knowledge of the big picture, such as its relative position in the network, or the global shape of the field to be monitored. The diversity of the deployment settings comes naturally from the diversity of geographical features of the underlying environment, and has essential influence on network design. It is thus desirable to automate the network design process and let the sensor nodes self-organize into a properly functioning network and carry out required tasks in an automatic manner.

The geometric properties of a sensor field represent an important character of the network, as sensor nodes are embedded in, and designed to monitor, the physical environment. First of all, the physical locations of sensor nodes impact on the system design in all aspects from low-level networking and organization to high-level information processing and applications. Clearly sensor placement affects connectivity and sensing coverage, which subsequently affects basic network organization and networking operations. Recently a number of research efforts have identified the importance of not only sensor locations, but also the global geometry and topological features of a sensor field. The 'topology' here means algebraic topology and refers to holes or high-order features. In the literature, uniformly random sensor deployment inside a simple geometric region (i.e., without holes) is arguably the most commonly adopted assumption on sensor distribution — but is rarely the case in practice. The real distribution usually adds specific requirements and constraints to the network design. These deployment specifics also play an important role in the selection of different protocols and the calibration of protocol parameters. Many algorithms and protocols proposed for a dense and uniform sensor field inside a simple geometric region, may have degraded performance when they are applied to an irregular sensor field with holes, etc.

Let us use routing as an example. Geographical routing, in which a packet is greedily forwarded to the neighbor that is geographically closest to the destination [64, 120, 121], has attracted a lot of interests. It is simple, elegant, and has little routing overhead. In a dense and uniform sensor field with no holes, geographical forwarding produces almost shortest paths and is robust to link or node failures and location inaccuracies. However, when the sensor field is too sparse, has holes or a complex shape, greedy forwarding fails at local minima. This is due to a mismatch of routing/naming rules with the real network connectivity. Two nodes that are geographically close may actually be far away in the connectivity graph. Local face routing can get the message out of a local minima but often incur high load on nodes along hole boundaries. To achieve load balanced routing when these topological features (e.g., holes) become prominent, the naming and its coupled routing protocol should represent the real network connectivity and adjust to these topological features accordingly [89, 122].

The global topology of a sensor field also has fundamental influence on how information gathered in the network should be processed, stored and queried. In a sensor field with narrow bottlenecks, more aggressive in-network processing is expected to minimize the traffic flowing through bottleneck nodes. In a centralized storage scheme, one or a few base stations are typically placed in the sensor field. These base stations are much more powerful than sensor nodes and they serve as data processing and storage centers, as well as gateway nodes through which users access the sensor data. Since communication is the major source of energy consumption, it is desirable to find a good placement of base stations such that the average distance from a sensor to its nearest base station is minimized and that no traffic bottleneck is created during the data delivery from sensor nodes to their respective base stations.

In a distributed storage scheme, the global geometry should be taken into account to achieve better load balance on storage nodes. Many existing information processing algorithms do not account for the global geometry of a sensor field yet. A typical example is the quadtree type of geometric decomposition hierarchy, which has been extensively used to exploit spatial correlation in sensor data (e.g., DIFS, DIM [71, 88]) for efficient multi-resolution storage. In a sensor field with holes, a standard geometric quadtree (the bounding rectangle is partitioned into four equal-size quadrants recursively) may become unbalanced with lots of big empty leaf blocks. An imperfect partition hierarchy subsequently affects the performance, especially load balance, of all algorithms and data structures built on top of it. In another example of geographical hash tables (GHTs) [34], a random rendervous node is chosen to hold the data of a certain type for users to query. Random sampling of a sensor node can be conducted by choosing the node closest to a random location. To achieve a uniform distribution, the sampling probability needs to be adjusted by the area of the corresponding Voronoi cell [123, 124]. In an irregular sensor field, the Voronoi cells have vastly varying areas. Thus the sampling efficiency suffers as a lot of trials end up being rejected.

One approach to deal with irregularly shaped sensor field is to develop virtual coordinates with respect to the true network connectivity, as in the case of routing [89, 122] or information storage and retrieval [75]. One may follow this line and re-develop algorithms for all the other problems on virtual coordinate systems. But both the development of virtual coordinates and topology-adaptive algorithms on top of that are highly non-trivial. We propose to develop a unified approach to handle complex geometry, in particular, a segmentation algorithm that partitions an irregular sensor field into nicely shaped pieces such that algorithms that assume a uniform and dense sensor distribution can be applied inside each piece. Across the segments, problem dependent structures specify how the segments and data collected in these segments are integrated. There is not much prior work on segmenting a sensor field. The mostly related one, by Kröller *et al.* [101], proposed a boundary detection algorithm with which one can organize sensor nodes by 'junctions' and 'streets'. Our goal is to further explore segmentation algorithms suitable for a discrete sensor field as well as applications that can benefit from it.

4.1.1 Challenges and Our Approach

We consider a static sensor network with an irregular shape. We take the viewpoint to regard the sensor network as a discrete sampling of the underlying geometric environment and develop a 'shape segmentation' algorithm. Although the analysis of geometric shapes has been extensively studied in graphics and computational geometry with many shape segmentation algorithms proposed in the literature [125–129], these algorithms typically work in a centralized setting with ample computational resources. Shape segmentation problem for a discrete sensor field faces a number of new challenges, and requires non-trivial algorithm design to achieve sufficient robustness to input inaccuracies.

- Sensor nodes start with no idea of the global picture. We consider the approach of collecting all information at a centralized node not a scalable solution. Segmentation algorithm needs to be automatic and distributed in nature.
- Sensor nodes may not know their geographical locations. Automatic and scalable localization (without GPS) is still a challenging problem. Even when they do, the locations may come with large inaccuracies.
- When sensor locations are not available, the distance between two nodes is often approximated by their minimum hop count value, which is always an integer. This rough approximation introduces inevitable noise to any geometric algorithms that use the hop count to replace the Euclidean distance.

We propose to adapt a shape segmentation scheme by using flow complex [125] to sensor networks. The algorithm uses only the connectivity information and does not assume that sensors know their locations. We first discover all the hole boundaries and the outer boundary. This can be done with any existing boundary detection algorithm. Indeed efficient boundary detection algorithms have been proposed with only the connectivity information [81, 97–101, 130]. We use the output of Wang *et al.* [81] in our segmentation algorithm. We let the boundary nodes flood inward and every node records the minimum hop count from the boundary. Each node is then given a 'flow direction', the direction to move away fastest from boundaries. A node may be singular with no flow direction and is named as a *sink*[1]. The sensor field is partitioned to segments in a way that nodes in the same segment flow to the same sink. This naturally partitions the sensor field along narrow necks. In the geometric version, all the sinks stay on the medial axis of the field, which is the set of points with at least two closest points on the boundary and constitutes a 'skeleton' of the shape. In a discrete network, sinks may appear far away from the medial axis due to local noises and connectivity disturbances. In addition, in degenerate cases such as a corridor with parallel boundaries, many nodes on the medial axis may be identified as sinks. We apply a local merging process such that nearby sinks along the medial axis with similar hop counts from the boundary, together with their corresponding segments, are merged. In the end, each segment is given a unique identifier by the sink(s). All the nodes in the same segment are informed of the identifier distributed along the reversed flow pointers. The algorithm is communication efficient. It involves a couple of limited flooding from the boundary nodes to the interior of the network. All the other operations only involve local computations. With given boundaries, the segmentation algorithm incurs a total transmission cost of $O(n)$ if nodes synchronize during message transmissions. We tested the segmentation algorithm under various topologies and node densities. We observed intuitive segmentation along narrow necks in a sensor field with reasonable average node degree (around $7 \sim 8$).

The segmentation algorithm is expected to run at the initialization stage to aid network design and the calibration of network protocols. The understanding of the global topology and in particular, the automatic grouping of the sensor nodes into segments with simple shape, provides a generic approach to handle sensor field with different node distribution. This enables the re-use of existing protocols on an irregular network and makes the development of new protocols transparent to the specifics of the shape of a sensor field. We have studied the influence of global

[1]Notice that the sink we refer to is not a data sink or aggregation center (base station), although the sinks are good indicators of where to place base stations or aggregation centers. We discuss the problem of base station placement in Section 4.5.2.

topological features on various fundamental problems in network design, e.g., routing, base station placement, data storage and uniform sampling, and evaluated the performance improvements by integrating segmentation algorithm with existing algorithms that currently assume a simple geometric sensor field.

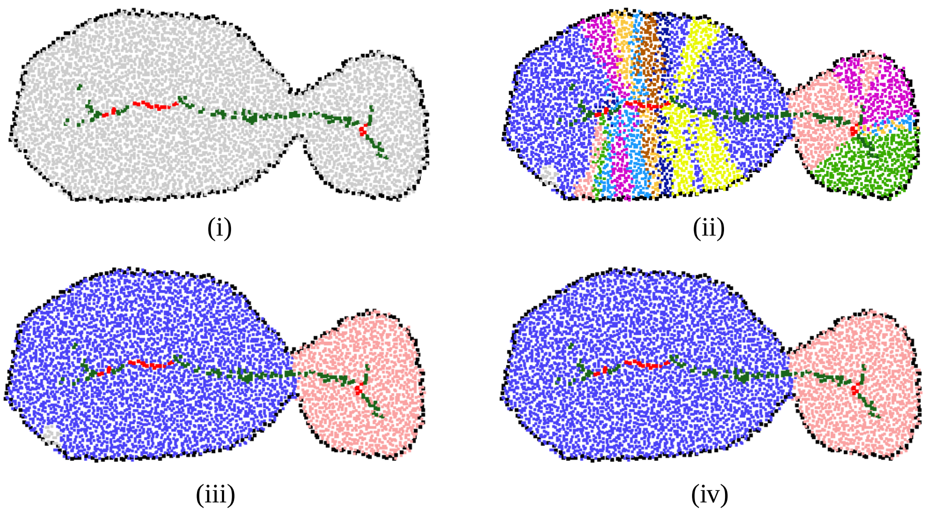


Figure 36: The fish network. 5000 nodes, generated by grid-perturbation distribution with variation. Avg. degree is 8. Boundary nodes are shown in black. (i) Medial-axis nodes shown in dark green. Sink nodes shown in red. (ii) The stable manifolds of the sink nodes, shown in different colors. (iii) Nearby sinks with similar hop counts to the boundary, along with their stable manifolds, are merged. Orphan nodes shown in grey. (iv) The final result after processing orphan nodes.

4.2 Segmentation in Continuous Domain

We first introduce some notations and definitions defined in the continuous domain [125]. In the next section we show how to adapt them in a discrete network. For a connected continuous region $\mathcal{R}$, denote by $\mathcal{B}$ its boundaries, represented by a set of closed curves, each bounding either an inner hole or the outer boundary. For a point $x \in \mathcal{R}$, the distance from x to the boundaries is define by function $h(x) = \min\{||x-p||^2 : p \in \mathcal{B}\}$. The *medial axis* is the set of points in $\mathcal{R}$ with at least two closest points on the boundary. The distance function h is continuous, and smooth everywhere except points on the medial axis. We call a point x a critical point, or a *sink*, if x is inside the convex hull of its closest points on $\mathcal{B}$, denoted by $\mathcal{H}(x)$. For example, sink s_1 has three closest points on the boundary and stays inside the triangle spanned by them. All non-critical points are called regular. The *driver*, $d(x)$, is defined as the closest point in $\mathcal{H}(x)$ (e.g., in Figure 37, the driver of p_1 is the smaller black dot). For a sink, the driver is itself. Now the *flow* is defined as a unit vector $v(x) = \frac{x-d(x)}{||x-d(x)||}$ (i.e., the direction that points away from its driver), if $x \neq d(x)$ and 0 otherwise. It has been proved in [125] that the flow direction follows the greatest descent of the distance function h. There are also a few easy observations of the flow vectors, as stated below.

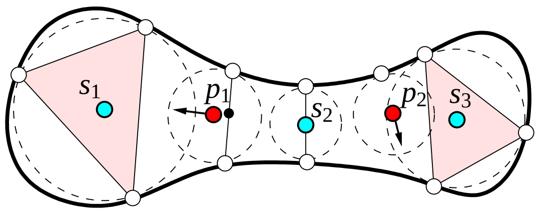


Figure 37: Two regular points (p_1 and p_2) with their flow vectors. Sinks (s_1, s_2 and s_3) stay inside the convex hull of their closest points on the boundary.

Observation 35 *For a connected continuous region $\mathcal{R}$ with boundary $\mathcal{B}$, the following holds.*

- *All sinks must stay on the medial axis.*
- *Any point p not on the medial axis will have a unique driver which is its closest point on the boundary. Thus p flows towards the medial axis.*
- *All the points will eventually flow to sinks.*

The *stable manifold* of a sink x, denoted as $\mathcal{S}(x)$ is simply the set of points that flow to it by following the flow directions. Our segmentation algorithm will partition the network by the stable manifolds. The discussion below concentrates on the properties of the generated segments in the continuous setting that are helpful for our algorithm development in the discrete setting.

4.2.1 Properties of generated segments

If all points flow to the same sink, the entire region becomes one stable manifold. Thus, there is at least one segment in any region. In Figure 37 there are a total of three sinks and two large stable manifolds (the stable manifold for s_2 is a degenerate segment). Sinks s_1 and s_3 correspond to local maxima of the distance function $h(x)$, sink s_2 is a saddle of $h(x)$. Rigorously, we define a *degenerate sink* to be the sink with only two closest points on the domain boundary and the rest of sinks as *non-degenerate*. We first look at properties of non-degenerate segments.

In a segment, the sink can be considered, to some extent, a 'center' of the segment. To understand what these segments are, we realize that they map to balls of maximal size. Rigorously, we have the following theorem. Define a ball centered at a point to be *locally maximal* if the ball is entirely inside the region $\mathcal{R}$ and by moving the center of the ball infinitesimally one cannot enlarge the size of the ball.

Theorem 36 *All locally maximal balls are centered at sinks.*

Proof. Consider a locally maximal ball B centered at node x. If x is not a sink, it must have a flow direction. Then B will become larger if the center of the ball is shifted a small distance in the direction of the flow, because the distance function increases along the follow direction. This contradicts to the fact that B is locally maximal. So x must be a sink. $\square$

This theorem gives some properties of the non-degenerate segments induced. Intuitively, we want to obtain a few number of large and 'fat' pieces. One way to measure the fatness of a segment $\mathcal{S}(x)$ is to consider the largest ball, completely inside $\mathcal{R}$, centered at a point inside $\mathcal{S}(x)$ against the minimum circumscribed ball of $\mathcal{S}(x)$ [2]. Theorem 36 says that we obtain k non-degenerate segments, in each segment the largest ball centered inside is exactly the locally maximal ball centered at the sink of this segment. If we merge two non-degenerate segments, the fatness of the resulting segment will be hurt since the size of the largest ball centered inside the segment does not increase but the size of the minimum circumscribed ball may increase. One may improve the fatness of segments by reducing the sizes of the minimum circumscribed balls, at the expense of introducing more segments.

[2]Note that the largest ball centered at a point inside the segment is only required to be completely inside $\mathcal{R}$ and may actually intrude outside this segment.

A second property of a non-degenerate segment is that it is simply connected and does not have holes.

Theorem 37 *A non-degenerate segment is simply connected and does not contain holes.*

Proof. The proof is by contradiction. Suppose there are one or more holes inside a non-degenerate segment S which has only one critical point (sink). Since the medial axis is a deformation retract of the domain $\mathcal{R}$ [131, 132], the medial axis has cycles corresponding to the holes inside S. All the points not on the medial axis will follow flow directions towards the medial axis, upon reaching which the flow directions follow the medial axis to reach the sink. Thus, inside the segment S we have the same number of cycles on the medial axis corresponding to the holes of S. Let us just focus on one such cycle C on the medial axis (and one hole H). All points on this cycle flow to the sink in this segment.

On the cycle C, we examine the distance function to the network boundary h. Note that h is continuous function and C is compact. Let's consider the point p and point q on C on which h reaches maximum and minimum on the cycle C respectively. The local maximum p is either a sink, or flows to a sink that lies outside C, but in either case, it is the sink at which all flows of S converge. Now we argue that the local minimum q is a degenerate sink also inside S, which will show contradiction to the claim.

There are two possible cases for the point q. q is either a junction on the medial axis, or a regular point on the cycle C. If q is a regular point, then q is in fact a saddle of the distance function h, as along the line segments connecting q to each of its two closest points on the boundary q_1, q_2, the distance function will decrease; and along the medial axis, the distance function increases. Now we argue that q must stay on the line segment connecting q_1q_2, thus proving q is a degenerate sink. If otherwise, then location of q will not coincide with its driver there will be a direction along the medial axis in which distance to the boundary decreases. This contradicts with the fact that q is a local minimum on C. An example is shown in Figure 38 (i).

If q is a junction point and a minimum, q stays outside the convex hull (i.e., the triangle) formed by its 3 closest points q_1, q_2, q_3 on the region boundary. Depending on where the driver stays, by moving infinitesimally along the three branches of the medial axis that join at q, there is only one direction that leads to an increasing distance away from the boundary — when moving away from the driver. See an example in Figure 38 (ii). On the other hand, since q is a local minimum on C there are two directions, moving along which will lead to increasing distances from the boundary. This is a contradiction, implying that q cannot be junction point on the medial axis.

In summary, any non-degenerate segment can not contain a hole inside. □

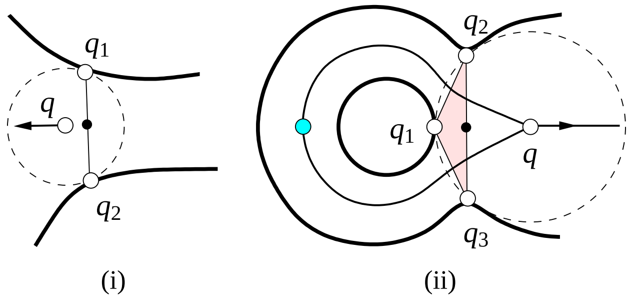


Figure 38: A non-degenerate segment can not contain holes. (i) when the minimum q is a regular point; (ii) when the minimum q is a junction.

The above two theorems show that the non-degenerate segments are simply connected, locally maximally fat segments. Now we look at degenerate segments and in fact we will need to merge them to come up with nicely defined fat segments.

For degenerate segments, their fatness, according to our first definition is already 1 — as the maximum ball centered inside a degenerate segment is actually the same as the minimum circumscribed ball, although most of this ball is intruding outside the degenerate segment. However, this is not what we want in a meaningful segmentation. In particular, when there are parallel boundary segments, there can be infinitely many degenerate segments, each being a line segment perpendicular to the boundary. We thus propose to merge them into segments of significant size.

To do that, we consider a second way to define the fatness of a segment as the ratio of the radius of the maximum inscribing ball and that of the minimum circumscribing ball. When we restrict the maximum ball to be completely inside the segment (not just be within the region $\mathcal{R}$), then the fatness of a non-degenerate segment becomes 0. Now we would like to merge nearby degenerate segments to improve the fatness (in the second definition). As a remark on the two definitions of fatness, the first definition captures the local geometry and curvatures and identifies the real bottlenecks of the shape; the second definition, though being more widely adopted in the past literature, is here used more as a rescue to handle degenerate segments.

For degenerate segments, the fatness measure in the second definition suggests that two degenerate segments should be merged if this improves the fatness of the segments. This represents one option in our algorithm to decide whether two segments should be merged or not. It is an automatic procedure to find a small number of fat segments such that merging any two will hurt the fatness. We will discuss the details of the implementation of this idea in the discrete network in Section 4.3.4.1.

With the above segmentation algorithm, we prove that non-degenerate segments and segments merged based on fatness measure do not contain holes inside. Again, the following theorem considers the continuous case only.

Theorem 38 *Segments merged from degenerate critical points based on fatness measure do not contain holes inside.*

Proof. Consider a segment formed by merging multiple degenerate segments together. Suppose for the sake of contradiction and w.l.o.g that all these segments are merged into a segment S with a hole in the middle.

First of all, we only merge degenerate segments. If the resultant segment S contains a hole, it must be that the cycle on the medial axis of S corresponding to this hole will have all the points on it as degenerate sinks. This says that S must an annulus — as a strict maximum (strictly greater than the minimum) of the distance function $h(x)$ on this cycle is a non-degenerate sink. Suppose the width of the annulus is W, the outer circumference has length F. Then $F \geq 2\pi W$. And the length of medial axis is $\geq \pi W$. When the merging process produces a segment that contains a part of the medial axis longer than W, this segment does not participate in merging any more, since the fatness will be reduced with further merging. Thus it is not possible to have a single segment covering the entire πW length cycle. $\square$

Last we remark that the properties we prove about the segments produced by the segmentation algorithm hold for now only in the continuous case. When we have a discrete network, the idea is to use the same intuition and the goal is to develop lightweight algorithm to automatically segment the network in a fully distributed manner.

4.3 Distributed Segmentation Algorithm in Sensor Networks

The flow and stable manifolds described in section 4.2 naturally partition a continuous domain into segments along narrow necks with each stable manifold as a segment. In this section we show how to implement the flow and the segmentation in a discrete sensor field, when we do not have node location information, nor the distance function. Unlike the continuous case, here we approximate the Euclidean distance function to the boundary by the minimum hop count to boundary nodes. As for the notion of closest *points* on the boundary, an interior node x has one or more closest *intervals* — each interval is a consecutive sequence of nodes on the boundary with minimum hop count from x. We want to find, for each sensor x, a flow pointer that points to one of its neighbors, signifying 'fastest' movement away from the boundary. The challenge is to assign these flow pointers and identify the sinks in a robust way such that there is no loop and an intuitive segmentation can be derived. We describe an outline of the algorithm followed by the details of each step.

- **Detect boundaries.** Find the boundaries of the sensor field using the algorithm described in [81]. This algorithm identifies boundary nodes and connects them into cycles that bound the outer boundary and interior hole boundaries of the sensor field.
- **Construct the distance field.** The boundary nodes simultaneously flood inward the network and each node records the minimum hop count to the boundary, as well as the interval(s) of closest nodes on the boundary. Nodes on the medial axis can identify themselves as the closest intervals they have are not topologically equivalent to a segment (two or more intervals, a cycle, etc.).
- **Compute the flow.** Each node x computes a flow pointer that points to its *parent* — the neighbor with a higher hop count from the boundary and the

most symmetric closest intervals on the boundary among all such neighbors. Nodes on the medial axis with no neighbor of higher hop count become *sinks*. See Figure 36(i) and (ii).

- **Merge nearby sinks.** Nearby sinks on the medial axis with similar hop count from the boundary are merged into a *sink cluster* and agree on a single *segment ID*.
- **Segmentation.** The nodes that ultimately flow to the same sink cluster are grouped into the same segment. We let each sink disseminate the segment ID along the reversed *parent* pointers. See Figure 36 (iii) for the merged segments.
- **Final clean-up.** Due to irregularities in node distribution, some nodes have locally maximum hop count to boundary but do not stay on medial axis — thus did not get recognized as sinks. These, and nodes that flow into them are left *orphan*. In the final clean-up phase, we merge the orphan nodes to their neighboring segments. Figure 36 (iv) shows the final result.

4.3.1 Detect boundaries

The segmentation algorithm can use any existing boundary detection algorithm to detect boundaries. Here, we choose to use the boundary detection algorithm proposed by Wang *et al.* [81]. This algorithm requires only the connectivity information. The boundaries of inner holes and outer boundary are assigned unique identifiers, and nodes along each boundary cycle are assigned ordered sequence numbers. Every boundary node thus knows the identifier and the length of the boundary to which it belongs, and its own sequence number on that boundary. We refer to the set of nodes on boundary j as B_j.

4.3.2 Construct the distance field

With the boundary nodes identified, we construct a distance field such that each node is given a minimum ‘distance’ to the sensor field boundary. Since we do not assume location information, our only measure of distance in the network is the number of hops to the boundary nodes. The problem is that a node typically has more than one nearest boundary nodes with the same hop counts away (thus may be identified to be on the medial axis). Hence we keep not the *closest node* but the *interval* of closest nodes. An interval I on the j^{th} boundary cycle is simply a sequence of nodes along the boundary cycle. It can be represented uniquely by a 4-tuple $(j, start_I, end_I, |B_j|)$, where $start_I$ and end_I are the two end points, $|B_j|$ is the length of the j^{th} boundary.

We have the boundary nodes synchronize among themselves [133, 134] and start to flood the network at roughly the same time. The boundary nodes initiate a flood with messages of the form (I, h) where I is an interval nearest to the transmitting node, and h is the distance to nodes in I. Initially, I is set to the boundary node itself, and h is set to 0. A node p keeps track of the set S_p of *intervals* of boundary nodes nearest to it. On receiving a message (I, h), a node p compares h to its current distance h_p (h_p is initially set to infinite at non-boundary nodes) to the boundary:

- If $h > h_p$, discard the current message.
- If $h < h_p$, discard all existing intervals, set $h_p := h$, $S_p := \{I\}$ and send $(I, h+1)$ to all neighbors.
- If $h = h_p$, merge I with adjacent and overlapping intervals on the same boundary if there is any. Otherwise, simply add I into S_p. Send $(I, h+1)$ to all neighbors.

To tolerate the noises caused by hop count measure, we consider two hop counts h_1 and h_2 equal if $|h_1 - h_2| \leq d$. Simulation results show that $d = 1$ works well in practice. Thus, each node keeps all closest intervals to boundaries, and the hop counts of any two nodes included in the intervals are at most one hop different. In a special case, a node may have a single closest interval which covers the entire boundary cycle, e.g., the center of a circular disk. We also treat such special nodes as medial-axis nodes. More rigorously, after this computation of sets of nearest intervals for all nodes in the network, nodes on the medial axis can be identified as follows:

Definition 39 ***Nodes on the medial axis****: A node p is a medial-axis node if its set of closest intervals S_p are not topologically equivalent to a segment.*

Based on the above definition, a node with a single closest interval is not on the medial axis. But a node with two or more closest intervals (e.g., s_1 in Figure 37) or a cycle of boundary nodes (e.g., the center of a disk) is on the medial axis. Our definition of the medial axis differs from the one used in existing articles (e.g. [89]), in which a node is on the medial axis if it has multiple closest points on the boundary — in our definition we do not include nodes with multiple closest boundary points forming a single segment along the boundary. The purpose of this new definition is to further eliminate possible noisy medial axis nodes due to the discreteness of the network setting. For the difference of the two definitions, we can rigorously say something in the continuous case. In particular, they only differ at some limit points: for each node p with multiple closest points along a segment on the boundary, it is guaranteed to find a node q, infinitesimally close to p, such that q has multiple closest intervals (and thus identified in our definition). This following theorem implies that while our definition does not include terminal vertices of medial axis, it does include points arbitrarily close to it.

Theorem 40 *In a continuous domain $\mathcal{R}$ with boundary $\mathcal{B}$, for each node p with multiple closest points along a segment on the boundary, we can find a node q infinitesimally close to p, such that q has multiple closest intervals.*

Proof. Node p has multiple closest boundary points forming a segment $\widehat{s_1s_2}$ on the boundary. Consider the ball B centered at p and with radius $|ps_1|$ (see Figure 39). The ball must be completely inside the field and has $\widehat{s_1s_2}$ as an arc. The curvature at all points on the arc $\widehat{s_1s_2}$, including the endpoints, is $1/|ps_1|$. And the curvature at boundary points infinitesimally away from s_1, s_2, outside the arc, is strictly smaller than $1/|ps_1|$. Now, take q on the bisector of the two points s_1, s_2 and of ε distance away from p, with $\varepsilon \to 0$, the ball B' centered at q will have two closest intervals, pass through boundary points s_1 and s_2 respectively. $\square$

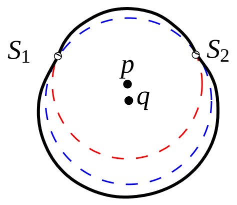


Figure 39: Node p is not on the medial axis, since it has a single closest interval; but p has a nearby point q on the medial axis.

What is the most appropriate analog of the continuous medial axis in a discrete network is yet to be debated. We find our definition more robust (produces fewer noisy medial axis nodes) and suitable for our purposes. Figure 36(i) shows the medial axis of the fish network found with this protocol. The protocol described above is easy to implement and works well in simulations. As pointed out in [89], if all boundary nodes initiate the flood at about the same time, then this method keeps the total communication cost very low. The distance field can also be constructed with two rounds of boundary flooding: in the first round each node records the hop count to the boundary; in the second round each node broadcasts their closest intervals. To simplify the theoretical analysis of message complexity, we use the two-round version in Section 4.3.7.1.

4.3.3 Compute the flow pointer

Once each node p learns its minimum distance h_p from the boundaries and the intervals of closest nodes at distance h_p from it, it can construct the flow pointer and find sinks locally. Observe that for any pair of neighboring nodes p and q, if $h_p < h_q$, then the closest intervals of p must be included in the closest intervals of q. Rigorously, $\forall I \in S_p, \exists I' \in S_q$ such that $I \subseteq I'$.

Each node creates a *flow pointer* to its *parent*, the neighbor who is strictly further away from the boundary than itself, and whose closest intervals are most *symmetric* with respect to its own closest intervals. In Figure 40, node p selects node b as its parent $v(p)$ because the mid point of b's interval is closer to the mid point of its own interval. The intuition behind this is to select the neighbor that represents the best movement away from all parts of the boundary. We make this notion rigorous by defining the angular distance of two neighboring nodes.

Definition 41 ***Mid point****: The mid point $mid(I)$ for an interval I defined on the boundary j, is the $\frac{|I|+1}{2}$-th element in the continuous sequence modulo $|B_j|$ of I, if $|I|$ is odd, else it is the mean of the $(\frac{|I|}{2})$-th and the $(\frac{|I|}{2}+1)$-th elements.*

Definition 42 ***Angular distance****: The angular distance $\delta(p,q)$ between neighboring nodes p and q with $h_p < h_q$ is defined as:*

$$\delta(p,q) = \sum_{I \in S_p} \min_{I' \in S_q} |mid(I) - mid(I')|$$

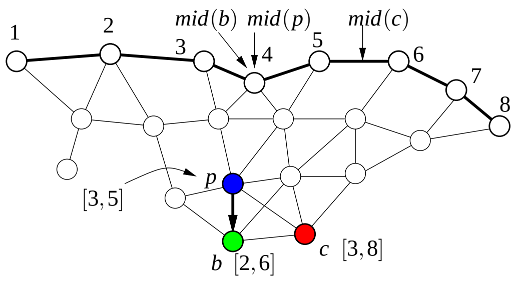


Figure 40: Node p selects node b as its parent $v(p)$, as b is the more symmetric neighbor. The closest intervals of node p, b, c are shown.

I and I' must be on the same boundary.

Using the function δ, each node p selects a neighbor q such that $h_p < h_q$, and the sum of distances from the mid points of its intervals to the corresponding intervals of q is less than that for any other neighbor of p.

Definition 43 ***Flow pointer**: Let H_p be p's neighbors with higher hop count from the boundary, i.e., $h_p < h_q$, for $q \in H_p$. Then the parent of p, $v(p)$ is defined as the neighbor in H_p with minimum angular distance, $v(p) = \arg\min_{q \in H_p} \delta(p,q)$.*

A typical node, for example p in Figure 40 would have only one boundary interval nearest to it. Nodes on the medial axis have more than one such interval, in which case, the parent is chosen based on the sum of mid point distances instead of a single distance, as described above.

Definition 44 ***Sink**: A node c is a* sink, *if c is a medial-axis node, and has locally maximum hop count from the boundary.*

The sinks of the fish network, by this definition, are shown in Figure 36(i). Sinks are those medial-axis nodes without a parent. With the flow, the sequence of directed edges starting at any non-sink node p ends at a unique root c. Since a node p selects its parent only if its parent has a higher hop count from the boundary, there cannot be a cycle in the directed graph implied by the flow. Thus, the nodes in the network are organized into directed forests, with the nodes in the same tree flow to the same root by following their flow pointers. In a continuous domain, the *stable manifolds* of the sinks form the segments. In a discrete network, the analog of the *stable manifold* of a sink c would be the directed tree rooted at c, such that any directed path in this tree ends at c.

4.3.4 Merge nearby sinks

The stable manifolds of the sinks, i.e., the trees rooted at the sinks, can be directly taken as the segmentation. This works fine with non-degenerate stable manifolds. However, there can be possibly a lot of degenerate stable manifolds and this may result in a heavily fragmented network (see Figure 36(ii)). This happens when there are a cluster of sinks on the medial axis, and there is a tree rooted at each. This situation becomes severe when some parts of boundaries run parallel to each other. See Figure 41(i). In this case we have a sequence of nodes on the medial axis where no node is farther from the boundary than its neighbors, and all these nodes become sink nodes.

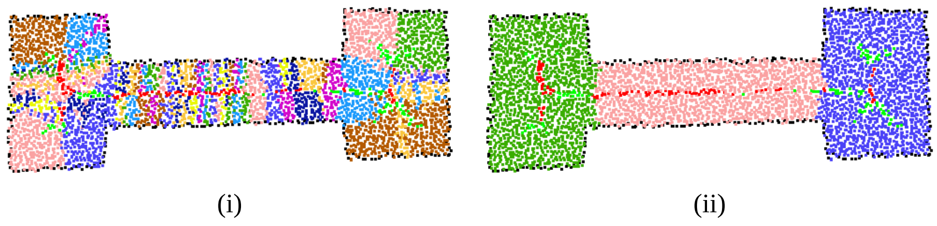


Figure 41: The corridor network. (i) Opposite boundaries run parallel to each-other, producing several sinks in succession, resulting in the fragmented segmentation. (ii) Segmentation with threshold-based merging.

We would like to merge nearby sinks with similar distances from boundary as well as their corresponding segments. We call the merged sinks a *sink cluster*. We propose two merging schemes here. In the first scheme, the sink of each segment locally maintains the fatness measure of its segment and chooses to merge with other segments only if merging helps to improve the fatness. Thus, this schemes automatically generates sufficiently ‘fat’ segments and users are hidden from the implementation details. When applications have specific requirements, for example, a few number of large segments, users may want to get involved to control the result. For that purpose, we propose the second scheme to merge nearby sinks together with their segments based on a user-defined threshold t. In the following, we discuss the details of these two merging schemes.

4.3.4.1 Merge by fatness

The fatness-based merging scheme consists of the following three steps. The sinks of the segments take responsibility of the following computation. Note that to handle degenerate segments we define fatness in the second definition as the ratio of the radius of the maximum inscribing ball and that of the minimum circumscribing ball.

- Measure the fatness of the segment by recording the radii of the largest inscribed ball and minimum enclosing ball.
- Identify if a segment is degenerate or non-degenerate.
- Merge nearby segments based on the fatness measure.

At first, each sink maintains the fatness of its segment. We use the hop count distances of the closest and the furthest nodes on the segment boundary to estimate the radii. A node identifers itself on the segment boundary if at least one of its neighbor resides in a different segment. Nodes on the segment boundary send messages through flow pointers towards the sink, so that sink nodes can record the smallest and largest hop count to approximate the radii of the largest inscribed ball and the minimum enclosing ball.

A sink recognizes its segment as a degenerate segment if the radii of the largest inscribed ball is small (e.g., the smallest hop count of nodes on the segment boundary is less than 2) or much smaller than that of the minimum circumscribed ball. Otherwise, the segment is a non-degenerate segment. Theorem 36 suggests that merging two non-degenerate segments will hurt fatness. So in this scheme, we only allow degenerate segments to merge with other segments.

The sink of a degenerate segment walks along the medial axis to search for other sinks to merge (in a sparse network when the medial axis is not connected, we search 2 or 3 hop neighbors for nearby medial-axis nodes). The fatness of the merged segment is measured as follows. The radius of the largest inscribed ball r_m is set to $(r_1+r_2+h(s_1,s_2))/2$, wherein r_1 and r_2 are the radii of the largest inscribed balls of original segments and $h(s_1,s_2)$ is the distance (approximated by hop counts) between the corresponding two sinks. The radius of the minimum circumscribed ball of the resulted segment is $R_m = max(R_1,R_2,r_m)$, where R_1 and R_2 are the radii of the minimum circumscribed balls of the original segments. Two sinks with their segments merge together if the fatness of both segments increase[3]. If two segments decide to merge, the sink of the larger segment becomes the leader of the merged sink cluster and takes responsibility of further merging.

It is possible that a sink can receive multiple merge requests at the same time. All requests are processed sequentially. Depending on the order of the merge, the resulted set of segments can be different. There are possibly thin segments left because all adjacent segments are fat enough and further merge will hurt their fatness. But, Theorem 45 guarantees that the number of segments is at most twice plus one the number of segments generated by the optimal algorithm based on fatness measure, and at least half of them have fatness at least half of the maximum. Examples of segmentation based on fatness measure are showed in Figure 42.

Theorem 45 *The number of segments produced by merging degenerate segments is at most one plus twice the number of segments generated by the optimal algorithm based on fatness measure. The fatness of at least half the segments is at least half of the maximum.*

Proof. The method above operates only on regions of degenerate sinks. It does not involve non-degenerate sinks, so we can safely leave those out of consideration. We then show that the claim holds for any connected sequence of degenerate sinks, hence for the overall network.

First of all, by merging the degenerate segments, one can verify that the fattest one can possibly get is a square with fatness $1/\sqrt{2}$. Consider a maximal connected set of degenerate sinks $\mathcal{S}$. Each segment in $\mathcal{S}$ must be of same width, say W. Let the length spanned by all segments in $\mathcal{S}$ be L. The optimal segmentation clusters $\mathcal{S}$ into at least $\lfloor L/W \rfloor$ segments each of fatness at most $1/\sqrt{2}$. Now, consider adjacent segments S_i, S_j after clustering. If l_i, l_j are the lengths of these segments respectively, then $l_i + l_j \geq W$, otherwise these would have been merged. Thus, this produces at most $2\lfloor L/W \rfloor + 1$ segments.

[3]Depending on applications, this condition can be relaxed to allow to merge as long as the fatness of at least one segment is improved, which allows a degenerate segment to be possibly merged with a non-degenerate segment. This will reduce the number of segments and remove thin segments, but may hurt the fatness of non-degenerate segments. We stick to the original condition in the following discussion

Also observe that since $l_i + l_j \geq W$, at least one of each such pair l_i, l_j is larger than $W/2$. The fatness of this segment would be $\geq 1/\sqrt{5}$. Which is more than half of $1/\sqrt{2}$. Thus at least half the segments are half as fat as the maximum fatness. $\square$

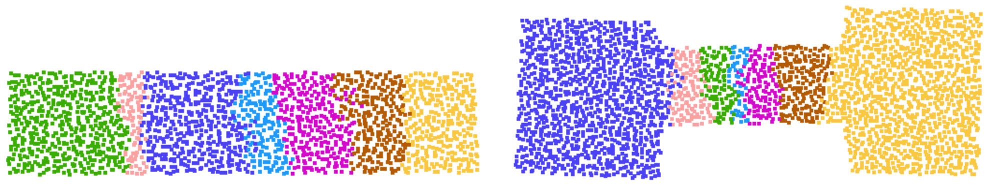


Figure 42: Segmentation with fatness-based merging. (i) The rectangular network; (ii) The corridor network.

4.3.4.2 Merge by user-defined threshold

The above fatness-based merge scheme guarantees that the generated set of segments are fat enough, and makes the whole process automatic and hidden from the users. However, some applications may have specific requirements on the segments other than the fatness criteria. For example, users may want to be involved in controlling the total number of segments generated. To satisfy that, we propose a merge scheme based on user-defined threshold t. We want to merge nearby sinks with similar distances from the boundary and cluster those sinks into a sink cluster. Here, a sink cluster K is represented by the tuple $(id, h_{\max}, h_{\min})$, where id is the minimum ID of all the sinks in the cluster, i.e., the ID of the *leader* of the cluster. $h_{\max}$ and $h_{\min}$ are the maximum and minimum distances from any sink to the boundary respectively. We set a user-defined threshold t to guarantee that $|h_{\max} - h_{\min}| \leq t$. Each sink node waits for a random interval to start the merging process to avoid contention. Initially each sink is by itself a sink cluster, and also a sink cluster leader. A sink cluster leader searches along the medial axis for nearby sinks (or sink clusters) to be merged. Specifically, a sink cluster leader c sends a *search message* of its current sink cluster $(id, h_{\max}, h_{\min})$ to all neighboring nodes on the medial axis. Each medial-axis node p of cluster $(id', h'_{\max}, h'_{\min})$, on receiving this message, executes the following rule (let $H_{\max} = \max(h_{\max}, h'_{\max})$ and $H_{\min} = \min(h_{\min}, h'_{\min})$) :

- if $|H_{\max} - H_{\min}| > t$, discard the message.
- else forward the message to all neighboring nodes on the medial axis. Furthermore, if p is a sink cluster leader, p would like to merge its sink cluster with c's sink cluster.

When two segments merge, the sink with the smaller ID becomes the leader of the sink cluster and sends out a new search message looking for sink clusters to be merged. The process terminates automatically when no merging request is received after a timing threshold.

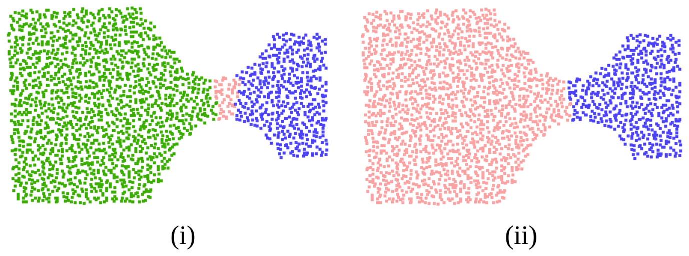


Figure 43: Network with 2200 nodes, with avg 6 neighbors per node. Segmentation with threshold (i) $t = 2$; (ii) $t = 4$.

The variable t is a threshold defined by the user. It determines the granularity of segmentation. Smaller values of t imply that merging step will stop at a small change of the distance from the boundary and hence collect fewer sinks together into sink clusters. Thus there will be more segments created by the algorithm. For larger values of t, the algorithm will create fewer and larger segments. Figure 43 shows the differences caused by variation in the value of t. In many such situations, the most preferable segmentation is likely to be dependent on the nature of the application. We leave it to the user's discretion to set the value of t. Note that the user can change the value of t even after the network has been deployed by flooding a message from any one node in the network. This would require a re-computation of only the last three steps of the algorithm, starting at merging of sinks.

4.3.5 Segmentation

Each sink cluster defines a segment, as all the trees rooted at nodes in the sink cluster. To create the segments of the network, each sink node c propagates the ID of the sink cluster to all the nodes in the tree rooted at *c*. This can be simply done by reversing the flow pointers. The ID of the sink cluster is also considered as the ID of the segment. Figure 36(iii) and Figure 41(ii) show the result of this construction.

4.3.6 Final clean-up

Due to noises and local disturbances, it is possible that some nodes have locally maximum hop count to the boundary, but are not medial-axis nodes. This is likely to happen in sparser regions of the network or near the boundaries if the boundaries detected are not tight. In such cases, the node at the local maximum and all nodes in the tree rooted at it are left without any segment assignment. We refer to such nodes as *orphan nodes* (e.g., the grey nodes in Figure 36(iii)). At the final clean-up stage, we assign the orphan nodes to a nearby segment. In a connected network, there always exists an orphan node *p* such that some neighbor *q* of *p* is not orphan. Each such node *p* selects randomly a non-orphan neighbor *q*, and merges to that segment. This is executed by all orphan nodes until all nodes are assigned a segment. More specifically, each orphan node *p* initially checks its neighborhood to find a non-orphan neighbor *q*. If this check fails, the orphan simply waits for further notifications from its neighbors. After an orphan joins a segment, it notifies all its neighbors. Thus, each orphan is involved in at most two message transmissions, one for checking neighborhood and the other for notification. Figure 36(iv) shows the final result.

Depending on the requirements of applications, the information about the newly formed segments can be disseminated across the network. Each segment has a natural leader — the sink node whose ID the segment takes. This sink node easily collects information about the segment such as node count, neighboring segments, bounding rectangle (if location information is available) etc. This information can be delivered to all nodes in the network, by transmitting along the medial axis and reversed flow pointers. Since the global features of the network have been abstracted into a compact presentation, each node only transmits and stores a small amount of data.

4.3.7 Evaluations

4.3.7.1 Algorithm complexity

We analyze the complexity of the segmentation algorithm with given boundaries, and consider it proportional to the number of messages transmitted. We assume the unit disk graph model during analysis. But the segmentation algorithm works with more general communication models with bi-directional links in practice. Except the boundary detection, the segmentation algorithm contains five steps.

The step of constructing the distance field incurs a few rounds of limited flooding. For the simplicity of analysis, we consider the two-pass implementation of this step and suppose boundary nodes synchronize among themselves during flooding [89]. In the first pass, nodes record the minimum hop count from the boundaries. If the boundary nodes flood inwards almost simultaneously, each node will receive the message from the closest boundary node earlier than any other boundary nodes, thus each node only broadcasts once and all messages received later are suppressed. In the second pass, nodes broadcast their closest intervals. Since nodes can remember all neighbors with lower hop counts in the first round, each node only constructs and broadcasts its final interval once after receiving messages from all neighbors with lower hop counts. An interval can be represented simply by its boundary-ID and end-points, which is $O(1)$ storage. Observe that since we merge adjacent intervals, a non-medial axis node will store exactly one segment, using $O(1)$ storage. A medial axis node that is not a vertex of the medial axis, will store exactly 2 intervals. A medial axis vertex can be the meeting point of at most a constant number of branches of the medial axis (in UDG model). Each such branch contributes at most 2 intervals, resulting in $O(1)$ intervals overall. Thus, at any given time a node can represent its intervals in $O(1)$. Therefore, the distance field can be constructed with $O(n)$ messages, where n is the number of nodes in the network.

Other steps only involve local operations, so the cost of each is at most $O(n)$. For example, in the final step of cleaning up orphan nodes, each node i first broadcasts once to find a neighbor with assigned segment. If it succeeds, node i joins one of its neighbors' segment and notifies all its neighbors. Otherwise, node i simply waits for further notification. Thus, every node is only involved in two message transmissions.

In summary, the proposed shape segmentation algorithm is efficient and incurs communication cost of $O(n)$ in total if nodes synchronize among themselves. Otherwise, the algorithm may incurs $O(n^2)$ communication cost in the worst case. Here, the big-O notation hides a constant including the density of the network.

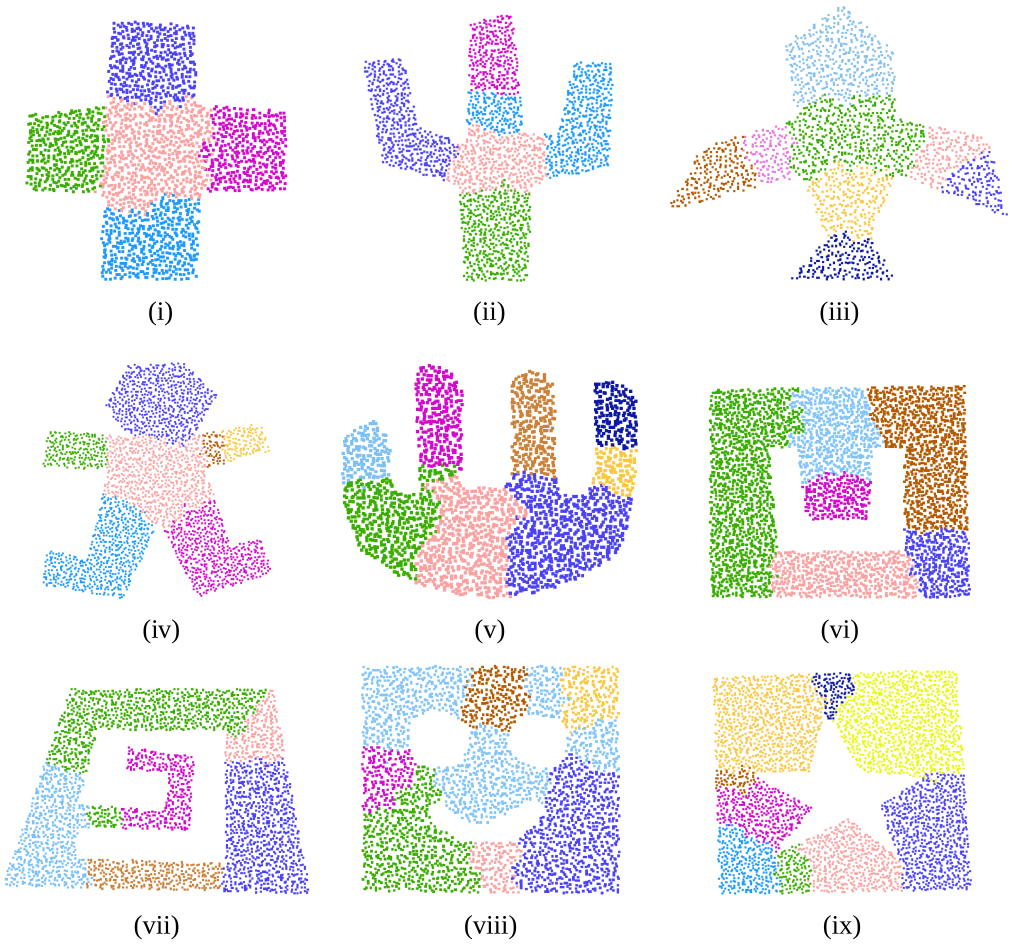


Figure 44: Segmentation results for miscellaneous shapes and densities. (i) cross: 2200 nodes, average 12 neighbors per node. (ii) cactus: 2100 nodes, average 9 neighbors per node. (iii) airplane: 1900 nodes, average 7.8 neighbors per node. (iv) gingerman: 2700 nodes, average 8 neighbors per node. (v) hand: 2500 nodes, average 6.5 neighbors per node. (vi) single-hole: 3700 nodes, average 13 neighbors per node. (vii) spiral: 2900 nodes, average 11 neighbors per node. (viii) smiley: 2900 nodes, average 8 neighbors per node. (ix) star: 3900 nodes, average 9 neighbors per node.

4.3.7.2 Simulations

We simulated the algorithm for different shapes of network, and found that an intuitive partitioning into pieces with regular shape is obtained. We use grid-perturbation distribution with variation in the simulations. These networks either represent practical scenarios, like an intersection of two roads (Figure 44(i)), rooms connected by a corridor (Figure 41), or some pathetically difficult cases we come up with. Several examples are shown in Figure 44. In general, the algorithm performs consistently well when the average degree is $7 \sim 8$ or higher. Good results can be obtained for networks of low density by considering a two or three hop neighborhood in the steps of finding flow pointers, merging the sinks and constructing segments. We used a three hop neighborhood in simulations.

4.4 Extraction of adjacency graph

After segmenting an irregular sensor field into a set of nicely shaped segments, we can apply existing protocols and algorithms inside each piece independently. However, to get global results about the entire network, it requires a high-level structure to integrate the segments together. We propose to extract a compact *adjacency graph* of the segments.

Definition 46 ***Adjacent Segments****: Two segments s_i and s_j are adjacent if there exists a node p in segment s_i that has at least one neighbor in segment s_j. In Figure 45(i), the green and pink segments are adjacent to each other.*

Definition 47 ***Adjacency Graph****: In an adjacency graph $G = (V,E)$, each segment s_i is denoted as a vertex v_i in V. There is an edge $e_{ij} \in E$ between v_i and v_j, if the corresponding two segments s_i and s_j are adjacent to each other. An adjacency graph is showed in Figure 45(ii).*

An adjacency graph is compact. Its size is proportional to the number of segments, which is a small constant in most scenarios. We construct this adjacency graph by using the nodes on the boundaries of the segments to discover the adjacency information locally. More specifically, the construction contains the following two steps:

- **Construct local adjacency graph.** Nodes on the boundaries of segments propagate the IDs of the adjacent segments along the flow pointers towards one of the sinks of the sink cluster. Intermediate nodes cache received messages (together with possible extra information, e.g., hop counts to the boundary nodes) for future usage. Messages with the same information can be suppressed if necessary. All sinks further forward the collected information to the sink cluster leader, which only requires one or two hop transmissions since all sinks are close to each other. Eventually, the leader can gather a local picture about what segments are adjacent to itself.
- **Construct global adjacency graph.** To get a complete adjacency graph, sink cluster leaders exchange their adjacency information via the shortest paths through segment boundary nodes built up during the first phase. After that, leaders push the global graph down to other nodes inside its segment via reversed flow pointers.

We may also augment the adjacency graph with additional useful information on its vertices and edges. For example, if the nodes are aware of their locations, vertices can be associated with the locations of sinks, which will be useful to get a global rough layout; edges can be associated with the Euclidean distance/hop counts between two sinks for finding shortest paths at the top level. In general, we can augment the graph with whatever information gathered by sink nodes, for example, the size of a segment, the number of nodes on the boundary between two segments, etc. These information would aid the design of efficient and load balanced protocols. The above two-phase construction algorithm makes the augmented adjacency graph presented at multiple resolutions at sensor nodes, which will further facilitate the applications. Each node only knows rough information about far-apart segments based on the adjacency graph; but it can store more detailed information about the closest adjacent segments (e.g. the shortest paths to the boundaries). We will elaborate the usage of segmentation together with adjacency graph through various applications in Section 4.5.

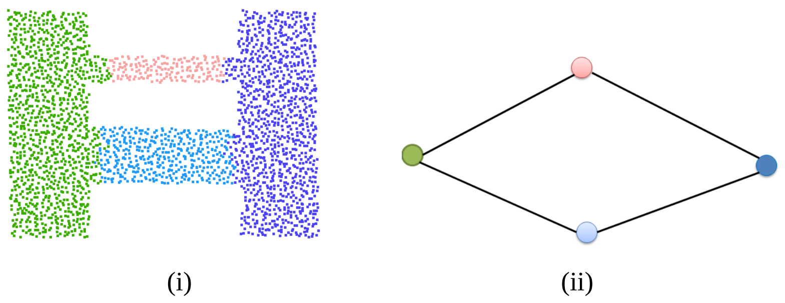


Figure 45: An example of adjacency graph. (i) An irregular sensor field with four segments; (ii) The corresponding adjacency graph.

4.5 Applications

In this section, we present several specific applications that benefit from shape segmentation. These applications are commonly encountered during network design, spanning fundamental problems such as facility placement and routing, as well as data processing, such as data storing and uniform sampling. At the application level, we assume that location information can be made available depending on applications' requirements. But our shape segmentation scheme runs without any geographical information. In fact, the geographical information only gives a node local picture of its neighborhood, but nodes are still unaware of the global features of the network. Simulation results show that shape segmentation improves performance in terms of efficiency and load balance, and facilitates the selection of protocols and the calibration of protocol parameters.

4.5.1 Routing in Irregular Networks

Multi-hop routing is one of the most fundamental functionalities of all kinds of communication networks. Readers can refer to a survey paper [135] to get a comprehensive understanding of routing challenges and protocols in sensor networks. Among the huge literature on routing, geographical routing becomes one of the most popular protocols in sensor networks because it is nearly stateless and avoids message flooding. However, as we mentioned before, inaccurate location information and complex topological features cause geographical routing to have degraded performance in practice. Routing protocols proposed upon virtual coordinate systems [89, 122] do overcome the above weaknesses, but involve non-trivial design of virtual coordinates dependent on the deployment features.

Network segmentation approaches this problem by partitioning the sensor field, so that existing algorithms are still useful inside each segment. It provides a generic recipe for both routing and other applications — inside a segment, existing protocols can still be reused; across segments, an application-dependent structure integrates segment-level data together. Thus, routing and other upper level applications can be conducted in a universal and flexible way. For example, we can choose different protocols for intra-segment and inter-segment routing, purely depending on application requirements. Routing across segments is achieved with the segmentation adjacency graph. Suppose a source node x in segment s_i wants to send packets to destination y in segment s_j. x first finds a desired high level path from s_i to s_j on the adjacency graph by using inter-segment routing. Inside each segment, packets are routed towards the boundary nodes or the destination with intra-segment protocol. The two-level routing scheme follows the philosophy of [122]. But the partitioning of the sensor field is done in a different manner. We discuss the details and possible protocols that can be used at these two levels.

- **Inter-segment routing.** We use the adjacency graph for global route planning. There can be multiple paths from one segment to the other. In Figure 45, two paths connect the left (green) segment to the right (dark blue) segment. Depending on available augmented information and application requirements, we can use different criteria to choose a desired path. For example, distance information (or hop counts) augmented on edges can be helpful to find the shortest path; the number of boundary nodes between two segments implies if there is a bottleneck, which can be used to distribute traffic loads on multiple paths. After selecting a path, packets are forwarded along the set of segments on the path sequentially.
- **Intra-segment routing.** Intra-segment routing is used for routing packets between a pair of nodes inside the same segment. Since each segment relatively has a nice shape (i.e., with simple geometry, without holes), most routing protocols proposed in the literature can be directly used for this purpose. We can use geographical routing if location information is available; we can also

use landmark-based routing by simply putting landmarks at the boundaries of the segments. What routing protocol to use inside a segment is totally independent of inter-segment routing. Each segment can even choose its own protocol. This gives more flexibility to users.

We compare the performance of routing aided by segmentation with GPSR [64] without segmentation. Simulations are conducted on the network showed in Figure 46(i) and Figure 47(i). 10000 source-destination pairs were chosen in each topology. For the topology in Figure 46(i), sources and destinations were selected from either the left segment or the right segment. In the other topology (Figure 47(i)), sources were from the segment at the left-upper corner, and destinations were at the right-bottom corner. We use GPSR combined with manhattan routing[4] for intra-segment routing and for inter-segment routing, we distribute traffic to multiple (e.g., 2 paths in both tested topology) high-level routes based on the size of the segments on the path.

average path length	topology of Fig 46(i)	topology of Fig 47(i)
routing with segmentation	39.73	36.38
GPSR	49.13	30.67

Table 2: The path length averaged over 10000 source-destination pairs in two different network fields, with and without segmentation.

From the simulations, we found that the path length averaged on 10000 pairs in both network fields are close to GPSR. In the topology of Figure 46(i), segmentation-aided routing produces shorter paths than GPSR. Since each segment has relatively nice shape, geographical routing can work well inside segments. Routing along the outer boundary, which happens more frequently in GPSR, is avoided by inter-segment routing. Figure 46 and Figure 47 show the load distribution in the networks, which is counted as the number of transmissions incurred at each node. The darker the color, the higher load a node has. It is clear that nodes with higher load are around boundaries without segmentation. With segmentation, more nodes are involved in packet forwarding. Thus, traffic is more evenly

[4]Basically, if a source at (x_s, y_s) wants to transmit packets to a destination at (x_d, y_d). Packets are first greedily forwarded to the closest node at (x_s, y_d) along a roughly vertical path, then that node forwards packets to the destination along a roughly horizontal path, or vice versa.

distributed among nodes.

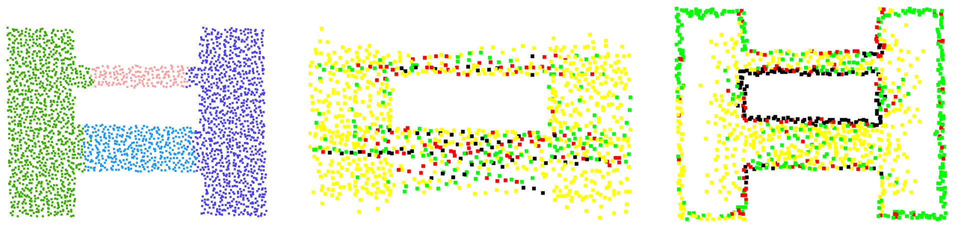

(i) (ii) (iii)

Figure 46: (i) Network topology (ii) Load distribution with segmentation-aided routing. (iii) Load distribution in GPSR without segmentation. Black nodes are with load $>$ 800 transmissions; red nodes are with load $>$ 500 transmissions; green nodes are with load $>$ 300 transmissions; yellow nodes are with load $>$ 100 transmissions

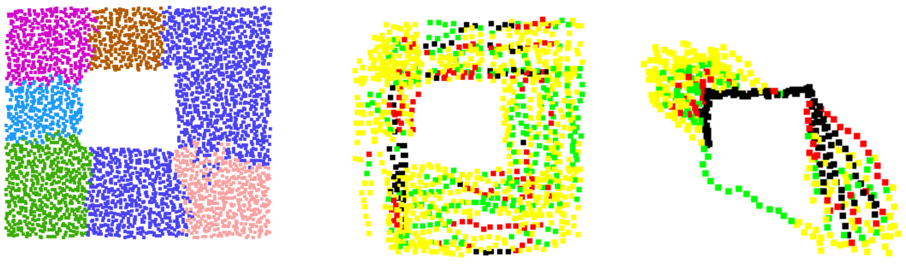

(i) (ii) (iii)

Figure 47: (i) Network topology (ii) Load distribution with segmentation-aided routing. (iii) Load distribution in GPSR without segmentation. Black nodes are with load $>$ 800 transmissions; red nodes are with load $>$ 500 transmissions; green nodes are with load $>$ 300 transmissions; yellow nodes are with load $>$ 100 transmissions.

4.5.2 Facility Location

The problem of base station placement is usually a concern at the very beginning of network design. An intuitive optimization criterion is to place multiple base stations in a way such that the average distance from a sensor to its nearest base station is minimized, so the total energy consumption for data transmissions between sensors and base stations can be minimized.

Solutions for the above placement problem can be classified into two categories. One class includes several centralized approaches based on mathematical models (e.g., integer linear programming) [136, 137]. Centralized approaches give optimal or near-optimal solutions, but they are not suitable for sensor networks wherein nodes are self-organized and do not have any global knowledge. By observing the similarity between the placement problem and the clustering problem, a second approach adapts existing clustering algorithms (e.g., *k*-means clustering [138]) to suggest the placements of base stations. Specifically, in *k*-means algorithm, *k* random locations are selected as the cluster centers. Then the rest of the nodes are partitioned into *k* clusters, with all the nodes closest to one center put in the same cluster. Then the centroid of each cluster is picked as the new center and the algorithm iterates until the centers do not move much. Although the adapted clustering algorithms work well in practice, there are a few limitations of those algorithms.

- The number of base stations to be placed (parameter *k*) may depend on the specific deployment of the sensor network, thus is hard to specify before hand.
- Typically *k* random locations are selected as the initial locations in the *k*-means algorithm. Different initial locations result in different final placement.
- The clustering algorithms try to place base stations based on the distances between cluster centers and the rest of the nodes, but do not respect the geometric features of the underlying physical environment. Inside one cluster bottlenecks can still occur.

Motivated by the limitations of the existing algorithms, we found that segmentation provides an alternative solution for the placement problem. The number of segments gives a natural choice for *k* and the set of sinks can be directly replaced by base stations. More important, the segmentation reflects the significant geometric features of a sensor field, and avoids placing a base station to cover two groups of sensors connected by a narrow bridge. To aggregate data from sensors in each segmentation, the aggregation tree has been implicitly constructed during the segmentation phase. Every node can simply follow the flow pointers to forward data to base stations. By aggregating data within each segment first, we can dramatically reduce network traffic through bottlenecks.

We compare the average distance (hop counts) between a sensor node and its closest base station under different network topologies. We first test the best k for an irregular sensor field by running the k-means algorithm with different k. Figure 48 shows the average distance from a sensor to its closest base station over k in a cross-shape network field (Figure 44(i)). As the number of base stations increases, the average distance is reduced, but the cost of deployment increases. To be the most cost effective with a reasonable budget, Figure 48 suggests to place $4 \sim 6$ base stations, since the average distance can not be reduced much with more than 6 base stations. This is consistent with the number of segments (5 segments) the segmentation algorithm gives. Thus, in the following comparisons, we only compare the segmentation approach to the k-means algorithm with k equal to the number of segments, which is a good indication of the number of base stations needed. We compare the segmentation approach with k-means algorithm. With

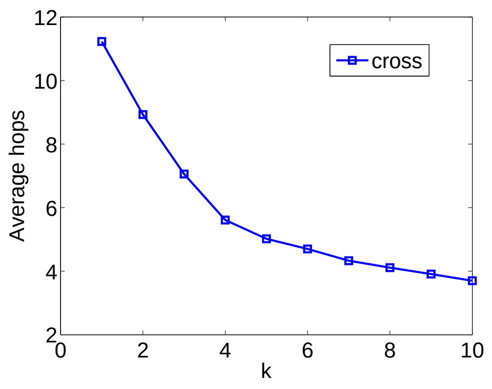


Figure 48: The average hops from a sensor to its closest base station over various k in a cross-shape network.

segmentation, one solution is to place a base station at the centroid location of the sink cluster at each segment. Each node chooses the base station inside the same segment as its own base station. Furthermore, we combine the segmentation with k-means algorithm by using the centroid location of sink clusters as the initial placement, then running k-means algorithm based on that. The performance of three approaches is showed in Table 3. The k-means algorithm with the initial position as the sinks of the segments works the best. One thing worth noticing is that the pure segmentation approach achieves comparable performance compared to *k*-means algorithm and the combined one, but avoids the cost of multiple iterations required in the *k*-means algorithm, and more importantly, it suggests a good choice for the parameter *k*, deciding which is typically a challenge in the standard *k*-means algorithm.

average hop counts	cross	plane	corridor	fish
k-means	5.02	5	7.42	11.58
shape segmentation	5.76	5.78	7.83	12.01
k-means over segmentation	4.95	4.94	7.36	11.55

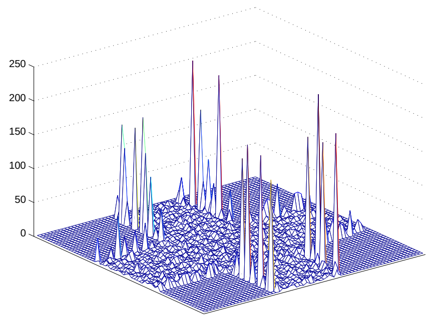
Table 3: The average hops from a sensor to its closest base station with and without shape segmentation.

4.5.3 Distributed Index

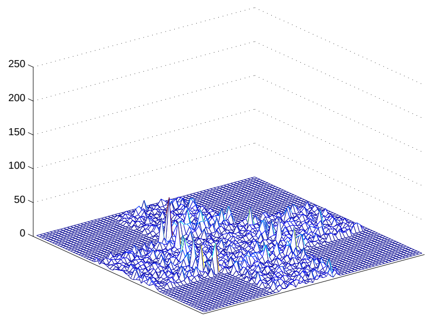
Distributed index for multi-dimensional data (DIM [71]) is a quadtree type hierarchy that supports efficient multi-resolution data storage and range query. The key idea of DIM is to map an event with certain values to a specific area called as *zone*, and store the event with geographic routing to the node owning that zone. The zone is determined by dividing the bounding rectangle of the network alternatively with a vertical or horizontal line until there is a single node inside the zone. When a node generates an event, it estimates the destination zone based on the event value and routes it towards there.

DIM provides a scalable index structure for data storage and performs well in a network field with simple geometric topology. However, it suffers a lot from load unbalancing in a complex shaped sensor field. For a network with arbitrary shape, there will be large empty space in the bounding rectangle. Some nodes (especially those boundary nodes) must take care of a larger zone, and hence store more data than others. Overloaded nodes would be depleted faster than other nodes, which may lead to network partitioning and shorten network lifetime.

With shape segmentation, we can avoid above problems by applying DIM on each segment. Specifically, we first divide the entire event range into several sub-ranges. Let N_i denote the number of nodes belonging to the i^{th} segment, and N



(i)



(ii)

Figure 49: (i) Distribution of storage load in basic DIM structure. (ii) Distribution of storage load in shape segmentation integrated DIM structure.

denote the total number of nodes. Sub-ranges are divided based on the ratio of N_i/N. The first segment takes care of events within range $[0, N_1/N)$, and the second segment takes care of the range $[N_1/N, (N_1+N_2)/N)$, and so on. A new generated event is divided into several sub-events, each of which is sent towards the corresponding segments respectively. Inside each segment, the sub-event is processed in the same way as the basic DIM algorithm.

To compare the performance of DIM with and without shape segmentation, we run simulations on various network scenarios. We generated 10000 events with values uniformly distributed in a fixed range $[0, 1000]$, and stored them into the network. Figure 49 shows the distribution of storage load for the cross network. We can see that the boundary nodes in the basic DIM structure suffer much higher loads than the rest of the network. On the other hand, with shape segmentation, since each segment has tighter bounding rectangle and each node is associated with an almost equal sized zone, data is seen to be well distributed across the network with no particular preference for occurrence of peaks. The peaks in Figure 49(i) reach 248, while the highest peak in Figure 49(ii) is only 65.

Shape segmentation also helps reduce communication cost by mapping events into more accurate locations. Table 4 shows that the average communication cost in terms of hop counts for every event insertion is much less with shape segmentation in all three different network scenarios, viz. cross (Fig. 44(i)), corridor (Fig. 41) and fish network (Fig. 36). In the cross and corridor network, shape segmentation saves

60% $\sim$ 70% cost. The gain in fish-type network is about 20%, not as significant as the previous two cases. The reason is that each piece of ‘fish’ does not tightly match with its bounding rectangle.

cost per event insertion	cross	corridor	fish
without shape segmentation	293.69	359.21	254.37
with shape segmentation	84.15	151.05	204.58

Table 4: Average data insertion cost for DIM with and without shape segmentation.

4.5.4 Random Sampling

We discuss the benefits of shape segmentation with another example - random sampling. Uniform random sampling of a sensor node is a fundamental operation that is used as a basic element in many scenarios such as geographical hash tables [34], geographical gossip [124] and information diffusion and storage [139].

The basic sampling procedure works as follows [123]. A node who wants to pick a random sensor in the network first chooses a random geographical location inside the bounding rectangle, and uses geographical routing to route towards that location. The message will eventually arrive at the node closest to the picked location. A node p is picked with a probability proportional to the area of its Voronoi cell. To achieve a uniform sampling distribution, the acceptance probability of sampling at each node needs to be adjusted, as the one with a large Voronoi cell is more likely to be picked. Basically, each sampled node will be accepted with probability $r_i = \min(\tau/a_i, 1)$, where τ is a given threshold and a_i is the area of the Voronoi cell associated with node i. If a node rejects a sample, it will pick a new location and repeat the above process. In an irregular sensor field, the Voronoi cells of different nodes have vastly varying areas. Nodes with large Voronoi cells are picked more likely, yet often get rejected afterwards. Thus, the sampling efficiency suffers as a lot of trials end up in vain. Furthermore, since the fate of each sample can only be determined at the destination node, samples may be rejected after traveling a long path, which incurs expensive communication cost and wastes network resources.

Random sampling integrated with shape segmentation can dramatically reduce the number of unnecessary trials, at the same time achieving uniform sampling. The adapted algorithm runs as follows. Each time before sampling, we first randomly select a segment. Each segment is selected with probability $P_i = N_i/N$. After that, we pick a random location within the bounding rectangle of the selected segment. Within each segment we apply the same sampling algorithm and sampling rejection policy as before. Segments are divided into Voronoi cells with much smaller variation, thus no node would reject samples with abnormally high probability.

We run simulations on the same three typical networks. Results are averaged on 10 rounds, and in each round, we randomly pick 100 samples. For the basic random sampling algorithm, we set τ to the ratio of the size of the network field and the total number of nodes. Each segment has its own τ as the ratio of the segment size and number of nodes belonging to that segment. In Table 5, we compare the average number of trials taken to get 100 samples. The basic random sampling algorithm tried 168, 149 and 136 times for ‘cross’, ‘corridor’ and ‘fish’ respectively. Shape segmentation reduces the number to 112, 115 and 123. Table 6 shows the average communication cost per sample. As expected, the cost in shape segmentation case is less than the basic case.

no. of trials	cross	corridor	fish
without shape segmentation	168	149	136
with shape segmentation	112	115	123

Table 5: Average number of trials for 100 random sampling.

cost per sampling	cross	corridor	fish
without shape segmentation	477.84	511.95	361.80
with shape segmentation	102.49	182.32	238.47

Table 6: Average cost per sampling.

With the same observation we got in DIM, shape segmentation shows different levels of improvements in different network scenarios. For these two applications, the performance more or less depends on whether the bounding rectangle is tight enough. We notice that this is due to an inherent assumption of the basic sampling algorithm that uses a bounding rectangle on the sensor field. Further improvement can be made by using a tighter polygon to approximate the shape of the segment in the basic sampling algorithm.

4.6 Discussions

We introduced a simple distributed algorithm that partitions an irregular sensor field into nicely shaped segments, by using the connectivity information. We show that segmentation is a generic approach to handle complex geometric features and improve the performance of algorithms that assume a nice regular sensor field. We mainly presented several common problems encountered during network design and management. But the applications of shape segmentation can go beyond that. For example, the recent work on information dissemination and collection by sweeps [140] can be directly integrated with shape segmentation and be applied inside each segment. Shape segmentation can also help the construction of virtual coordinate systems. Take a landmark-based routing scheme [122] for an example in which the placement of landmarks has a critical impact on its performance. Since the segments have a nice shape, a few landmarks inside each segment would suffice for routing in and between segments.

We summarize the impact of shape segmentation as follows:

- Provides at a global level a compact way to represent the underlying diverse geometric features of a sensor network field, and makes the network design and protocol development transparent to the specific deployment.
- Facilitates the design and development of new topology-adaptive protocols and makes existing protocols that assume nice shaped field reusable.
- Gives users great flexibility to 'mix-and-match' protocols and calibrate important protocol parameters with respect to the specific deployment.

In shape segmentation, a generally unsolved issue is that there is no well accepted definition on good segmentation so far. The choice of appropriate segmentation may also depend on the applications. For example, a spiral-like sensor field is equivalently nice as a long corridor for routing protocols, but it needs to be segmented further for applications that require a quad-tree type hierarchy. We proposed two schemes to give certain guarantees on the fatness of the segments but also provide flexibility for the upper level applications to pick a definition and choose proper segmentation granularity. One interesting problem is to classify applications into several categories so that more precise segmentation definitions can be found for each category. We regard this as our future work.

Chapter 5

Provide Easy Programming Paradigm

5.1 Introduction

Programming a sensor network application remains a difficult task, since the programmer is burdened with low-level details related to distributed computing, careful management of limited resources, unreliable infrastructure, and energy optimizations. Thus, developing a powerful programming framework for sensor network is critical to realizing their full potential as collaborative processing engines. There has been some progress in developing operating system prototypes [141,142] and programming abstractions [143, 144]; however, these abstractions have provided only minimal programming support. Prior work on viewing the sensor network as a distributed database provides a declarative programming framework which is amenable to optimizations. However, it lacks expressive power, and the developed database engines (TinyDB [145], Cougar [15]) for sensor networks implement only a limited functionality. On the other hand, the recently proposed Kairos [146] framework is expressive, but is based on a procedural language and hence, difficult to translate to efficient distributed code. Thus, the overall vision of a programming framework that automatically translates a high-level user specification to efficient distributed code remains far from realized. In general, a perfect programming paradigm for sensor networks must achieve the following.

- Be sufficiently expressive.
- Be declarative, i.e., provide users with a high-level abstraction of the network, while hiding all the network machinery such as distributed computing, efficient storage, communication efficiency, etc.
- Be amenable to automatic optimizations (especially, related to energy consumption) without much input from user.

We motivate use of deductive approach for programming of sensor networks, and design and develop a query engine for distributed evaluation of general (with stratified negation) deductive queries. In particular, our developed system facilitates automatic translation of high-level deductive queries into optimized nesC node that runs on individual sensor nodes.

Proposed Deductive Approach. We propose a programming framework based on a deductive paradigm; our proposed framework is declarative, fully expressive (Turing complete), and most importantly, amenable to automatic translation into efficient distributed code. Deductive approach has been recently used with success for declarative specification of network routing protocols [42] and overlay architectures [147]. In the context of programming sensor networks, our deductive approach is motivated by the basic observation that sensor networks essentially gather sets of "facts" by sensing the physical world, and sensor network applications manipulate these facts. We believe that the collaborative (involving multiple nodes) functionality of a sensor network application can be easily represented using fact-manipulation deductive rules. The local arithmetic computations such as signal-processing, data fusion, etc. may be inefficient to represent using deductive rules, and hence, are embedded in locally-processed built-in functions written in procedural code. Embedding such local computations in locally-processed procedural functions does not affect the communication efficiency of the translated code. The above approach facilitates easy high-level specification of an application, and is amenable to optimizations. To realize the overall vision of a powerful programming framework, we develop techniques for communication-efficient evaluation of deductive programs in resource-constrained sensor networks over streaming data. Based on the developed query processing techniques, our system will automatically translates a given high-level specification of an application into optimized distributed code that runs on individual nodes.

5.2 Prior Approaches, and Deductive Framework for Sensor Networks

In this section, we start with an overview of prior approaches for programming sensor networks. Then, we give an overview of deductive programming, and illustrate the power of our approach through various illustrations. Finally, we propose some extensions and restrictions to the deductive framework to tailor it to programming of sensor networks.

5.2.1 Prior Approaches for Programming Sensor Networks

NesC and Programming Abstractions. The Berkeley motes platform provides the C-like, fairly low-level programming language called *nesC* [142] on top of the TinyOS [141] operating system. However, the user is still faced with the burden of low-level programming and optimization decisions. There has been some work done on developing programming abstractions [143, 144, 148–151] for sensor networks; however, these abstractions provide only minimal programming support. Finally, authors in [152] propose an interesting novel approach of expressing computations as "task graphs," but the approach has limited applicability.

Sensor Network as a Distributed Database. Recently, some works [15, 46, 145] proposed the powerful vision viewing the sensor network as a distributed database. The distributed database vision is declarative, and hence, amenable to optimizations. However, the current sensor network database engines (TinyDB [145], Cougar [15]) implement a limited functionality of SQL, the traditional database language. In particular, they only handle single queries involving simple aggregations [14, 16, 153] or selections [154] over single tables [155], local joins [16], or localized/centralized joins [49] involving a small static table. These approaches are appropriate for periodic data gathering applications. SQL is not expressive enough to represent general sensor network applications. Moreover, due to the lack of an existing SQL support for sensor networks, there is no real motivation to choose SQL. Our deductive approach is essentially an expansion of the initial vision of viewing the sensor network as a database. In effect, we propose use a more expressive deductive approach, and propose to build a full-fledged efficient logic query engine for sensor networks.

Procedural Languages. Recently proposed Kairos [146] provides certain global abstractions and a mechanism to translate a centralized program (written in a high-level procedural language) to an in-network implementation. In particular, it provides global abstractions such as `get_available_nodes`, `get_neighbors`, and remote data access. Kairos is the first effort towards developing an automatic translator that compiles a centralized procedural program into a distributed program for sensor nodes. However, Kairos does not focus much on communication efficiency; for instance, the abstraction `get_available_nodes` gathers the entire network topology, which may be infeasible in most applications.

In some sense, our approach has the same goals as that of Kairos – to automatically translate a high-level user specification into distributed code. However, since Kairos approach is based on a procedural language, it is much harder to optimize for distributed computation. Through various examples in Section 5.2.2.1, we suggest that our proposed framework will likely yield more compact and clean programs than the procedural code written in Kairos. Moreover, the deductive programs for the examples in Section 5.2.2.1 yield efficient distributed implementations involving only localized joins.

In general, we feel that procedural languages are unlikely to be very useful in a restricted setting such as sensor networks, since they are not declarative and would be hard to distribute and optimize for communication cost.

5.2.2 Overview of Deductive Programming

Predicate logic is a way to represent "knowledge" and can be used as a language for manipulating tables of facts. In logic data model, each table (relation) of facts is looked upon as a predicate having an argument for each table attribute. The simplest model of predicate logic, Datalog, consists of a set of declarative logic rules, possibly involving recursion and negations. Datalog without recursion is as expressive as the traditional database language SQL without aggregations. In our proposed programming framework, we use full first-order logic which extends Datalog by allowing function symbols in the arguments of predicates, and thus, making the framework Turing complete [156]. We illustrate the need for function symbols in Example 49 of Section 5.2.2.1. Essentially, in full first-order logic, the arguments of a predicate may be arbitrary terms, where a term is recursively defined as follows. A *term* is either a constant, variable, or $f(t_1, t_2, \ldots, t_n)$ where each t_i is a term and f is a function symbol. In this general context, a logic rule is written as

$$H \ :- \ G_1, G_2, \ldots, G_k.$$

H is called the *head*, and $G_1, \ldots, G_k$ are the *body subgoals*. The head and the subgoals are of the form $p(t_1, t_2, \ldots, t_m)$ where p is a predicate and t's are arbitrary terms.

Built-In Predicates, and Added Features. Certain predicates that are given a conventional interpretation such as $X < Y$, are called *built-in* and can appear in the body subgoals. In our framework, a user may define additional built-in predicates, in which case the user provides the procedural code to evaluate the predicate. Note that built-in predicates can be easily used to specify built-in functions, and hence, we use *built-in functions* directly in the logic rules. For sake of ease in programming, we allow restricted use of negated subgoals, lists, Prolog-like *setOf* and *bagOf* constructs which allow construction of lists in a similar way as Group-By construct of SQL.

Motivating Characteristics of A Deductive Approach. In short, our choice of deductive approach is motivated by its following characteristics. Firstly, a deductive programming framework is declarative and hence, amenable to optimizations. In our context, the optimization of logic programs is largely embedded in the efficient data storage schemes, in-network implementation of join, join-ordering, and query optimization techniques. Secondly, a deductive framework augmented with function symbols is fully expressive; in particular, it is more expressive than the prior distributed database approach. Extensive use of function symbols (or lists) does make optimizations difficult, but we anticipate that function symbols will be used in limited contexts and hence, allow their use for full expressibility. Thirdly, a deductive framework has strong theoretical foundations and can be easily extended to include other specialized deductive frameworks.

Prior Use of Datalog in Declarative Networking. Recently, Datalog without negations has been used for declarative specification of network routing protocols [42] and overlay architectures [147], resulting in very compact and clean specifications.

The approach was shown to be efficient, secure, expressive for intended purposes, and amenable to query optimizations. This recent success of use of deductive queries for declarative networking adds to the promise of our deductive approach for programming sensor networks.

5.2.2.1 Illustrating the Power of Deductive Approach

As done in [146] to illustrate Kairos' expressibility and flexibility, we also illustrate the power of our approach by describing how it may be used to program a few different distributed computations that have been proposed for sensor networks: vehicle tracking, localization, routing tree construction, and vehicle trajectories. We start with discussing the use of built-in functions to embed arithmetic computations.

Representing Signal-Processing, Data Fusion, and Other Arithmetic Computations. Certain aspects of sensor network applications involve local arithmetic computations such as signal processing, data fusion, synthesis of base data, etc. Such arithmetic computations may be too inefficient to represent in a deductive framework, and hence, are embedded in locally-processed built-in functions coded in a procedural language. Such a representation does not compromise on the communication efficiency on the translated distributed code. Distributed arithmetic computations are embedded in built-in aggregates with specialized distributed implementations. For instance, in vehicle tracking [109,157], arithmetic computations involve estimating belief states, information utilities, and estimate of the future target location; the first two computations are local, while the last computation requires the *maximum* aggregate. See Example 48 below. Finally, certain other arithmetic techniques such as data compression may be embedded in the query engine.

Example 48 Signal Processing in Vehicle Tracking. The given program represents the algorithm for tracking vehicles described in [157]. The algorithm uses probabilistic and signal-processing techniques to maintain posterior distribution (*belief state*) of the vehicle location.

$$
\begin{array}{ll}
U(i_1,t+1,u_1) & :-\ P(i,t,v),G(i,i_1),Z(i,t+1,z), \\
 & \quad Z(i_1,t+1,z_1),u_1 = I(v,z,z_1) \\
P'(i_1,t+1) & :-\ P(i,t,v),G(i,i_1),G(i,i_2),U(i_1,t+1,u_1), \\
 & \quad U(i_2,t+1,u_2),u_1 < u_2 \\
P(i_1,t+1,F(v,z)) & :-\ P(i,t,v),Z(i,t+1,z),G(i,i_1), \\
 & \quad \text{NOT } P'(i_1,t+1)
\end{array}
$$

At any time instant, only one node namely the leader node is active. The leader applies a measurement of its observation and produces an updated belief state about the vehicle location. The updated belief is then passed onto one of the neighboring nodes with the highest "utility information," which becomes the new leader, and the process repeats. In the given program, we have used the same variable symbols as used in [157]. For a node i at time t, $P(i,t,v)$ signifies the belief state value v, $U(i,t,u)$ signifies the information utility value u, and $Z(i,t,z)$ signifies the sensed value z. Also, $G(x,y)$ represents the network edges, F and I are locally-processed built-in functions. The function F represents the Equation 3 of [157] which computes the updated belief state at the new leader node, and I computes the information utility of a local node. The first logic rule in the given program computes the information utility of a neighbor i_1 of the leader node i, and the third rule computes the new leader node and the new belief state. The predicate $P'(i_1,t+1)$ signifies that i_1 does *not* have the highest information utility. The given logic program is more compact than the corresponding procedural code written in Kairos (see [146]). .

Example 49 Parallel Trajectories: Need for Function Symbols. We now illustrate the need for function symbols in our programming framework. Essentially, function symbols are required when we want to create non-atomic values.

$$
\begin{array}{ll}
traj([R_1,R_2]) & :-\ report(R_1),report(R_2),close(R_1,R_2), \\
 & \quad NOT\ \ notStartReport(R_1) \\
notStartReport(R_2) & :-\ report(R_1),report(R_2),close(R_1,R_2) \\
traj([X|R_1,R_2]) & :-\ traj([X|R_1]),report(R_2),close(R_1,R_2) \\
completeTraj([X|R]) & :-\ traj([X|R]),NOT\ \ notLastReport(R) \\
notLastReport(R_1) & :-\ report(R_1),report(R_2),close(R_1,R_2) \\
parallel(L_1,L_2) & :-\ completeTraj(L_1),completeTraj(L_2), \\
 & \quad isParallel(L_1,L_2)
\end{array}
$$

Here, we use R to represent the triplet (x,y,t) signifying the location (x,y) and time t of vehicle detection, and compute vehicle trajectory paths from the base data *report(R)*. For simplicity, we assume that at any instant there is only one sensor detecting the target, so the *trajectory* can be directly synthesized using a sequence of *report* tuples. For clarity, we use lists instead of function symbols; the list notation $[X|Y]$ signifies X as the head-sublist and Y as the tail-element. We use two locally-processed built-in functions: *close* checks if two reports can be consecutive points on a trajectory (i.e., close enough in the spatial and temporal domains), and *IsParallel* checks if two trajectories are parallel.

Example 50 Shortest-Path Tree. Here, we give a logic program for constructing a shortest path tree (H) with a given root node (A). in a given network graph G.

logicH Program:

$$\begin{array}{lll} H(A,A,0). & & \\ H(A,x,1) & :- & G(A,x) \\ H'(y,d+1) & :- & H(\_,y,d'),(d+1)>d',H(\_,x,d),G(x,y) \\ H(x,y,d+1) & :- & G(x,y),H(\_,x,d),\ NOT\ H'(y,d+1) \end{array}$$

The predicate $H(x,y,d)$ is true if there is a path of length d from A to y using the edge (x,y); essentially, $H(x,y,d)$ gives the set of edges added in the breadth-first search at d^{th} level. The predicate $H'(y,d+1)$ is true if there is already a path from A to y of length shorter than $d+1$; the last two terms in the third rule are to ensure safety (to bound d). The given logic program is more compact than the 20 lines of procedural code written in Kairos [146]. More importantly, it can be easily translated into distributed code that incurs near-optimal communication cost.

Example 51 Another Version of Object Tracking. Here, we present another version of vehicle tracking, which uses a simple algorithm based on DARPA NEST demonstration software, described in [143, 144]. Each node takes periodic magnetometer readings and compares them to a threshold value. Nodes above the threshold communicate with their neighbors and elect a leader. We define leader to be a node with the largest magnetometer reading within its 2-hop neighborhood. The leader computes the centroid of its neighbors' sensor readings, and transmit to a base station. As in previous examples, let $G(x,y)$ be the network graph. Let

$V(x,v,l,t)$ be the base table, where v is the above-threshold[1] magnetometer reading and l is the estimate of the object location by node x at time t. In the below program, the table/predicate $Leader(x,t)$ signifies that the node x is a leader at time t. The predicate *Leader* is defined in terms of the predicate $NotL(x,t)$ which signifies that x is *not* a leader at time t. Alternatively, we could have used an aggregation function *max* for defining *Leader*. Now, we define the predicate $Loc(x,y,l,t)$ to collect the object-location estimates of the leader neighbors. In particular, predicate $Loc(x,y,l,t)$ is true if x is a leader at time t, y is within 2-hops of x, and l is the object-location estimate at node y. Finally, the predicate $Ctrd(x,c,t)$ computes the centroid of location-estimates for each leader x based on the $Loc(x,y,l,t)$ facts. We use the Prolog-construct *bagOf* to assemble all the location-estimates in a list L, for each instantiation of (x,t) (i.e., *GroupBy* x,t in terms of SQL). The built-in predicate $centroid(L,c)$ is used to compute the centroid of the values in list L.

$$\begin{array}{lll}
NotL(x,t) & :- & V(x,v,\_,t), V(y,v_1,\_,t), G(x,y), v < v_1 \\
NotL(x,t) & :- & V(x,v,\_,t), V(y,v_1,\_,t), G(x,z), G(z,y), v < v_1 \\
Leader(x,t) & :- & NOT\ NotL(x,t), V(x,\_,\_,t) \\
Locs(x,y,l,t) & :- & Leader(x,t), V(y,\_,l,t), G(x,z), G(z,y) \\
Locs(x,y,l,t) & :- & Leader(x,t), V(y,\_,l,t), G(x,y) \\
Locs(x,x,l,t) & :- & Leader(x,t), V(x,\_,l,t) \\
Ctrd(x,c,t) & :- & bagOf(l,(x,t)\widehat{\ } Locs(x,y,l,t),L), centroid(L,c)
\end{array}$$

Limitations of the Deductive Approach. As with any programming framework, deductive programming has its own limitations. In particular, logic programs are sometimes non-intuitive or difficult to write; e.g., the shortest path tree program of Example 50 is clean and compact, but quite non-intuitive compared to a procedural code. As such the deductive framework is targeted towards expert and trained users, for whom the relief from worrying about low-level hardware and optimization issues would far offset the burden of writing a logic program.

[1]The check for reading being above a given threshold can be done locally, and hence, ignored in the given logic program for clarity.

5.2.2.2 Restriction of XY-Stratification

In this section, we discuss the need to restrict our framework to XY-stratification. We start with discussing various levels of stratifications and recursions.

Programs without Negations. The basic Datalog programs have no negated sub goals. Such programs are severely limited in its expressive power and cannot express many of the queries of practical interest. For this set of programs, our evaluation techniques support arbitrary recursions with tuple insertions, but no deletions.

Stratified Programs. In-order to support more general queries, we must add negations in the queries. However, evaluation of logic programs with unrestricted negation and recursion is infeasible in sensor networks, since it will require a series of distributed fixpoint checks for evaluation of well-founded semantics [158]. Therefore, we evaluate restricted negations rather than arbitrary negations. The most restrictive programs with negations are called as stratified programs. In stratified programs, there are no cycles through negations in the program's "dependency" graph. In other words, there may be positive cycles, but no negative cycles at the predicate level. Same as the first case, we only support tuple insertions.

Locally-Stratified Programs. The slightly more general usage of negations is locally-stratified programs. In locally-stratified programs, for any ground atom A, it is not possible for the negation of atom A to appear in a resolution path from A. However, this notation is not useful in sensor networks, since determination of local-stratification is undecidable [159] and depends on the given instance of the base data. Note that in sensor networks, the base data is dynamic and not even available until run-time.

XY-Stratified Programs. For the above reasons, we restrict ourselves to XY-stratified programs [160] which are essentially programs that are locally-stratified with an ordering imposed (by built-in arithmetic functions) on the argument values of the derived facts. The atoms of header predicate must be in the same level or higher level strata of the atoms of predicates in the body. For instance, the program of Example 50 is locally-stratified for any G (base data) because $H(x,y,d+1)$ depends directly or indirectly on $H(x,y,d')$ only for $d > d'$. Such ordering helps the compiler easily check if a program is XY-stratified. More important, XY-stratified programs are locally non-recursive, which make the support of tuple deletions feasible.

5.3 Query Evaluation in Sensor Networks

For in-network evaluation of logic programs, we choose the bottom-up approach (instead of top-down approach) because the bottom-up approach is amenable to asynchronous distributed implementation and incremental evaluation, has minimal main-memory requirements, and requires the simpler term-matching operator [161] (instead of unification). The seminaive bottom-up approach with magic-sets is at least as efficient as the top-down approach [161, 162]. Magic-set transformations can increase the program size considerably, but the program code is stored in the flash memory which is ample.

For a distributed implementation of the bottom-up approach, each fact table (base table or derived) is partitioned across the entire network, possibly, using indexes or hashing. Each sensor node *independently* "handles" all facts hashed to itself. Here, *handling a fact* t involves evaluation of logic rules that involve the subgoal corresponding to t. Newly derived facts are hashed to appropriate locations, where duplicates are eliminated. In our context, the seminaive trick is subsumed in the incremental maintenance of intermediate results (which is anyway needed for streaming data) and duplicate elimination of derived facts. Evaluating a logic rule may usually involve a join of multiple tables, which may involve broadcast/routing of facts to appropriate locations to search for matching facts. Thus, at the core of the distributed bottom-up approach lies an in-network algorithm for join of multiple data streams as discussed in previous section. In case of logic rules with function symbols or lists, evaluation of a join predicate involves the simple "term-matching" operator [161].

The details of in-network implementation of join [32] can be found in Chapter 2. We omit the details here.

5.3.1 Evaluation of XY-Stratified Deductive Queries

In this subsection, we discuss the evaluation of XY-stratified deductive queries. In particular, we discuss how to handle negations and deletions in a locally non-recursive logic program.

High-Level Plan. Perpendicular Approach (PA) can maintain a join-query result in response to simultaneous insertions. We start with generalizing PA to handle deletions into the operand streams. Eventhough the base operand streams may be insert-only streams, generalizing PA to handle deletions is fundamental to generalizing it to evaluate XY-stratified deductive programs. As a second step, we generalize PA to maintain (and thus, evaluate) query results represented by single deductive rules involving negated subgoals. In later steps, we include recursion into single deductive rules, and then, generalize it to general single-stratum and arbitrary stratified programs.

Generalizing PA to Handle Deletions. Consider data streams $R_1, R_2, \ldots, R_n$ in a sensor network. Let $R_1, R_2, \ldots, R_n$ also denote the *current* sliding windows of respective data streams, and let the join-query result $R_1 \bowtie R_2 \ldots \bowtie R_n$ be stored (as a set, without duplicates) in a distributed manner across the network based on some hashing scheme.

Let us consider deletion of a tuple t_1 from the stream R_1. For now, lets assume that there are no other insertions/deletions. To maintain the join-query result, we need to compute $t_1 \bowtie R_2 \ldots \bowtie R_n$ and "delete" it from the maintained join-query result. However, due to set semantics, $(R_1 - t_1) \bowtie R_2 \ldots \bowtie R_n$ may *not* be equal to $(R_1 \bowtie R_2 \ldots \bowtie R_n) - (t_1 \bowtie R_2 \ldots \bowtie R_n)$. We can solve the above problem using one of the following means: (i) Store results as bags, or keep a count of multiplicity of each result tuple as suggested in [163], (ii) Keep the actual set of derivations (as described later) for each result tuple, or (iii) Use the rederivation technique of [163]. The counting technique (or bag semantics) is difficult to implement accurately for a fault-tolerant technique such as Perpendicular Approach, since fault-tolerance yields non-deterministic duplication of results. Also, counting technique is not applicable to general recursive queries [163]. We discuss an approximate implementation of counting technique later. The rederivation technique of [163] will require distributed computation of maintenance queries, and hence, will result in a lot of communication overhead. However, the technique of keeping the actual set of derivations (as described below) incurs no additional communication overhead and guarantees correctness. Storage of set of derivations does incur a space overhead, which may be minimal if most tuples have only a small number of derivations.

Definition 52 Source Node; Tuple ID; Derivation of a Tuple. *Source node of a tuple is the node in the network where the tuple is generated (for a base tuple) or hashed (for a derived tuple). We use $I(t)$ to denote the source node of a tuple t.*

The tuple-ID is an identifier that uniquely identifies each tuple in a (base or derived) data stream. For our purposes, we use $(I(t), \tau_t)$ as the ID of a tuple t, where τ_t is at local timestamp at $I(t)$ when the tuple t was inserted.

A derivation of a derived tuple t is the list *of tuple IDs, one from each of the operand streams, that match/join to yield t. Note that a tuple may have multiple different derivations. In a general deductive program, a derivation of t includes the rule-ID used to derive the tuple, but does not include the tuple IDs corresponding to negated subgoals. Due to recursive rules, a derivation may include a tuple-ID from the same table as t.*

Now, to accurately maintain $T = R_1 \bowtie R_2 \ldots \bowtie R_n$ in response to deletions from an operand stream, we store (and maintain) *set* of all derivations with each tuple in T. When a tuple t_1 is deleted from R_1, we compute $T_1 = t_1 \bowtie R_2 \ldots \bowtie R_n$ along with the derivation of each tuple in T_1. Then, for each derived tuple t in T, we subtract the set of derivations of t in T_1 from the set of derivations of t in T. Set of derivations are similarly maintained in response to insertions into operand streams. The tuple t is deleted from (inserted into) from T if the resulting set of derivations of t becomes empty (non-empty) (see proof in Theorem 53). The computation of T_1 constitutes the join-computation phase for deletion of t_1. In the storage phase, the tuple t_1 is deleted from all the nodes where it was stored in the storage phase of its insertion (i.e., from all the nodes on the horizontal line at its source node, in case of PA). Note that a deletion of a derived tuple occurs only at its source node (due to the hashing scheme).

Theorem 53 *(1) A tuple is inserted only at the insertion of the first derivation; (2) a tuple is deleted only at the deletion of last derivation.*

Proof. (1) It is easy to easy that the first derivation of a tuple means the creation of a new tuple. Since the join results are stored as a set, other derivations of the same tuple would not result in the insertion of that tuple.

(2) A tuple needs to be deleted if it has no derivations any more, so it should be deleted when its last derivation is deleted. To prove the second part of the theorem, we also need to prove that the tuple can still be derived when some derivations (but not the last one) are deleted. Since we restricted ourself to instance-acyclic logic programs, each tuple at the n^{th} strata only depends on tuples at the same or lower level stratum. For a derivation $D = <r_1, r_2, \ldots, r_n>$, r_i corresponds to either a ground atom or a tuple with derivations $<r'_1, r'_2, \ldots, r'_m>$, and $S(r'_i) <= S(r_i)$ (S(r) denotes the strata level of r). Since there are finite number of predicate instances at each level of strata and no instance cycles, if we substitute each r_i recursively, r_i can be eventually represented by a set of ground atoms. Therefore, as long as a tuple has some derivations, that means it can still be derived by a set of ground atoms, which validates its existence.

□

Deductive Rule with Negated Subgoals. We now generalize our approach to maintain a query result T represented by a safe deductive rule with negated subgoals. Let

$$T :\text{-}\ R_1, \ldots, R_n,\ NOT\ S_1, \ldots, NOT\ S_m$$

Above, each R_i or S_j (not necessarily distinct) is a data stream in the sensor network. As mentioned in Definition 52, a derivation of a tuple contains tuple IDs corresponding to only the positive subgoals. For a safe rule, such a derivation list uniquely defines the derived tuple. To maintain T, in response to an isolated insertion/deletion t_1^r into the stream R_1, we first compute T_1^r : $-t_1^r, R_2, R_3, \ldots, R_n,\ NOT\ S_1, \ldots, S_m$ (along with the derivation of each tuple in T_1^r) as follows. Essentially, in the join-computation phase, we compute and propagate partial results of $t_1^r \bowtie R_2 \bowtie \ldots \bowtie R_n$ (join of only the positive subgoals), and delete partial or complete results that match with a tuple from some S_j. Then, we add/subtract (for insertion/deletion t_1^r) the set of derivations in T_1^r from the original set of derivations in T. Similarly, to process an isolated insertion/deletion from S_1, we first compute $T_1^s = R_1, \ldots, R_n, t_1^s, NOT\ S_2, \ldots, NOT\ S_m$, and then add/subtract the set of derivations from the original set of derivations in T. To maintain T in face of simultaneous updates across the network, we use the following strategy.

- Suppose the difference between local clocks of any pair of nodes is bounded by $\Delta\tau_c$. For a tuple inserted into or deleted from a data stream, we start its join-computation phase after the completion of its storage phase, and wait for $\Delta\tau_c$ time, i.e., the join-computation phase starts after $\tau_s + \Delta\tau_c$ time. The inserted tuples are kept for $\tau_w + \tau_s + \tau_j + \Delta\tau_c$ time before expiration. For a deleted tuple, the tuple is first marked as "deleted" and physically deleted from the local memory after $\tau_w + \tau_s + \tau_j + \Delta\tau_c$ time.
- In the join phase, a new tuple with timestamp τ (insertion or deletion) only matches with tuples with timestamps before τ. The following theorem proves the correctness of the above strategy.

Theorem 54 *The above described strategy correctly maintains the query result*

$$T \texttt{:-}\ R_1, \ldots, R_n,\ NOT\ S_1, \ldots, NOT\ S_m,$$

in face of simultaneous updates to the given operand streams, under bounded message delays.

Proof. Let $R_1, \ldots, R_n, S_1, \ldots, S_m$ denote the sliding windows of the respective streams and $D = <r_1, r_2, \ldots, r_n>$ denote a possible derivation of some tuple t. Here, each r_i is a tuple-ID of a tuple in R_i, and r_i may be equal to r_j for $i \neq j$. We show that our described strategy for handling simultaneous updates correctly maintains the query-result, by showing that the insertion and deletion of D can be correctly maintained by handling updates of r_i and s_j.

Let B be the tuple (corresponding to an r_i or an s_j) with timestamp τ whose join-computation phase was completed the last among all such r_i's and s_j's. Such a tuple B exists, due to our update strategy. Since the join-computation phase of B starts at $\tau + \tau_s + \Delta\tau_c$, at that time, all inserted tuples with timestamp before τ have been stored, so B can find all matches. In particular,

- If B corresponds to an insertion of r_i, and no s_j' is matched, then D would be inserted; otherwise, no change happens.
- If B corresponds to an deletion of r_i, and no s_j' is matched, then D would be deleted; otherwise, no change happens.

- If B corresponds to an insertion of s_j, and all r_i's have been inserted, then D would be deleted; otherwise, no change happens.
- If B corresponds to an deletion of s_j, and all r_i's have been inserted, then D would be inserted; otherwise, no change happens.

In summary, the derivation D is indeed computed during the join-computation phase of B, and hence, correctly updated.

Together with Theorem 53, the evaluation strategy correctly maintains query results.

□

Multiple Rules with Common Head Predicate. In the above paragraphs, we have outlined a generalized Perpendicular Approach that maintains a query result represented by a single non-recursive deductive rule (with negated subgoals), in response to simultaneous insertions or deletions to operand tables. Such a scheme can be easily generalized to maintain a deductive program consisting of mul tiple deductive rules (with negation over base tables) with the same head predicate. Essentially, we assign a unique ID to each deductive rule, and include the rule-ID in the derivation of each result tuple. Then, maintenance of a program with multiple deductive rules becomes equivalent to maintaining each rule independently.

Incorporating Recursion. As mentioned before, the query-result tuples are hashed to appropriate locations in the network. Thus, the query result can be looked upon as a (derived) stream of tuples distributed across the network, with insertions and deletions occuring across the network. A tuple in the output result is considered to be generated/deleted at the hashed location only when the set of derivation changes from null to non-null and vice-versa. The above facilitates evaluation of recursive deductive rules, as long as the negation is only over base data streams, since the recursive subgoal can be treated just like any operand stream. Here, the derivation of a result tuple may contain the tuple ID of a tuple from the same table. However, the number of derivations of any tuple will still be finite (even for recursive programs).

General Single-Stratum Programs. Above, we have described that our scheme works for incremental evaluation of queries represented by (a union of) multiple deductive rules, with recursion and/or negation as long as the negation is over base streams. Generalization of the scheme and the correctness argument, for a general single-stratum deductive program is straightforward. In a single-stratum deductive program, there may be multiple deductive rules with negation and/or recursion, but the negation is only over the base data streams. Essentially, each defined IDB (derived view) is hashed/stored appropriately across the network, and treated as a data stream. The correctness of the approach follows from the previous result (Theorem 54) and the fact that the number of derivation of any tuple always remain finite (for programs without function symbols).

General Stratified (Multiple Strata) Logic Programs. It is interesting to note that the above scheme also works for general stratified (in particular, for XY-stratified) programs due to the following observation. Consider the set of predicates $\mathcal{P}_n$ being derived by the n^{th} stratum. By the definition of strata, each negated subgoals in (the rules defining) $\mathcal{P}_n$ is over a predicate from a lower stratum. Moreover, the lower-stratum predicates can be essentially looked upon as base predicates/tables for a higher-stratum. In other words, higher-strata predicates are essentially recursive programs with negation over only lower-strata predicates which can be considered as based tables for higher-strata predicates. Thus, higher-strata predicates can be maintained due to updates (insertions or deletions) in the lower-strata predicates exactly as outlined before. Note that the change in the set of derivations for each tuple in the lower-strata is *not* required to be propagated to the higher-strata; we only need to propagate actual insertions (when the set of derivations changes from null to non-null) and actual deletions (when the set of derivations changes to null) to the higher-strata predicates.

The above facilitates asynchronous computation of fixpoint, i.e., we don't need to wait for the fixpoint of lower-strata predicates to be reached (which never happens, due to the streaming base data) before evaluating higher-strata predicates. However, a deduced fact in a higher-strata predicate may have to be later retracted/deleted due to updates in the lower-strata; or, we could wait for certain time before "finalizing" a fact. The latter is acceptable/reasonable due to bounded-size sliding windows [36, 37] for streams and implicit temporal correlation in sensor data. Our correctness arguments and claims essentially guarantee that the fixpoint will eventually be reached if and when the streaming base data stops.

Other Generalizations. The above scheme can be easily generalized to handle built-in functions, since the evaluation of join-conditions and execution of built-in functions is done only locally. For the same reason, incorporating function symbols in deductive rules only requires extending the evaluation of join-condition using the term-matching operator. However, introduction of function symbols in deductive programs gives rise to many issues such as non-termination of programs, infinite derivation sets, difficulty in optimizing programs.

Aggregates are typically represented in logic rules by using the Prolog's all-solutions predicate to construct a list of values to be aggregated, then, computing the desired aggregate. However, an efficient *implementation* should aggregate the elements iteratively (for incremental aggregates) without actually constructing the list. Thus, we would use TAG [14] or fault-tolerant synopsis diffusion [19] techniques for incremental aggregates (without actually constructing the list). For non-incremental aggregates, we need to first construct the list.

5.3.2 System Architecture

In this subsection, we give an overall architecture of our system for in-network processing of logic queries, and address memory requirements of our system.

Overall Query Processing Architecture. Figure 50 depicts our overall system architecture and high-level plan of in-network evaluation of logic queries. Basically, the user specifies a deductive program, consisting of Datalog-like logic rules possibly, with recursion, XY-stratified negation, function symbols, lists and Prolog's all-solutions predicates (to represent aggregates succinctly). The user specified logic-program is first optimized using magic-set [161] transformations, and then translated into machine code which represents distributed bottom-up incremental evaluation of the given user program. The compiled code is then downloaded into each sensor node. Within each sensor node, there is a layer of in-network implementations of relational operators (such as join), aggregates, and built-in predicates/functions. The above layer is in addition to the usual layers of routing and networking layers.

Memory Requirements. Currently available sensor nodes (motes) have 4 to 10 KB of RAM and 128 KB or more of on-chip flash memory. The memory capacities have evolved over years [164], and latest Intel mote is being designed with 64 KB RAM [3]. In our system, the user program essentially consists of the generic join

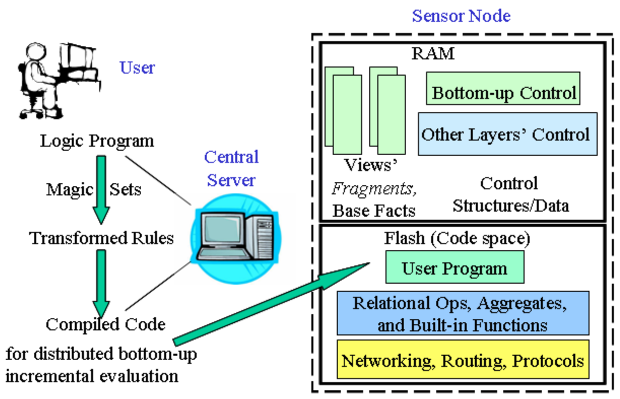


Figure 50: System Architecture.

interface, the list of join-conditions for the deductive rules, and code for the built-in functions. This is in addition to the other networking layers. A typical on-chip flash memory is large enough to easily contain the native code for various system layers and the user program.

Now, the strain on sensor nodes' main memory is due to (i) run-time control structures used by various system layers and the bottom-up approach, and (ii) materialization of views (i.e., storage of intermediate derived facts) during execution of user program. Note that the list of join-conditions are read-only part of the user program, and hence, can reside on the on-chip flash memory. During program execution, we need to load into RAM only one rule's join-condition list at a time. Thus, the run-time control structures are expected to take minimal main memory. The materialized views are stored in a distributed manner across the network. So, the *total main memory available for storing materialized views is the cumulative main memory of the entire network*. Thus, we expect the available main memory resources to be sufficient for most user programs. For instance, for the *logicJ* program of Figure 52(a), the materialized views are H, H', and J, and based on the storage scheme discussed in Section 5.4, each node y stores only tuples of the form $H(\_, y, \_)$, $J(x, \_)$ or $H'(y, \_)$ where x is a neighbor of y. Thus, the total number of tuples stored at any node is at most 2 to 3 times its degree. In a stable state, each node contains a single tuple of H. Note that in general the materialized views are required for communication efficiency and are inherent to the user program, rather than the programming framework. Since views are maintained as sliding windows, the space required for materialized views can be adjusted depending on the available memory and desired accuracy of results. In addition, the techniques for selection of views to materialize can be used to further satisfy the given memory constraints while maintaining sufficient accuracy of results.

Computation Load of Our Approach. Most of the processing in our system is in the form of distributed evaluation of logic rules or local built-in functions. The bottom-up evaluation of logic rules requires simple local operations such as term-matching [161], join-predicate evaluations, arithmetic comparisons, etc., and hence, result in minimal processing load. The processing load due to arithmetic-intensive local built-in functions is inherent to a user program, i.e., largely independent of the programming framework. Thus, our overall framework and approach is not expected to increase the processing load on the network.

5.4 System Implementation and Performance Evaluation

In this section, we present details of our current system implementation, and present performance results that illustrate the efficacy of our proposed approach and query evaluation techniques.

5.4.1 Current System Implementation

The main focus of our system is to automatically translate high-level user program written in form of deductive rules to nesC code that runs on individual sensor nodes. The generated code must represent our outlined query evaluation strategy. In particular, we translate a given user program into distributed nesC code as follows. First, we developed nesC interface components for various in-network join implementations corresponding to the Naive Broadcast, Local Storage, and Perpendicular Approaches as described in Chapter 2. These components reside on each node, and are very generic, i.e., do not need to be changed (or newly generated) for a specific user program. Any given user (deductive) program is now translated into the database schema (list of predicates and attributes) and the list of deductive rules (i.e., the list of subgoals and join conditions for each rule). The list of

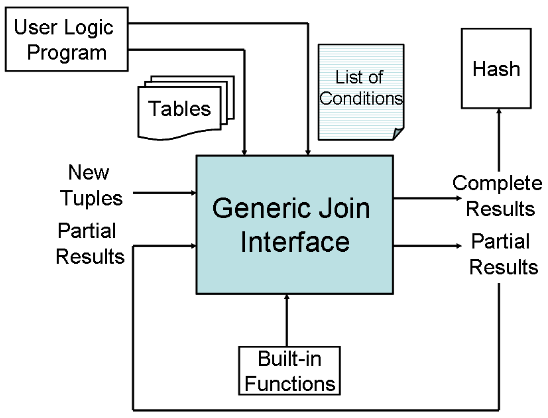


Figure 51: The *join interface* at a sensor node. Newly generated (base or derived) tuples are fed into the join interface, which generates partial and/or complete results by joining with local tables. The complete results are sent to the *storage interface* for hashing, while the partial results are forwarded to the next node on the vertical path (for the Perpendicular Approach). Partial results received from other nodes are treated similarly. In addition, newly generated tuples are also routed for storage.

rules and join-conditions are used by the generic join implementation to evaluate the predicates in the program. See Figure 51. Our current version of the system can handle general deductive programs without function symbols. In addition, the current implementation handles simple arithmetic built-in functions and predicates such as addition, subtraction, equality, less than, etc. In the current implementation, the hashing scheme of the derived results (i.e., the choice of join-attribute to use for geographic hashing) is given by the user.

The current implementation has been written in nesC and tested on TOSSIM, and the Perpendicular Approach join implementation is based on a 2D grid topology. In the immediate future implementations, we plan to (i) incorporate the generalized version of Perpendicular Approach join-implementation for arbitrary topologies, and (ii) incorporate use of arbitrary user-defined built-in functions (written in procedural code), aggregations, and function symbols.

Comparison of Program Sizes. In general, the deductive programs are expected to be much shorter and compact (few logic rules) compared to the corresponding nesC code. However, logic programs are sometimes non-intuitive to write. As such deductive framework is targeted towards expert and trained users, for whom the relief from worrying about low-level hardware and optimization issues would far offset the burden of writing a logic program. The size of the generated/translated nesC code is of not much relevance to the performance comparison – since the translation is done automatically and a typical flash memory of a sensor node is large enough to easily store the executable of resulting program code. In our framework, the generated code essentially includes the set of join conditions and the procedural code for user-defined built-in functions; the code for the join implementation is common to all user programs.

5.4.2 Performance Evaluation

In this subsection, we present our simulation results for implementation of one of the deductive program examples. In particular, we present our results for the *logicH* program of Example 50 for the shortest path tree. The *logicH* program of Example 50 incorporates quite a non-trivial combination of negation and recursion, and the resulting query evaluation algorithm is quite different from the native implementation.

Evaluations of other programs. Other examples of deductive programs from Section 5.2.2.1 naturally yield communication-optimal translations. For instance, in the program of Example 48, if we use a hashing scheme such that $P(i,\_,\_)$, $U(i,\_,\_)$ and $Z(i,\_,\_)$ are stored at node i, each tuple only costs one transmission (one broadcast to all neighbors), which results in essentially the same algorithm and performance as the distributed code in a procedural language. Similar argument and analysis holds for the other programs shown in Section 5.2.2.1 and also other programs.

Below, we start with discussing details of the distributed evaluation of *logicH* program, and then, present simulation results comparing the performance of the code translated from the deductive program with the procedural (nesC) program.

Distributed Evaluation of *logicH* Program. For convenience, we repeat the *logicH* program here; recall that *logicH* can handle general graphs with cycles.

logicH Program:

$$\begin{array}{ll}
H(A,A,0). & \\
H(A,x,1) & :-\ G(A,x) \\
H'(y,d+1) & :-\ H(\_,y,d'),(d+1)>d',H(\_,x,d),G(x,y) \\
H(x,y,d+1) & :-\ G(x,y),H(\_,x,d),\ NOT\ H'(y,d+1)
\end{array}$$

Hashing Schemes, and Join Strategy. The above *logicH* program produces a shortest-path tree rooted at node A. We assume that the fact $G(x,y)$ is available (stored) at both the nodes x and y, which essentially means that each node is aware of its immediate neighbors. Since y is the only join-attribute in $H(x,y,d)$, a tuple $H(x,y,d)$ is hashed to the node y. Similarly, a tuple $H'(y,d)$ is hashed to the node y. Based on the above hashing scheme, all pairs of joining tuples reside in neighboring nodes.

Naive-Broadcast Approach. The Naive-Broadcast Approach of evaluating the joins in the third and four rules on *logicH* entail that one of the tables be broadcast to neighboring nodes, while the other table be stored locally at each node. Thus, we broadcast and store each tuple $H(\_,y,\_)$ at all the neighbors of y. Thus, the above approach requires only one message transmission for each update (insertion/deletion of tuple) into H tables. The above hashing scheme and broadcast strategy means that $H'(y,\_)$ and $H(\_,y,\_)$ can derived at their hashed locations itself. Thus, the *only* communication cost incurred in the entire evaluation of the logic program is the replication of each H to the neighbors of the generating node.

Perpendicular Approach. In the Perpendicular Approach, tuples are stored and propagated for join-computation along horizontal/vertical paths. During the join-computation phase of a rule involving a negated subgoal, we first compute the complete result corresponding to the positive subgoals, and then, check for existence of tuples corresponding to the negated subgoals at the hashed location. For instance, in the case of the *logicH* program, $H(\_,y,d)$ is inserted after checking if there is a $H'(y,d)$ at y.

Distributed Evaluation. For the *logicH* program, initially, the node A generates the fact $H(A,A,0)$ using the first logic rule, and each neighbor x of A then generates a fact $H(A,x,1)$ using the second rule. Recall that derivation of a new fact is looked upon as generated at its *hashed* node. Thus, based on our hashing strategy, the fact $H(A,A,0)$ is considered to be generated at node A, and $H(A,x,1)$ is considered to be generated at node x. In the Naive-Broadcast approach suggested in previous paragraph, each insertion or deletion of an H tuple is broadcast to the neighbors of the generating node. Such a broadcast of an $H(\_,\_,l)$ tuple may result in new derivations of some $H'(\_,l+p)$ tuple ($p > 0$; due to the third logic rule) and/or some $H(\_,\_,l+1)$ tuple (due to the fourth logic rule). If the set of derivations of a tuple t becomes non-empty from empty in the above process, then the tuple t is considered to be an insertion to the corresponding table. Insertion of a $H'(\_,l)$ tuple may result in deletions of a tuple $H(\_,\_,l)$ due to the fourth logic rule. Since the given program is XY-stratified with finite strata (bounded by the diameter of the network), the above process is guaranteed to terminate to a fixed point.

Optimization. The *logicH* program for shortest path tree can be optimized by a simple aggregation or "pushing down projection." Note that the evaluation of the third and fourth logic rules in *logicH* is independent of the value of the first argument of the subgoal predicates H. Thus, we do not need to process an insertion $H(z,x,d)$, if there already exists a tuple $H(z',x,d)$. Thus, we need to only process insertions or deletions of $J(y,d)$ where $J(y,d) \ :- \ H(x,y,d)$. We can thus rewrite the *logicH* program as follows.

logicJ Program:

$$
\begin{array}{ll}
H(A,A,0). & \\
H(A,x,1) & :- \ G(A,x) \\
J(y,d) & :- \ H(x,y,d) \\
H'(y,d+1) & :- \ J(y,d'),(d+1) > d',J(x,d),G(x,y) \\
H(x,y,d+1) & :- \ G(x,y),J(x,d), \ NOT \ H'(y,d+1)
\end{array}
$$

Simulation Results. We now compare the performance of our translated/generated code for the *logicH* and *logicJ* programs with the optimized distributed code written in nesC.

Simulation Setup, Various Programs, and Performance Metrics. We run our simulations using the TOSSIM simulator on a sensor network with a grid topology. Unless being varied, the total number of nodes is chosen to be 49 (in a 7×7 grid network). In certain simulations, we vary the message loss probability to compare the robustness of various approaches. We simulate a message loss probability of P by ignoring a message at the receiver with a probability of P.

We compare the performance of various programs using two performance metrics, viz., the *result inaccuracy* and *total communication cost.* Here, we define the *result inaccuracy* as the ratio of the number of missed shortest paths over the total number of shortest paths computed by a centralized program.

In our simulations, we compare the performance of three programs: the optimized distributed nesC code, generated code for the *logicH* program, and generated code for the *logicJ* program. In both the above generated codes, we use the Naive-Broadcast Approach of join computation, because of just 1-hop spatial constraint of the joins involved. In addition, we show the effectiveness of the Perpendicular Approach of computing join by simulating the programs for larger "transitivity factor."

Varying Message Loss Probability. In this first set of experiments, we vary the message loss probability and compare the result inaccuracy of various programs in networks with different size. First, we confirmed that when there are no message losses, the accuracy of the result is 100% for all the programs. In Figure 52, we observe that the generated code for the *logicH* and *logicJ* programs (with Naive-Broadcast Approach of join computation) compute about 80% correct shortest paths for message loss probability up to 10%. The result inaccuracy continues to increase with increase in message loss probability. However, we observe that the performace of the translated code is close to the procedural code, and logicH is even more robust (i.e., have a lower result inaccuracy value) than procedural code for small values of message loss probability.

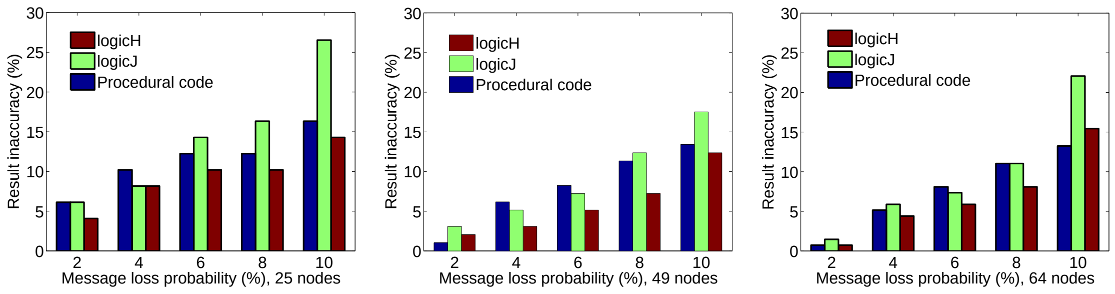


Figure 52: Effect of message loss probability on result inaccuracy, for varying network sizes.

Varying Network Size. In Figure 53, we plot the total communication cost incurred by various programs for varying network size. We observe that the total communication cost incurred by all programs is largely proportional to the total number of nodes in the network. Moreover, the generated codes for *logicH* and *logicJ* programs (with Naive-Broadcast Approach) perform close to the procedural code, with *logicJ* performing much better . The communication cost incurred by *logicJ* is about twice as that of procedural code, due to the insertions and deletions of $J(y,d)$.

The total communication cost of all approaches deceases when the message loss probability increases.

Varying Transitive Factor. In this set of experiments, we modify the various programs to compute shortest-path tree in G^k (for various k), where G is the network graph and G^k is defined as the graph wherein there is an edge between any two nodes x and y that are within k-hops in the network graph G. We refer to k as the *transitive factor*. Note that here we are only changing the definition of what a shortest-path tree is, while keeping the network graph (transmission radii and neighborhoods of each node) the same. Our logic programs can be easily changed to reflect the above. In Figure 54, we show the effect of k on the total communication cost incurred by various programs for computing the shortest-path tree in G^k for varying k. We focus on comparing the performance of Perpendicular and Naive-Broadcast here. We notice that the communication cost of both approaches increases with increase in k. This is because with the increase in k, the number of paths with shortest length increases which results in more number of $H(x,y,d)$ tuples for the same y and d. However, PA grows slower than Naive-Broadcast, and eventually beats Naive-Broadcast when $k > 4$, which suggests that PA is more suitable for join operators that involve far-apart tuple matching.

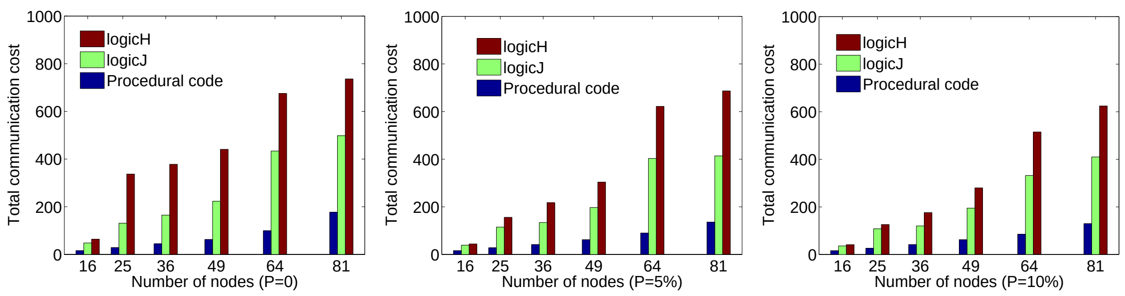


Figure 53: Total communication cost incurred by various programs for varying network size, for three different message loss probabilities.

Summary of Simulation Results. From the above simulations, we can see that the code translated from the logic program would produce correct results, at the same time, results in similar communication cost as the native implementations. For applications with local join, Naive Broadcast would be the best choice, while PA is expected to outperform other approaches in a large scale network with multi-hop

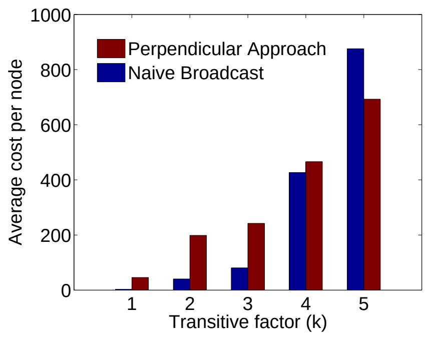


Figure 54: Average communication cost per node incurred by the generated code using Naive-Broadcast and Perpendicular Approach for varying transitive factor.

tuple matching. One additional feature of the deductive approach is its robustness, which is an important factor in failure-prone sensor networks. In addition, we observed that the main-memory used at each node for the *logicH* and *logicJ* programs were minimal (8-10 tuples per node, with at most two derivations per tuple).

5.5 Discussions

We proposed and motivated the deductive framework, and designed a full-fledge query engine for in-network evaluation of deductive queries in a sensor network. We presented implementation details of our system that compiles a given user deductive program into distributed code that runs on individual nodes. There are many challenges that need to be addressed for an optimized (in terms of main memory usage and communication efficiency) implementation of an in-network deductive query engine, including (i) Efficient implementation of the counting approach for incremental maintenance of join queries; such an implementation is unlikely to be fully accurate but will have minimal space overhead, (ii) Automatic determination of attributes to use for hashing derived results to minimize overall communication cost, (iii) Efficient implementation of the Rederivation approach of [163] which will pave the way of in-network evaluation of general deductive programs, with locally-stratified [159] negation, and (iv) The problem of selection of views to materialize in the context of sensor networks. The above challenging issues are of great interest to us, and will be addressed in future works.

Chapter 6

Conclusions

Wireless sensor network is a rapid growing area. It contains rich research problems, requires multi-disciplinary knowledge and enables collaboration among multiple fields, including networking, database, geometry, signal processing and security, etc.

This dissertation focused on exploring the full potentials of sensor networks as data processing engines by investigating on key challenges in collaborative processing and query evaluation. We remark that the techniques we discussed in this dissertation aim to provide high-level abstract of functionalities of sensor networks as collaborative processing engines. While the sensor networks become more and more heterogeneous with a mix of various devices of different capabilities, our techniques should be thought of as virtual abstracts that can be implemented on top of a hierarchical and heterogeneous physical network. More powerful nodes can easily simulate these algorithms and act as proxies for sets of mote-level nodes.

Information processing is a fundamental problem for sensor networks due to its inherited data-centric feature. With the flourish of new novel applications, it is worth continuing on improving the accessibility, interactivity and shareability of sensor data, to server more diverse communities of end users. Along that direction, lots of interesting problems are worth pursuit, e.g., multi-query optimization, mobility-aided information processing, security and privacy issues of data communication and sharing, etc. The techniques proposed in this dissertation form foundations for future researches and exploration.

Bibliography

[1] “Crossbow technology, inc.” http://www.crossbow.com.

[2] “Sun spot project.” http://www.sunspotworld.com/.

[3] Intel Research, “Intel mote,” http://www.intel.com/research/exploratory/motes.htm.

[4] G. J. Pottie and W. J. Kaiser, “Wireless integrated network sensors,” *Communications of the ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[5] D. Estrin, R. Govindan, and J. Heidemann, “Embedding the internet: Introduction,” *Communications of the ACM*, vol. 43, no. 5, pp. 38–41, May 2000.

[6] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, “Next century challenges: Scalable coordination in sensor networks,” in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, August 1999, pp. 263–270.

[7] D. Culler, D. Estrin, and M. Srivastava, “Guest editors’ introduction: Overview of sensor networks,” *IEEE Computer*, vol. 37, no. 8, pp. 41–49, August 2004.

[8] “Motive, inc.” http://www.moteiv.com/.

[9] “Nokia’s sensorplanet. http://www.sensorplanet.org.”

[10] “Microsoft sensormap. http://atom.research.microsoft.com/sensormap/.”

[11] “http://www.greatduckisland.net/.”

[12] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, “An analysis of a large scale habitat monitoring application,” in *SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 214–226.

[13] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, “Fidelity and yield in a volcano monitoring sensor network,” in *OSDI ’06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 381–396.

[14] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “Tag: a tiny aggregation service for ad-hoc sensor networks,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[15] Y. Yao and J. Gehrke, “The cougar approach to in-network query processing in sensor networks,” *ACM SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[16] Y. Yao and J. E. Gehrke, “Query processing in sensor networks,” in *CIDR’03: Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[17] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler, “The tenet architecture for tiered sensor networks,” in *SenSys ’06: Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 153 – 166.

[18] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, “Medians and beyond: new aggregation techniques for sensor networks,” in *SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 239–249.

[19] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson, “Synopsis diffusion for robust aggregation in sensor networks,” in *SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 250–262.

[20] H. Breu and D. G. Kirkpatrick, “Unit disk graph recognition is np-hard,” *Comput. Geom. Theory Appl.*, vol. 9, no. 1-2, pp. 3–24, 1998.

[21] L. Barriére, P. Fraigniaud, and L. Narayanan, “Robust position-based routing in wireless ad hoc networks with unstable transmission ranges,” in *DIALM ’01: Proceedings of the 5th international workshop on Discrete algorithms and methods for mobile computing and communications*, 2001, pp. 19–27.

[22] F. Kuhn and A. Zollinger, “Ad-hoc networks beyond unit disk graphs,” in *DIALM-POMC ’03: Proceedings of the 2003 joint workshop on Foundations of mobile computing*, 2003, pp. 69–78.

[23] P. Levis, N. Lee, M. Welsh, and D. Culler, “Tossim: accurate and scalable simulation of entire tinyos applications,” 2003.

[24] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, “Processing complex aggregate queries over data streams,” in *SIGMOD ’02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 61–72.

[25] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, “The design of an acquisitional query processor for sensor networks,” in *SIGMOD’03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 491–502.

[26] D. Donoho, “Compressed sensing,” *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[27] P. Bonnet, J. Gehrke, and P. Seshadri, “Towards sensor database systems,” in *MDM ’01: Proceedings of the Second International Conference on Mobile Data Management*, 2001, pp. 3–14.

[28] D. A. Schneider and D. J. DeWitt, “A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment,” vol. 18, no. 2, 1989, pp. 110–121.

[29] K.-L. Tan and H. Lu, “Processing multi-join query in parallel systems,” in *SAC ’92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, 1992, pp. 283–292.

[30] J. P. Richardson, H. Lu, and K. Mikkilineni, “Design and evaluation of parallel pipelined join algorithms,” vol. 16, no. 3, 1987, pp. 399–409.

[31] H. Lu, M.-C. Shan, and K.-L. Tan, “Optimization of multi-way join queries for parallel execution,” in *VLDB ’91: Proceedings of the 17th International Conference on Very Large Data Bases*, 1991, pp. 549–560.

[32] X. Zhu, H. Gupta, and B. Tang, “Join of multiple data streams in sensor networks,” Stony Brook University, Tech. Rep., 2007, http://www.cs.sunysb.edu/~hgupta/ps/joinSN.pdf.

[33] K. Fall and K. Varadhan, “The *ns* manual,” available from http://www-mash.cs.berkeley.edu/ns/.

[34] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, “GHT: A geographic hash table for data-centric storage in sensornets,” in *Proc. 1st ACM Workshop on Wireless Sensor Networks ands Applications*, 2002, pp. 78–87.

[35] A. Arasu, S. Babu, and J. Widom, “The cql continuous query language: semantic foundations and query execution,” *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[36] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management,” *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[37] L. Ding, N. Mehta, E. Rundensteiner, and G. Heineman, “Joining punctuated streams,” in *EDBT’04: 9th International Conference on Extending Database Technology*, 2004, pp. 587–604.

[38] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, “Model-based approximation querying in sensor networks,” *VLDB Journal*, no. 4, pp. 417–443, 2005.

[39] H. Gupta, V. Navda, S. R. Das, and V. Chowdhary, “Efficient gathering of correlated data in sensor networks,” in *MobiHoc ’05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, 2005, pp. 402–413.

[40] H. G. et al., “Deductive approach for programming for sensor networks,” Stony Brook University, Tech. Rep., 2007, http://www.cs.sunysb.edu/~hgupta/ps/logicSN.pdf.

[41] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, “The design and implementation of a declarative sensor network system,” in *SenSys ’07: Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007, pp. 175–188.

[42] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, “Declarative routing: extensible routing with declarative queries,” in *SIGCOMM ’05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 2005, pp. 289–300.

[43] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, “Declarative networking: language, execution and optimization,” in *SIGMOD ’06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 97–108.

[44] S. Li, S. H. Son, and J. A. Stankovic, “Event detection services using data service middleware in distributed sensor networks,” pp. 502–517, 2003.

[45] L. Guibas, “Sensing, tracking, and reasoning with relations,” *IEEE Signal Processing Magazine*, vol. 19, no. 2, 2002.

[46] R. Govindan, J. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker, “The sensor network as a database,” University of Southern California, Computer Science Department, Technical Report, 2002.

[47] S. Madden and J. M. Hellerstein, “Distributing queries over low-power wireless sensor networks,” in *SIGMOD ’02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 622–622.

[48] B. Bonfils and P. Bonnet, “Adaptive and decentralized operator placement for in-network query processing,” in *Proceedings of Information Processing in Sensor Networks*, 2003, pp. 47–62.

[49] D. J. Abadi, S. Madden, and W. Lindner, “Reed: robust, efficient filtering and event detection in sensor networks,” in *VLDB ’05: Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 769–780.

[50] H. Gupta and V. Chowdhary, “Communication-efficient implementation of join in sensor networks,” *Ad Hoc Networks*, vol. 5, no. 6, pp. 929–942, 2007.

[51] A. Pandit and H. Gupta, “Communication efficient implementation of range-joins in sensor networks,” in *DASFAA’06: International Conference on Database Systems for Advanced Applications*, 2006, pp. 859–869.

[52] S. Y. Cheung, M. H. Ammar, and M. Ahamad, “The grid protocol: A high performance scheme for maintaining replicated data,” *IEEE Transaction on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 582–592, 1992.

[53] T. W. Yan and H. Garcia-Molina, “The sift information dissemination system,” *ACM Transactions on Database Systems*, vol. 24, no. 4, pp. 529–565, 1999.

[54] X. Liu, Q. Huang, and Y. Zhang, “Combs, needles, haystacks: balancing push and pull for discovery in large-scale sensor networks,” in *SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 122–133.

[55] R. Sarkar, X. Zhu, and J. Gao, “Double rulings for information brokerage in sensor networks,” in *MobiCom’06: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, September 2006, pp. 286–297.

[56] R. M. et al., “Query processing, resource management, and approximation in a data stream management,” in *CIDR’03: First Biennial Conference on Innovative Data Systems Research*, 2003.

[57] D. Carney, U. *C*Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, “Monitoring streams: a new class of data management applications,” in *VLDB ’02: Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 215–226.

[58] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, “TelegraphCQ: Continuous dataflow processing,” in *SIGMOD’03: ACM SIGMOD international conference on Management of data*, 2003, pp. 668–668.

[59] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, “NiagaraCQ: a scalable continuous query system for internet databases,” in *SIGMOD’00: ACM SIGMOD international conference on Management of data*, 2000, pp. 379–390.

[60] A. Das, J. Gehrke, and M. Riedewald, “Approximate join processing over data streams,” in *SIGMOD ’03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 40–51.

[61] M. H. et al., “A stream database server for sensor applications,” Purdue University, Tech. Rep., 2002.

[62] S. Madden and M. Franklin, “Fjording the stream: An architecture for queries over streaming sensor data,” in *ICDE’02: 18th International Conference on Data Engineering*, 2002, pp. 555–566.

[63] N. Bulusu, J. Heidemann, and D. Estrin, “GPS-less low cost outdoor localization for very small devices,” *IEEE Personal Communications Magazine*, vol. 7, no. 5, pp. 28–34, 2000.

[64] B. Karp and H. Kung, “GPSR: Greedy perimeter stateless routing for wireless networks,” in *MobiCom ’00: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, 2000, pp. 243–254.

[65] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, “Locality-sensitive hashing scheme based on *p*-stable distributions,” in *SCG ’04: Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253–262.

[66] J. Wu and J. Cao, “Connected k-hop clustering in ad hoc networks,” in *ICPP ’05: Proceedings of the 2005 International Conference on Parallel Processing*, 2005, pp. 373–380.

[67] T. Ibaraki and T. Kameda, “On the optimal nesting order for computing n-relational joins,” *ACM Transactions on Database Systems*, vol. 9, no. 3, pp. 482–502, 1984.

[68] C. Intanagonwiwat, R. Govindanj, and D. Estrin, “Directed diffusion: A scalable and robust communication paradigm for sensor networks,” in *MobiCom’00: Proceeding of the 6th Annual Int’l Conference on Mobile Computing and Networking*, August 2000, pp. 56–67.

[69] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin, “Data-centric storage in sensornets,” *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 137–142, 2003.

[70] J. Li, J. Jannotti, D. Decouto, D. Karger, and R. Morris, “A scalable location service for geographic ad-hoc routing,” in *Proceedings of 6th ACM/IEEE International Conference on Mobile Computing and Networking*, 2000, pp. 120–130.

[71] X. Li, Y. J. Kim, R. Govindan, and W. Hong, "Multi-dimensional range queries in sensor networks," in *Proceedings of the first international conference on Embedded networked sensor systems*. ACM Press, 2003, pp. 63–75.

[72] S. Funke, L. Guibas, A. Nguyen, and Y. Wang, "Distance-sensitive routing and information brokerage in sensor networks," in *DCOSS'06: Proceedings of IEEE International Conference on Distributed Computing in Sensor Systems*, July 2006, pp. 234–251.

[73] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang, "A two-tier data dissemination model for large-scale wireless sensor networks," in *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*. New York, NY, USA: ACM Press, 2002, pp. 148–159.

[74] I. Stojmenovic, "A routing strategy and quorum based location update scheme for ad hoc wireless networks," SITE, University of Ottawa, Tech. Rep. TR-99-09, September, 1999.

[75] Q. Fang, J. Gao, and L. J. Guibas, "Landmark-based information storage and retrieval in sensor networks," in *INFOCOM'06: The 25th Conference of the IEEE Communication Society*, April 2006, pp. 1–12.

[76] D. Braginsky and D. Estrin, "Rumor routing algorthim for sensor networks," in *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, 2002, pp. 22–31.

[77] B. Nath and D. Niculescu, "Routing on a curve," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 155–160, 2003.

[78] H. S. M. Coxeter, *Introduction to Geometry*, 2nd ed. New York: John Wiley & Sons, 1969.

[79] P. Samuel, *Projective Geometry*. New York: Springer-Verlag, 1988.

[80] Q. Fang, J. Gao, and L. Guibas, "Locating and bypassing routing holes in sensor networks," in *Mobile Networks and Applications*, vol. 11, 2006, pp. 187–200.

[81] Y. Wang, J. Gao, and J. S. B. Mitchell, "Boundary recognition in sensor networks by topological methods," in *MobiCom'06: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, September 2006, pp. 122–133.

[82] D. Jea, A. A. Somasundara, and M. B. Srivastava, "Multiple controlled mobile elements (data mules) for data collection in sensor networks." in *IEEE/ACM Int'l Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005, pp. 244–257.

[83] W. Lindner and S. Madden, "Data management issues in periodically disconnected sensor networks," in *Proceedings of Workshop on Sensor Networks at Informatik*, 2004.

[84] Z. Vincze and R. Vida, "Multi-hop wireless sensor networks with mobile sink," in *CoNEXT'05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*. New York, NY, USA: ACM Press, 2005, pp. 302–303.

[85] A. Kansal, M. Rahimi, W. J. Kaiser, M. B. Srivastava, G. J. Pottie, and D. Estrin, "Controlled mobility for sustainable wireless networks," in *IEEE Sensor and Ad Hoc Communications and Networks (SECON'04)*, 2004, pp. 1–6.

[86] R. Shah, S. Roy, S. Jain, and W. Brunette, "Data MULEs: Modeling a three-tier architecture for sparse sensor networks," in *IEEE SNPA Workshop*, May 2003, pp. 30–41.

[87] E. M. Arkin and R. Hassin, "Approximation algorithms for the geometric covering salesman problem," *Discrete Appl. Math.*, vol. 55, no. 3, pp. 197–218, 1994.

[88] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker, "DIFS: A distributed index for features in sensor networks," in *Proceedings of First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003, pp. 163–173.

[89] J. Bruck, J. Gao, and A. Jiang, “MAP: Medial axis based geometric routing in sensor networks,” in *MobiCom ’05: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, August 2005, pp. 88–102.

[90] L. J. Guibas, “Sensing, tracking and reasoning with relations,” *IEEE Signal Processing Magazine*, vol. 19, no. 2, pp. 73–85, March 2002.

[91] F. Zhao, J. Shin, and J. Reich, “Information-driven dynamic sensor collaboration,” *IEEE Signal Processing Magazine*, vol. 19, no. 2, pp. 61–72, 2002.

[92] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus, “Tracking a moving object with a binary sensor network,” in *SenSys ’03: Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003, pp. 150–161.

[93] W. Kim, K. Mechitov, J.-Y. Choi, and S. Ham, “On target tracking with binary proximity sensors,” in *IPSN ’05: Proceedings of the 4th international symposium on Information processing in sensor networks*, 2005, p. 40.

[94] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh, “Vigilnet: An integrated sensor network system for energy-efficient surveillance,” *ACM Transactions on Sensor Networks*, vol. 2, no. 1, pp. 1–38, 2006.

[95] N. Shrivastava, R. M. U. Madhow, and S. Suri, “Target tracking with binary proximity sensors: fundamental limits, minimal descriptions, and algorithms,” in *SenSys ’06: Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 251–264.

[96] J. Liu, P. Cheung, L. Guibas, and F. Zhao, “Apply geometric duality to energy efficient non-local phenomenon awareness using sensor networks,” *IEEE Wireless Communication Magazine, special issue on Wireless Sensor Networks: Theory and Systems*, december 2004.

[97] S. Funke and C. Klein, "Hole detection or: how much geometry hides in connectivity?" in *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, 2006, pp. 377–385.

[98] S. Funke, "Topological hole detection in wireless sensor networks and its applications," in *DIALM-POMC '05: Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing*, 2005, pp. 44–53.

[99] S. P. Fekete, A. Kröller, D. Pfisterer, S. Fischer, and C. Buschmann, "Neighborhood-based topology recognition in sensor networks," in *ALGOSENSORS*, ser. Lecture Notes in Computer Science, vol. 3121, 2004, pp. 123–136.

[100] S. P. Fekete, M. Kaufmann, A. Kröller, and N. Lehmann, "A new approach for boundary recognition in geometric sensor networks," in *Proceedings 17th Canadian Conference on Computational Geometry*, 2005, pp. 82–85.

[101] A. Kröller, S. P. Fekete, D. Pfisterer, and S. Fischer, "Deterministic boundary recognition and topology extraction for large sensor networks," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2006, pp. 1000–1009.

[102] R. Ghrist and A. Muhammad, "Coverage and hole-detection in sensor networks via homology," in *IPSN'05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, 2005, pp. 254–260.

[103] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek, "Beyond average: Toward sophisticated sensing with queries," in *IPSN'03: Proceedings of Information Processing in Sensor Networks*, April 2003, pp. 63–79.

[104] S. Gandhi, J. Hershberger, and S. Suri, "Approximate isocontours and spatial summaries for sensor networks," in *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, 2007, pp. 400–409.

[105] A. Hatcher, *Algebraic Topology*. Cambridge University Press, 2002.

[106] J. S. B. Mitchell, “A new algorithm for shortest paths among obstacles in the plane,” *Annals of Mathematics and Artificial Intelligence*, vol. 3, no. 1, pp. 83–105, 1991.

[107] “Contour tracking vedios: http://www.cs.sunysb.edu/∼xjzhu/contour.html.”

[108] R. Sarkar, X. Zhu, J. Gao, L. J. Guibas, and J. S. B. Mitchell, “Iso-contour queries and gradient routing with guaranteed delivery in sensor networks,” in *INFOCOM’08: Proceedings of the 27th Annual IEEE Conference on Computer Communications*, April 2008, pp. 960–967.

[109] M. Chu, H. Haussecker, and F. Zhao, “Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks,” *International Journal of High Performance Computing Applications*, vol. 16, no. 3, pp. 90–110, 2002.

[110] J. Liu, F. Zhao, and D. Petrovic, “Information-directed routing in ad hoc sensor networks,” *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 4, pp. 851–861, April 2005.

[111] J. Faruque and A. Helmy, “Rugged: Routing on fingerprint gradients in sensor networks,” in *ICPS ’04: Proceedings of the The IEEE/ACS International Conference on Pervasive Services*, 2004, pp. 179–188.

[112] J. Faruque, K. Psounis, and A. Helmy, “Analysis of gradient-based routing protocols in sensor networks,” in *DCOSS’05: IEEE/ACM Internationl Conference on Distributed Computing in Sensor Systems*, 2005.

[113] F. Ye, G. Zhong, S. Lu, and L. Zhang, “GRAdient Broadcast: A robust data delivery protocol for large scale sensor networks,” *ACM Wireless Networks (WINET)*, vol. 11, no. 2, March 2005.

[114] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore, “Contour trees and small seed sets for isosurface traversal,” in *SCG ’97: Proceedings of the thirteenth annual symposium on Computational geometry*, 1997, pp. 212–220.

[115] J. W. Milnor, *Morse Theory*. Princeton, NJ: Princeton University Press, 1963.

[116] M. de Berg and M. van Kreveld, "Trekking in the alps without freezing or getting tired," *Algorithmica*, vol. 18, pp. 306–323, 1997.

[117] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," in *SODA'00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, 2000, pp. 918–926.

[118] S. P. Tarasov and M. N. Vyalyi, "Construction of contour trees in 3D in $O(n \log n)$ steps," in *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, 1998, pp. 68–75.

[119] P. Skraba, Q. Fang, A. Nguyen, and L. Guibas, "Sweeps over wireless sensor networks," in *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, 2006, pp. 143–151.

[120] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia, "Routing with guaranteed delivery in ad hoc wireless networks," *Wireless Networks*, vol. 7, no. 6, pp. 609–616, 2001.

[121] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger, "Geometric ad-hoc routing: Of theory and practice," in *PODC '03: Proceedings of 22nd ACM Int. Symposium on the Principles of Distributed Computing*, 2003, pp. 63–72.

[122] Q. Fang, J. Gao, L. Guibas, V. de Silva, and L. Zhang, "GLIDER: Gradient landmark-based distributed routing for sensor networks," in *INFOCOM '05: Proceedings of the 24th Conference of the IEEE Communication Society*, vol. 1, March 2005, pp. 339–350.

[123] B. A. Bash, J. W. Byers, and J. Considine, "Approximately uniform random sampling in sensor networks," in *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, 2004, pp. 32–39.

[124] A. G. Dimakis, A. D. Sarwate, and M. J. Wainwright, “Geographic gossip: efficient aggregation for sensor networks,” in *IPSN ’06: Proceedings of the fifth international conference on Information processing in sensor networks*, 2006, pp. 69–76.

[125] T. K. Dey, J. Giesen, and S. Goswami, “Shape segmentation and matching with flow discretization,” in *WADS ’03: Proceedings of workshop on Algorithms and Data Structures*, 2003, pp. 25–36.

[126] T. Sebastian, P. Klein, and B. Kimia, “Recognition of shapes by editing shock graphs,” in *ICCV ’01: Proceedings of International Conference on Computer Vision*, 2001, pp. 755–762.

[127] K. Siddiqi, A. Shokoufandeh, S. J. Dickinson, and S. W. Zucker, “Shock graphs and shape matching,” in *ICCV ’98: Proceedings of International Conference on Computer Vision*, 1998, pp. 222–229.

[128] F. F. Leymarie and B. B. Kimia, “The shock scaffold for representing 3d shape,” in *Proceedings of 4th International Workshop Visual Form*, 2001, pp. 216–228.

[129] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. L. Kunii, “Topology matching for fully automatic similarity estimation of 3d shapes,” in *SIGGRAPH ’01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 203–212.

[130] S. Funke and N. Milosavljevic, “Network sketching or: How much geometry hides in connectivity?–part ii,” in *SODA ’07: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007, pp. 958–967.

[131] H. I. Choi, S. W. Choi, and H. P. Moon, “Mathematical theory of medial axis transform,” *Pacific Journal of Mathematics*, vol. 181, no. 1, pp. 57–88, 1997.

[132] A. Lieutier, “Any open bounded subset of R^n has the same homotopy type than its medial axis,” in *SM ’03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, 2003, pp. 65–75.

[133] J. Elson, “Time synchronization in wireless sensor networks,” Ph.D. dissertation, University of California, Los Angeles, May 2003.

[134] S. Ganeriwal, R. Kumar, and M. B. Srivastava, “Timing-sync protocol for sensor networks,” in *SenSys ’03: Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003, pp. 138–149.

[135] J. N. Al-Karaki and A. E. Kamal, “Routing techniques in wireless sensor networks: a survey,” *IEEE Wireless Communications*, vol. 11, no. 6, pp. 6–28, 2004.

[136] Y. Shi and Y. T. Hou, “Approximation algorithm for base station placement in wireless sensor networks,” in *SECON ’07: Proceedings of IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2007, pp. 512–519.

[137] R. Chandra, L. Qiu, K. Jain, and M. Mahdian, “Optimizing the placement of integration points in multi-hop wireless networks,” in *ICNP ’04: Proceedings of International Conference on Network Protocols*, 2004, pp. 271–282.

[138] J. B. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[139] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran, “Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes,” in *Proc. Symposium on Information Processing in Sensor Networks (IPSN’05)*, April 2005, pp. 111–117.

[140] P. Skraba, Q. Fang, A. Nguyen, and L. Guibas, “Sweeps over wireless sensor networks,” in *IPSN ’06: Proceedings of the fifth international conference on Information processing in sensor networks*, 2006, pp. 143–151.

[141] D. C. et al., “TinyOS,” http://www.tinyos.net, 2004.

[142] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” in

Proceedings of Programming Language Design and Implementation (PLDI), 2003.

[143] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004, pp. 3–3.

[144] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2004, pp. 99–110.

[145] S. R. Madden, J. M. Hellerstein, and W. Hong, "TinyDB: In-network query processing in tinyos," http://telegraph.cs.berkeley.edu/tinydb, 2003.

[146] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *DCOSS'05: International Conference on Distributed Computing in Sensor Systems*, 2005.

[147] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 75–90, 2005.

[148] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992, pp. 256–266.

[149] R. Newton and M. Welsh, "Region streams: functional macroprogramming for sensor networks," in *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, 2004, pp. 78–87.

[150] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: a programming model for event-driven embedded systems," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, 2003, pp. 698–704.

[151] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and

A. Wood, “Envirotrack: Towards an environmental computing paradigm for distributed sensor networks,” in *ICDCS’04: Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004.

[152] A. Bakshi, J. Ou, and V. K. Prasanna, “Towards automatic synthesis of a class of application-specific sensor networks,” in *CASES ’02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002, pp. 50–58.

[153] S. Madden, R. Szewczyk, M. Franklin, and D. Culler, “Supporting aggregate queries over ad-hoc wireless sensor networks,” in *Workshop on Mobile Computing and Systems Applications*, 2002.

[154] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “The design of an acquisitional query processor for sensor networks,” in *SIGMOD’03: ACM SIGMOD international conference on Management of data*, 2003, pp. 491–502.

[155] S. Madden and J. M. Hellerstein, “Distributing queries over low-power wireless sensor networks,” in *SIGMOD’02: ACM SIGMOD international conference on Management of data*, 2002, pp. 622–622.

[156] S. A. Tarnlund, “Horn clause computability,” *BIT*, vol. 17, no. 2, 1977.

[157] J. Reich, J. Liu, and F. Zhao, “Collaborative in-network processing for target tracking,” in *European Association for Signal, Speech and Image Processing*, 2002.

[158] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases*. Addison-Wesley Publishing Co., Inc., 1995.

[159] P. Cholak and H. A. Blair, “The complexity of local stratification,” *Fundamenta Informaticae*, vol. 21, no. 4, 1994.

[160] C. Zaniolo, N. Arni, and K. Ong, “Negation and aggregates in recursive rules: the LDL++ approach,” in *Deductive and Object-Oriented Databases*, 1993, pp. 204–221.

[161] J. D. Ullman, *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. New York, NY, USA: W. H. Freeman & Co., 1990.

[162] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, “Controlling the search in bottom-up evaluation,” in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992, pp. 273–287.

[163] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, “Maintaining views incrementally,” in *SIGMOD ’93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 157–166.

[164] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, “The mote revolution: Low power wireless sensor network devices,” in *Proceedings of Hot Chips 16: A Symposium on High Performance Chips*, 2004.