Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

4Sensing - Decentralized Processing for Participatory Sensing Data

(Heitor José Simões Baptista Ferreira nº 29207)

2º Semestre de 2009/10

16 de Junho de 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

4Sensing - Decentralized Processing for Participatory Sensing Data

(Heitor José Simões Baptista Ferreira nº 29207)

Orientador: Prof. Doutor Sérgio Marco Duarte
Co-orientador: Prof. Doutor Nuno Manuel Ribeiro Preguiça

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

2º Semestre de 2009/10
16 de Junho de 2010

Acknowledgements

I would like to express my sincere gratitude to all those that contributed to this dissertation:

First of all I would like to thank Professor Sérgio Duarte and Professor Nuno Preguiça for their attentive supervision and clear-minded guidance without whom this work would not be possible.

My thanks also go to my colleagues and Professors in DI/FCT/UNL for the friendly and supportive environment that made my stay in this Faculty enjoyable and productive.

I am also deeply grateful to my family and friends for all the support, encouragement and helping me keep a good sanity level.

Finally, I want to acknowledge the support from the project PTDC/EIA/76114/2006, Live-Feeds - Disseminação P2P de Conteúdos Web Syndication, that funded part of this work.

Resumo

Participatory sensing representa um novo paradigma aplicacional, impulsionado por factores sociais e tecnológicos, que ganha actualmente importância como domínio de investigação. A sua aplicação é potenciada pela crescente disponibilidade de telemóveis equipados com sensores, como a câmara, GPS e acelerómetro, permitindo a aquisição e agregação de dados por iniciativa de um conjunto de utilizadores, cobrindo áreas alargadas sem os custos associados a uma rede de sensores de larga escala.

Embora exista um conjunto significativo de trabalhos de investigação nesta área, as soluções apresentadas assentam normalmente sobre sistemas centralizados. Nem sempre esta é uma abordagem indicada para este tipo de aplicações, por implicar o armazenamento de dados privados sob um único domínio administrativo e também pelos custos financeiros que exige não serem adequados a iniciativas comunitárias.

Esta dissertação foca os aspectos da gestão de dados em aplicações *participatory sensing* sobre uma infra-estrutura distribuída - e em particular o seu processamento - tomando em conta áreas relacionadas como redes de sensores, gestão de dados em sistemas *peer-to-peer* e processamento de *streams*. A solução apresentada aborda um conjunto de requisitos comuns de gestão de dados, desde a aquisição, processamento, armazenamento e pesquisa com o objectivo de facilitar o desenvolvimento de aplicações.

São propostas três arquitecturas alternativas - RTree, QTree e NTree - avaliadas comparativamente utilizando um caso de estudo - SpeedSense, uma aplicação que agrega dados de dispositivos móveis equipados com GPS, permitindo a monitorização e previsão do estado do trânsito num cenário urbano.

Palavras-chave: Sensoriamento participativo, processamento distribuído, data streaming, computação móvel

Abstract

Participatory sensing is a new application paradigm, stemming from both technical and social drives, which is currently gaining momentum as a research domain. It leverages the growing adoption of mobile phones equipped with sensors, such as camera, GPS and accelerometer, enabling users to collect and aggregate data, covering a wide area without incurring in the costs associated with a large-scale sensor network.

Related research in participatory sensing usually proposes an architecture based on a centralized back-end. Centralized solutions raise a set of issues. On one side, there is the implications of having a centralized repository hosting privacy sensitive information. On the other side, this centralized model has financial costs that can discourage grassroots initiatives.

This dissertation focuses on the data management aspects of a decentralized infrastructure for the support of participatory sensing applications, leveraging the body of work on participatory sensing and related areas, such as wireless and internet-wide sensor networks, peer-to-peer data management and stream processing. It proposes a framework covering a common set of data management requirements - from data acquisition, to processing, storage and querying - with the goal of lowering the barrier for the development and deployment of applications.

Alternative architectural approaches - RTree, QTree and NTree - are proposed and evaluated experimentally in the context of a case-study application - SpeedSense - supporting the monitoring and prediction of traffic conditions, through the collection of speed and location samples in an urban setting, using GPS equipped mobile phones.

Keywords: Participatory sensing, decentralized processing, data streaming, mobile computing.

Contents

List of Figures

1. Introduction

As the Web 2.0 paradigm emerged, with flagship services such as Wikipedia, YouTube and blogging, collaboration between global communities of Internet users started taking shape. The essential outcome was the dilution of the previously well established frontiers between content publishers and consumers. User generated content became the norm instead of the exception, as a reflection of a general need for communities to create their own discourse and meaning, instead of relying on the "ivory tower" of rooted institutions.

More recently, the principles that shaped Web 2.0 started spreading beyond culture to citizenship and government. The new discourse on data transparency, participation and collaboration on the public affairs, supported by networked services and applications, demonstrate the willingness for citizens to have an active role in shaping their lives and their community.

An essential outcome of these movements is the collaborative creation of new data sources that can be analyzed, mixed and visualized, creating new interpretations and new services. *Participatory sensing* taps into this fertile ground and adds a mix of new technologies, allowing rich information to be extracted from the physical world. Multiple sensors are now commonly present in consumer electronics devices, thanks to miniaturization and commoditization. Mobile phones, in particular, have evolved into a generic computing platform and can now integrate a growing array of sensors, from camera, sound recorder and GPS, to accelerometer and proximity sensors. Together with their data communication capabilities, this makes these devices specially convenient to create advanced mobile sensing networks. Sensor readings can be combined and processed to create rich context information, such as user activity. Data from multiple sources can be aggregated, leveraging the massive adoption of mobile phones, to create a detailed view of physical reality over a wide area. Given its opportunistic nature, leveraging users inherent mobility and daily activities, this detailed view is possible without the prohibitive costs associated with the explicit deployment of a dense, wide area, sensing infrastructure. Emergent subjects, such as open-source hardware, personal fabrication and the DIY[1] culture expand the range of possibilities (even beyond the atmosphere, given the successful DIY explorations with weather balloons at 30km altitude [25, 49]). The expected outcome from this new paradigm is the creation of rich data sets, a data commons, supporting new services, discourses and interpretations.

The following sections will give further context on the subject; Section 1.1 discusses the conditions for a successful implementation of a data commons stemming from *participatory sensing* applications. Section 1.2, presents some deployed services and applications that illustrate the principles described in this introduction.

[1] Do-It-Yourself

1.1 Data Commons

In essence, *participatory sensing* potentiates the construction of a new data commons, built from the decentralized collection of data by citizens participating in urban sensing networks. It facilitates open grassroots collaboration for the construction of meaning through data visualization, analysis and interrelation (mashups), crossing domains such as science, politics, citizenship and art. This data commons can be seen as a source for new public discourses over the previously invisible and for new forms of engagement in the public domain [7].

The successful establishment of this data commons requires a well designed ecosystem spanning technical, legal and social spheres. As the application domains shift from the scientific to the public realm and, consequently, the control model shifts form centralised to distributed, the new urban and participatory scenario for sensor networks raises new questions regarding the reliability of collected data and interpretation, as well as its social impact. Answers to data reliability can be endemic to the system, by using redundancy and agreement to filter out spurious data. Interpretation reliability could stem from an open "peer review" model, by providing discussion mechanisms and encouraging critical practices. A proper Intellectual Property rights framework should be put in place to spur innovation (similar in spirit to the Creative Commons approach for creative content)

Central to the successful growth of a data commons is the lowering of entry barriers that can alienate participation and innovation. Privacy concerns should be mitigated by the application of appropriate computer security, anonymization techniques and control over data granularity. An open infrastructure should facilitate the discovery, aggregation and republication of data sources to create new integrated analysis and visualizations. Finally, supporting frameworks and development models should reduce barriers represented by development efforts and infrastructural costs, while fluid user interfaces should facilitate active participation on data collection.

1.2 Applications

The purpose of this section is to briefly showcase some real-world examples illustrating principles of (or related to) *participatory sensing*. Applications were chosen from academia, industry or community efforts, covering a wide spectrum of subjects, with the sole requirements that they represent services already deployed or being rolled-out.

1.2.1 Transportation

Launched in November 2008, Mobile Millennium [37] is a partnership between government, industry and academia, involving UC Berkeley College of Engineering, Nokia and NAVTEQ. The service provides traffic information through a mobile phone interface, overlaid on a map, covering the San Francisco Bay area. Data is obtained from mixed sources including road sensors, but essentially from anonymized position and speed readings, obtained from the phones

of the Mobile Millennium user base.

1.2.2 Mapping

The OpenStreetMap project [43] provides free geographic data, such as street maps and points of interest. The motivation underlaying the project stems from the legal and technical restrictions subjacent to the available commercial services. Map data is gathered by the OpenStreetMap community, using simple equipment, such as GPS devices or GPS equipped mobile phones. Other sources of data include public domain aerial or satellite imagery, or even simple hand drawn maps [9]. On the aftermath of the Haiti earthquake of January 2010, a collaborative effort of hundreds of OpenStreeMap users resulted in the production, in 48 hours, of a detailed map of the affected areas that was used by search and rescue team on the ground [1, 42].

1.2.3 Environment

Four UK universities, led by the Imperial College London, launched recently a trial throughout UK for urban pollution monitoring using mobile and static sensors [2]. Small "electrochemical cells" can be carried by pedestrians and use their mobile phone for data upload. Heavier devices are designed to be deployed in cars or buses and also use the mobile phone network. The goal of the project is to understand how traffic generated pollution behaves in the urban environment by generating high-resolution spatial-temporal pollution maps.

PEIR [41]- Personalized Estimates of Environmental Exposure and Impact - from the CENS group, UCLA, is an extension of the "footprint calculator" concept that uses GPS traces acquired from the end-user's mobile phone and matches this information with several data sources, including weather conditions and estimated traffic patterns. PEIR produces estimates of personal impact and exposure in four categories, including smog exposure and carbon footprint impact.

The Gulf Oil Mapping project is an effort by the Grassroots Mapping global community [36] (partially funded by the Legatum Center and the Center for Future Civic Media at MIT). The project involves the use of helium balloons, kites and other inexpensive technology, such as common digital cameras, to produce aerial imagery covering the oil spill in the Gulf of Mexico. The produced imagery will be released to the public domain for environmental and legal use.

1.2.4 Citizenship and Crisis Response

Ushahidi [27] is a platform for incident mapping that aggregates reports published by community users on the ground. It has been used in several crisis scenarios, from the post-election violence that occurred in Kenya in 2007 to the Haiti earthquake in January 2010. Cuidemos El Voto [53] is a community service, based on the Ushahidi platform, that allows citizens to report irregularities on the electoral process, and produce real-time visualization of reports on a map. Cuidemos El Voto was used to monitor irregularities during the electoral process for the federal elections of July 2009 in Mexico.

FixMyStreet [14] and SeeClickFix [47] are community services that allow users to report issues on public spaces with the intention to raise awareness, discuss and to prompt corrective measures. With FixMyStreet, based in the UK, users can report problems using a web interface or an iPhone (using the embedded camera and GPS), that are then forwarded by email to the local councils. SeeClixFix, based in the US, has a similar spirit, but acts as a broker between citizens, government and interested organizations. Users can report problems on a particular location, officials or organizations can create a watch area and be notified when problems are reported within that area.

1.2.5 Entertainment

CitySense [6], by Sense Networks, allows mobile phone users to locate night-life hot-spots in the San Francisco area. Years of historical localization data have been collected and analyzed to create a baseline of normal city activity. Using this baseline, real-time location data is used to infer and display unusually high activity areas, allowing users to find out where everyone is going. Besides, CitySense collects GPS data to infer behavior patterns, classifying users into "tribes" to produce (in a future version) a personalized night-life activity map.

1.3 Problem Definition and Goals

Although significant case-studies have already been conducted regarding participatory sensing, the solutions exposed are typically tailored to their specific application domains or based on assumptions, regarding the support infrastructure, that may prove hard to materialize. Proposed infrastructures usually center around one of two extremes. On one side lies a centralized approach, where a central back-end services the mobile node base. On the other extreme, a peer-to-peer infrastructure is composed exclusively of mobile nodes that cooperate and share resources.

Both extremes have their motivations and advantages. However, they also have relevant shortcomings that limit their applicability. A centralized infrastructure has the advantage of being simpler in terms of development models, implementation of rich features, system maintenance and security enforcement. On the downside, three aspects are particularly relevant for participatory sensing applications. The collection and storage of personal, privacy sensitive, information under the administration of a central entity has relevant privacy implications. The second aspect refers to the creation of data silos where a central entity, and not the end-user base, owns the data and determines the possible usages. Finally, a centralized infrastructure supporting a large scale deployment requires a financial backing, thus hindering an important class of applications, emerging from community needs, where infrastructural and financial resources are not readily available.

Mobile peer-to-peer infrastructures, such as those proposed in Mobile Chedar [30] and the Urbanets [45] concept, have the advantage of being particularly well suited for establishing

spontaneous ad-hoc networks, without requiring a central back-end. However, the inherent resource limitations of mobile devices, and the management overhead introduced by a peer-to-peer system, limits the applicability of these systems.

4Sensing explores an hybrid approach with the goal of overcoming the problems identified above. A mixed infrastructure, where mobile nodes are connected to fixed computers structured in a peer-to-peer overlay network has strong advantages. It can be composed exclusively of personal computing resources, thus supporting grassroots initiatives; resource-heavy operations can be delegated to the fixed infrastructure, thus overcoming the limitations of mobile devices, and data is decentralized and not owned by a single entity.

Developing applications in the context of this hybrid architecture poses specific challenges regarding all aspects of data management - from acquisition, to processing and persistence. 4Sensing addresses these challenges, with the goal of providing a framework and development model that supports, and simplifies, the development of a wide range of participatory sensing applications. With this purpose, a set of data management requirements have been identified:

Data acquisition Central to a data management system is the ability for applications to express their data needs - in terms of required sensors, tasking modalities, sampling rates, and delivery modes - abstracting from specific sensor platforms and hardware. Besides, applications should be able to integrate domain specific data processing tasks, such as filtering and event detection.

Data processing and aggregation A data management framework has to support a wide range of data processing operations, where data processing refers to any transformation activities, such as extraction of features, filtering, computation of higher level inferences and aggregation. Data aggregation has an important role in data reliability and accuracy, mitigating the effect of spurious data. It also supports summarization of storage intensive raw data and the inference of relevant statistical trends.

Querying A querying model must provide a unified data view, abstracting the complexity of a distributed system from the querying model. The data management system must transparently route queries to the relevant data sources and deliver results without exposing the network structure to applications.

Persistence Data persistence is an important tool that supports the access to historical information, the creation of models for higher order inferences and the analysis of trends based on past data. Challenges related to persistence include the balanced sharing of storage resources, dealing with storage capacity limits, and the ability to find data efficiently.

Data sharing A support framework for the sharing of data enables the *data commons* concept. Sharing must address the balance between flexibility and control over the granularity of published data in order to establish a trusted ecosystem involving end-users and applications developers.

1.4 Main Contributions

This dissertation proposes a distributed data management model for the generic support of *participatory sensing* applications, with the following contributions:

- A generic framework supporting the data life-cycle of applications based on geo-referenced data;
- A flexible decentralized model for data processing leveraging the advantages of a hybrid fixed/mobile infrastructure;
- The development and comparative evaluation of three alternative models for data distribution in an hybrid infrastructure - RTree, QTree and NTree.

1.5 Document Structure

This document is structured around six chapters, whose content is described below:

Chapter 1 - Introduction Presents the context of the participatory sensing domain, the motivations and underlying goals of this work and its main contributions;

Chapter 2 - Models Describes the proposed models, focusing on the application architecture, the data modeling aspects, data processing and distribution. Finally, the three distribution strategies - RTree, QTree, NTree are discussed in detail;

Chapter 3 - Prototype Presents the 4Sensing prototype, describing the main architectural components;

Chapter 4 - Case Study and Experimental Evaluation Presents SpeedSense - the case-study application implemented in a simulation environment - and the experimental evaluation performed over this simulation;

Chapter 5 - Related Work Presents research work related to the theme of this dissertation. The chapter starts by presenting data management issues and approaches to those issues found in the literature; the second part discusses selected works in more detail;

Chapter 6 - Conclusions and Future Work Presents the main conclusions, reviewing the goals of the dissertation and discussing relevant directions for future research.

2. Models

4Sensing applications use sensor data gathered by a community of users, profiting from their inherent mobility, creating a global view that incorporates the individual contributions of each user into a meaningful service. From a functional perspective, it can be broken down into data *acquisition*, *aggregation*, and *usage*. Data acquisition refers to the process through which sensor data is gathered, and aggregation generally refers to the combined processing of data acquired by individual users, subordinated to the particular application requirements. Usage comprises the modes of display and interaction over data, through visualization (e.g., the production of visual maps over a geographic region), services (e.g., planning efficient driving routes taking into account the expected traffic conditions) or publishing - allowing external applications to combine data produced by the application in new ways (application *mashups*).

Applications are community driven, in the sense that they are created, managed and used by a community of users that are actively engaged and have an expressed interest in the services provided. Although several applications share a common framework, and possibly, resources and data, each application is self-contained, and has a well-defined base of participants. Users explicitly opt-in to engage in particular applications for which they are willing to contribute sensor data and computing resources. This *participatory* model contrasts with an *opportunistic* approach (see [31]), where users participate as members of a sensing infrastructure and have limited awareness and control over which applications they are actively engaged with.

2.0.1 Architecture

Applications are supported by a distributed system consisting of a high number of mobile nodes equipped with sensors, and a fixed support infrastructure. A mobile node can be any mobile computing platform, typically a mobile phone, and is expected to be a limited environment in terms of computing power and battery life. Mobile nodes are connected to a fixed infrastructure, composed by a set of nodes structured in an overlay network, that supports the more resource intensive operations, such as data processing, routing and storage. Fixed nodes can be personal computers, virtual machines running in a utility computing infrastructure, or servers hosted by independent entities. Applications are hosted in this hybrid environment, supported by a service middleware running in both mobile and fixed nodes.

2.0.1.1 Core Services

Core services refers to the set of services supporting the domain applications, providing a data management framework and abstracting the distributed nature of the underlying system. Services are supported through both a fixed middleware layer and a mobile counterpart hosted on mobile nodes, and are exposed to applications through well defined interfaces.

On the fixed infrastructure, core services implement a set of constructs that allow applications to define their data acquisition, processing and querying requirements, while on the mobile

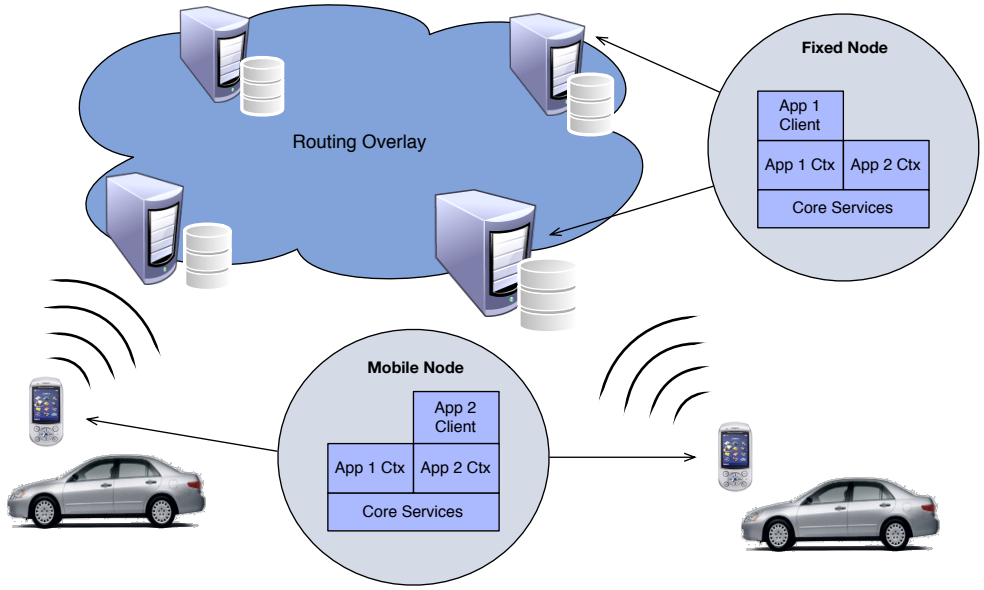


Figure 2.1: High-level Architecture

side, core services support data acquisition. A querying interface is exposed in both the mobile and fixed counterparts.

2.0.1.2 Applications

Applications can be broken down into client and service components. A client implements a user interface, supporting the interaction between user and service. It can be hosted on a mobile or desktop environment and interacts with the service component through a query interface. Application services define the application logic regarding the complete data lifecycle - how data is acquired, processed, queried and stored - using the constructs provided by the core services. Services are supported by *application contexts* deployed on mobile and fixed components.

2.0.2 Example Applications

This section presents three example *participatory sensing* applications that will be used to illustrate various aspects of the proposed model.

2.0.2.1 SpeedSense

SpeedSense - the case-study used in this dissertation - assists drivers in urban areas, such as the Lisbon metropolitan region. It receives GPS data collected by mobile phones, carried by the community of users while driving as part of their daily routines. Based on this information, SpeedSense can infer current and predicted traffic conditions, and provide a set of services including up-to-date average driving speeds, route planning, and travel time estimation.

2.0.2.2 NoiseMap

NoiseMap, is a mashup application that builds a map of estimated noise levels across a city, using data published by SpeedSense. Noise levels are based on car density and average speed, being classified as low, average and high. The noise map is obtained by dividing space using a regular grid, obtaining the car density and speed for each grid cell and computing a scalar speed level from these values. The noise level classification is obtained from this scalar value according to predefined thresholds.

2.0.2.3 PotholePatrol

The PotholePatrol example is inspired by the *Pothole Patrol* system [11]. The application uses data gathered from accelerometer and GPS sensors mounted on motor vehicles to detect road hazards such as potholes. Given the high sampling rates required by the application, a pothole detection algorithm runs on the mobile nodes, providing potential pothole locations. Detections gathered from several nodes are clustered in order to filter out spurious detections that are not corroborated by a certain number of nodes.

2.1 Data Model

2.1.1 Data Representation

Applications collect *raw* data from sensors embedded in mobile devices that can be encoded in a wide range of formats, such as scalar values, image, video, and sound recording. Aggregate values, inferences and event detections are examples of *derived* data that applications represent and manipulate. In an abstract form, data can be modeled as a tuple with a particular structure or *type*, defined as a set of named attributes. For instance, a GPS reading comprises latitude, longitude, elevation, bearing, speed and accuracy attributes. Additionally, sensor reading reflects a particular moment in time and a location in space, contextual characteristic that are conveyed as metadata - a timestamp records the time when the sensor reading was sampled, while a geographical coordinate references its location in space. Derived data also has an explicit spatial nature, but it does not necessarily have a temporal nature - e.g, a pothole detection is based on several individual detections obtained in different moments in time.

2.1.2 Streams

4Sensing applications manage a continuous flow of data resulting from sensing and data processing activities. Generally, a stream is a sequence of data tuples with a common structure and semantic. Regarding its origin, a *data acquisition stream* is a tuple sequence produced by mobile nodes according to a *sensor definition* (explained in Section 2.1.3). SpeedSense bases its inferences on a GPS data acquisition stream - the sequence of GPS readings acquired by mobile nodes. The stream of average speeds over a time period is a *derived stream* obtained by averaging GPS readings.

Orthogonal to this definition, a *live stream* is an unbounded tuple sequence, created as data is acquired and delivered to the system, while a *stored stream* is a finite tuple sequence representing the past of a live stream up to the present moment in time. The stored stream for GPS readings, in a given moment in time, contains all the GPS reading delivered before and up to that moment. Similarly, in PotholePatrol, a pothole detection live stream could be addressed, although it does not have a strong relevance for the application; PotholePatrol relies on the stored detection stream to implement its clustering algorithm.

2.1.3 Data Acquisition

In order to cater for a broad spectrum of application domains, an acquisition framework must be flexible and allow applications to specify suitable acquisition modalities. For this effect, an acquisition model defines a set of dimensions, described in the following sections, that together characterize how data is fetched and, possibly, persisted. Applications express these modalities through *sensor definitions*. Table 2.1 presents example definitions for the SpeedSense and PotholePatrol applications.

Application	QoS	Sampling	Tasking	Longevity
SpeedSense (GPSReading)	online	periodic	query-driven	volatile
PotholePatrol (PotholeDetector)	deferred	event-based	application-driven	persistent

Table 2.1: Example sensor definitions

2.1.3.1 Sensor

Identifies the acquisition source, by naming a particular sensor. Sensors are modeled as software components running on the mobile devices that act as adaptors, abstracting hardware dependencies. A base set of sensors can be assumed to be part of a standard acquisition framework, such as GPS, camera, etc., but applications can define their own, interfacing directly with hardware sensors, or integrating other adaptors in order to create virtual sensors - such as a positioning sensor leveraging GPS, GSM and Wi-Fi information. What is assumed in the proposed model is that applications can name particular sensors, either standard or defined by applications.

2.1.3.2 Quality of Service

Relative to acquisition latency - i.e., the time a sample takes to become available in the fixed infrastructure - applications can operate in two modes. In *online* data acquisition, latency is low, so that data is ready for usage shortly after acquisition. This is a requirement for applications that monitor the evolution of a phenomena as it happens, as is the case in SpeedSense and NoiseMap. Typically, this approach involves short-lived data and long standing (*continuous*) queries, that result in a data stream reflecting an evolving reality.

Other applications may be able to operate under more relaxed delivery requirements, where continuous connectivity is not always available, or is too costly. In this case, applications can operate according to a *deferred* data acquisition model. Deferred acquisition is associated with opportunistic or intermittent connectivity, or disconnected operation models, where there is an unbounded lag between acquisition and availability. These applications normally focus on phenomena that are less dynamic, or use data to infer trends instead of a recent state.

2.1.3.3 Sampling Modality

Sampling can be characterized as *periodic* or *event driven*. In the former case, acquisition is parameterized by a sampling rate, while in the latter, a virtual sensor is responsible for identifying relevant occurrences - as is the case of the pothole detector in PotholePatrol. Events can also be explicitly produced by users, acting as virtual sensors, as in Microblog [18] where users proactively submit georeferenced multimedia content.

2.1.3.4 Sensor Tasking

Sensor tasking refers to the process through which applications control mobile sensor activity. Applications may require data from a subset of the available mobile nodes, depending on which queries they are currently running. This *query-driven* acquisition is relevant for online traffic monitoring in SpeedSense, where it is more cost-effective to acquire speed samples only in areas that are actually being monitored. In PotholePatrol, acquisition is *application-driven* - data is considered relevant independently of the running queries. This is usually the case in applications that operate in a deferred mode - since there is a lag between acquisition and querying, it is not possible to determine a priori which sensors are relevant. But this is not necessarily the case - in BikeNet [10] users can issue queries to task remote sensors, while responses are delivered when possible, according to an opportunistic connectivity model.

2.1.3.5 Longevity

A sensor definition also characterizes the longevity of the acquired data. Under an application-driven model, given a lag between acquisition and consumption it is necessary to guarantee that data will be available when queries are issued - in this case data must be *persistent*. In other cases, data is short-lived, or *volatile* and has no value beyond immediate consumption.

2.1.4 Virtual Tables

Similarly to the relational paradigm, where a data set with a common schema is represented by a relation, or table - in 4Sensing, applications model data according to *virtual tables*. A virtual table specifies a *derived* stream in terms of one, or more, input streams - either a set of data acquisition streams or another virtual table - and a stream transformation. The term virtual is used here because it names a construct that has a dual nature, depending on how data is queried, providing a common abstraction over current and past data. A *continuous* query over a virtual table will produce a live stream that reflects the most recent acquired data, while an *historical* query results in replaying past data - the stored stream. Continuous and historical queries are discussed further in Section 2.1.5.

Virtual tables can specify a stream by direct derivation from one or more sensor definitions. To provide the current average speed per street segment, SpeedSense defines the *TrafficSpeed* table, as presented in Listing 2.1. This table specifies the stream of the current average speed and car density, per street segment - a sequence of tuples with the structure (*segmentId*, *avgSpeed*, *density*) - obtained by averaging the data obtained from the *GPSReding* acquisition feed. Note that temporal and spatial metadata is not explicitly represented. Alternatively, a stream can be derived indirectly through another table; to implement route planning, SpeedSense needs to know only about the congested streets - i.e., streets where speed is below a given threshold - assuming an average speed where a congestion has not been detected. The *TrafficHotspots* table serves this purpose. It specifies a stream, derived from *TrafficSpeed*, where each tuple represents a traffic detection event. Stream derivation from sensor definition and from a virtual table is represented respectively by the *sensorInput* and *tableInput* declarations in Listing 2.1. For simplicity, only single table derivation is considered although the model can be extended for multiple-table derivation.

Listing 2.1 Examples of virtual tables and respective input definitions

```
TrafficSpeed(segmentId, avgSpeed, density)
    sensorInput(GPSReading)
TrafficHotpots(segmentId, avgSpeed, confidence)
    tableInput(TrafficSpeed)
TrafficTrends(segmentId, avgSpeed, confidence)
    tableInput(TrafficHotspots, stored)
PotholeDetection(clusterId, confidence)
    sensorInput(PotholeDetector, stored)
NoiseLevel(cellId, noiseLevel)
    tableInput(TrafficSpeed)
NoiseTrends(cellId, noiseLevel)
    tableInput(NoiseLevel, stored)
```

SpeedSense also supports planing for routes that will take place in the future, based on predicted traffic speeds - for instance, a user could request the best route to the airport on a monday morning. For this effect, SpeedSense can use past data to predict future conditions. Past hotspot detections, which convey average speeds per street segment, can be retrieved from the *TrafficHostpots* stored stream. *TrafficTrends* derives from this stream using the *stored* modifier. The table does not have a continuous semantic since it represents historical data.

In PotholeDetection, there is little interest in the online detection of street anomalies. To be considered potholes, detections have to be corroborated by several vehicles, which typically occur in several moments over time. The *PotholeDetections* table specifies a stored stream - it represents confirmed detections, given the historic of individual detections. Again, in this case there is no continuous semantic.

NoiseMap can display both an up-to-date map of noise levels, or a trend map based on past inferences. The *NoiseLevel* table derives from *TrafficSpeed*, where tuples represent average noise levels per cell based on live speed and density data. *NoiseTrends* specifies a stored stream based on the *NoiseLevel* history.

In the context of a particular virtual table, a *base* stream refers to the original source of data tuples - either a live acquisition stream of a stored stream - *TrafficSpeed* and *TrafficHotspots* have *GPSReading* as their base stream, while *TrafficTrends* has the *TrafficHotspots* stored stream as its base.

2.1.5 Queries

Applications access data by issuing queries, which express constraints over virtual tables. A query generates a tuple stream produced by the target virtual table, resulting from the application of the query restrictions over its base stream.

Queries can be either *continuous* or *historical* in terms of how they access virtual tables. *Continuous* access characterizes the usage of data as soon as it is delivered - through a live stream. Typical usages involve online monitoring and event detection over a phenomena that is changing in time. On the other side, *historical* queries access stored streams and are suitable for the inference of trends and patterns from data related to the past.

Continuous queries specify a spatial constraint - an *area of interest*, such as rectangular bounding box. The result of this type of query is the tuple stream produced by the target virtual table, where each tuple reflects a particular moment in time and has an associated timestamp. Successive tuples may reflect changes in the monitored phenomena occurring during query evaluation.

Additionally, an historical query can define temporal constraints, expressed as predicates over the timestamps of stored tuples. A council office might be interest in using the PotholePatrol application to gather road anomalies detected in the local area. A query specifying a bounding box covering the council region would produce pothole detections inferred from all available data. Using a temporal restriction, the user can filter out detections based on data with more that one month. In SpeedSense, in order to query the average speed on a monday at 9am, only

the tuples stored in *TrafficHotspots* with timestamps in the specified range are considered. The query model assumes that every data tuple has associated spatial metadata, so that query restrictions can be applied. For the assumption to hold, the stream transformations defined by a virtual table must guarantee that the resulting tuples have spatial metadata - an issue that will be addressed in Section 2.2.2.

2.2 Pipeline Model

As described in Section 2.1.4, an application data model is defined through virtual tables, in terms of input streams and stream transformations. A stream transformation is the outcome of applying a set of operators over an input stream, resulting in a derived stream. This transformation can be modeled as a sequence of operators (a *pipeline*), each performing a specialized function. When a query is issued, the target table input streams are instantiated (possible involving the instantiation of source tables) and fed into the associated pipeline, where they are processed in sequence by each operator - transforming, aggregating, filtering and classifying data - the output of the pipeline becoming the result of the query.

2.2.1 Stream Operators

A stream operator performs a particular function, such as data partitioning, aggregation, filtering, computation of higher order inferences or event detection. Each operator receives a stream of data tuples, producing a derived stream that becomes the input of the next operator. Operators are parametrized by one or more input and output types - that specify the expected attributes. They can operate over each tuple individually or over finite tuple sequences, modeled as a stream terminated by an *end-of-stream* indicator. Finite sequences originate from stored streams or from the decomposition of a live stream through application of the *timeWindow* and *set* operators. Operators can be stateless or store state information that is maintained during query evaluation. Output can be immediately produced as the result of input transformation - this is usually the case for stateless operators - or deferred until a condition. A filtering operator can output data only when a particular attribute has changed significantly relatively to previous tuples, and an aggregation operator will produce an aggregate tuple that results from processing a tuple set. The specific operators are described in the following sections, Listing 2.2 shows their pseudo-code representation that will be used later when pipeline listings are presented.

2.2.1.1 Processor

The *processor* operator is the main extension basis for implementation of application specific processing, such as data *mapping* and unit conversion. Another example would be an interpolation operator that, given GPS and accelerometer readings, produces a single tuple using linear interpolation.

Filter and *Classifier* are specialized processors. A *filter* outputs the received input, only if

Listing 2.2 Pseudo-code representation for stream operators

```
process<inputT1, outputT1,...,inputTn, outputTn>
    (processingFunction1,...,processingFunctionn)
filter<inputT>(filterCondition)
classify<inputT, outputT>(classificationCondition, classifierFunction)
groupBy<inputT>(partitioningCondition, subPipeline)
timeWindow<inputT>(size: <size>, slide: <slide>)
set<inputT>(mode: change | eos, partitioningCondition)
aggregate<inputT, outputT>(
    min|max|count|sum|avg(attribute1, aggAttribute1)
    ...
    min|max|count|sum|avg(attributen, aggAttributen)
)
```

a given condition is satisfied. In particular, input can be forwarded only if a given attribute has changes above a given threshold in relation to the last output. A *classifier* is used to generate inferences from aggregated data. It applies an application specific condition to an input tuple and forwards a classification tuple if this condition is satisfied. Classification and filtering is further discussed in 2.2.3

2.2.1.2 GroupBy

The *groupBy* operator partitions data according to a set of tuple attributes, or any partitioning condition, producing a set of independent data streams. This construct can be used to maintain independent state data for the partitioned streams, as in grouped aggregation, and can be explored to support parallel processing of partitioned streams. Besides the partitioning condition, *groupBy* defines a sequence of operators to process each of the independent streams modeled as a sub-pipeline. For simplicity this sub-pipeline is assumed to be a linear sequence of stream transformations not including itself a *groupBy* operator.

A *groupBy* component is particularly relevant for the computation of group aggregates, e.g., to produce an average speed per street segment. It operates by dispatching each input tuple to the appropriate sub-pipeline, according to the partitioning condition. The resulting stream is the combined output of each of the sub-pipelines into a unique stream. When applied over a finite tuple sequence, *groupBy* results in the transformed sequence, i.e. the output of each partitioned stream will be followed by *end-of-stream*.

2.2.1.3 TimeWindow

Continuous query can be used to compute an aggregate value over the whole execution time - for instance, to compute the average speed over a 30 minute period - but more frequently, applications are interested in receiving updates on the current state of the monitored phenomena - such as the current average speed. A *timeWindow* can be used for this effect, by decomposing a input stream into a sequence of time periods.

A *timeWindow* is defined in terms of size and slide parameters. The window size defines the length, in seconds, of the window extent, while the slide parameter defines the temporal relation between successive extents. A window with a size of 15 seconds and slide of 10 seconds will output a tuple set received in the last 15 seconds, every 10 seconds. The window assumes that data is received with limited jitter i.e. the difference in arrival time between two tuples is approximately the same as the difference between their timestamps.

2.2.1.4 Set

Set operates by placing input tuples into buckets according to a partitioning function, or attribute, each bucket storing the most recent tuple received. For instance, using a segment identifier as the partitioning attribute for a *set* operator results in the set of most recent tuples for distinct segments How the tuple set is forwarded depends on its parametrization. When operating in *change* mode - usually as part of a continuous pipeline - the tuple set is forwarded whenever it changes, i.e., when a new input is received (in this case, *set* has the semantic to a *partitioned window* with one row in CQL [3]). In *eos* mode - as part of an historical pipeline or preceded by a *timeWindow* - a set is forwarded only when a *end-of-stream* indicator is received.

2.2.1.5 Aggregator

An *aggregator* operates over a finite tuple sequence. Its output results from the application of aggregation operations over one or more input attributes of the input sequence. Aggregation operations - maximum, minimum, count, sum and average - specify an input attribute and the resulting aggregate attribute.

2.2.2 Pipeline Structure

Aggregation usually involves an initial *groupBy* partitioning operation, so that related data can be meaningfully aggregated together. A previous operation may be necessary to classify data into related groups - such as street segments or cells in a regular grid - an operation referred as *mapping*. Mapping associates data with the spatial extent of the partition and possibly a partition identifier used as the *groupBy* partitioning condition.

Listing 2.3 shows examples of speed averaging in SpeedSense, for continuous and historical cases. In *TrafficSpeed*, the pipeline is sourced with a live stream of GPS samples and the process operator maps a (latitude, longitude) pair to a particular street segment. This live stream is

broken down into window extents by the *timeWindow* operator, *groupBy* partitions the stream based on the segment identifier, and *aggregate* averages all tuples in each sub-stream, producing averages every 10 seconds - Figure 2.2 shows a graphical representation of the *TrafficSpeed* pipeline. *TrafficTrends* is sourced with the *TrafficSpeed* stored stream, restricted by the temporal restrictions stated on the query. The rest of the pipeline is equivalent to the continuous case except for not using a window, resulting in an *AggregateSpeed* tuple for each segment followed by *end-of-stream*. The second historical case, *TraficTrendsHour*, shows how mapping and partitioning can be used in the time domain to compute average segment speeds per hour of the day.

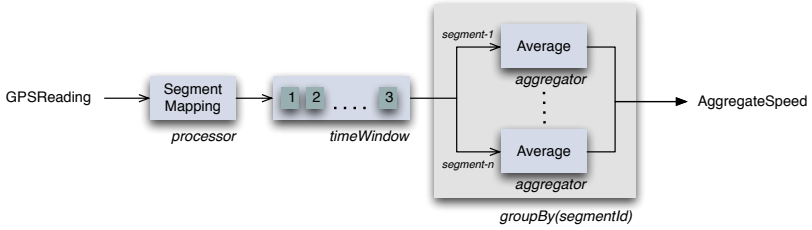


Figure 2.2: TrafficSpeed pipeline

As referred in Section 2.1.5, stream transformations have to guarantee that resulting tuples have associated spatial metadata. For this to hold true, the aggregation operator has to assign a spatial extent to its output. The spatial extent is already present in each individual tuple, given by the mapping operation, the same extent is then assigned to the aggregate value. But mapping on the spatial domain is not mandatory. Consider, for instance, that NosieMap supports querying for an overall noise level in a particular area, instead of a grid. In this case, it would average speed and car density for all street segments bounded by the query area. Since there is no stream partitioning based on spatial extent, the aggregation operation will receive tuple sets with different extents. In this case, the spatial extent of the aggregated values is the area of query, which is consistent with the query semantic. In a continuous query, tuples also have a temporal dimension. In this case, the aggregator assigns the maximum timestamp on the window extent to its output.

2.2.3 Classification and Filtering

The classification operation is applied after aggregation in order to generate higher order inferences. As an example, TrafficHotspots uses a congestion detector as a classifier. It receives a tuples representing average speed per segment, and uses the number of averaged readings as the basis of the classification condition. It produces a congestion detection tuple by adding a state attribute and an associated degree of confidence. Filtering can be used to restrict a result stream, or to filter irrelevant transitions in a continuous stream. Both classification and filtering are illustrated in Listing 2.4.

Listing 2.3 SpeedSense aggregation virtual tables

```
TrafficSpeed(segmentId, avgSpeed, density) (
    sensorInput(GPSReading)
    process<GPSReading in, MappedGPSReading out>(
        out.segmentId = segmentId(in.lat, in.lon) out.extent = segmentExtent(out.segmentId))
    timeWindow(size: 15s, slide: 10s)
    groupBy(segmentId) (
        aggregate<MappedGPSReading, AggregateSpeed>(
            avg(speed, avgSpeed)
) ) )

TrafficTrends(segmentId, avgSpeed) (
    tableInput(TrafficSpeed, stored)
    groupBy(segmentId) (
        aggregate<AggregateSpeed, AggregateSpeed>(
            avg(avgSpeed, avgSpeed)
) ) )

TrafficTrendsHour(segmentId, hour, avgSpeed) (
    tableInput(TrafficSpeed, stored)
    process<AggregateSpeed in, AggregateSpeedHour out>(
        out.hour = hour(in.timestamp))
    groupBy(segmentId, hour) (
        aggregate<AggregateSpeedHour, AggregateSpeedHour>(
            avg(avgSpeed, avgSpeed)
) ) )
```

Listing 2.4 Classification and filtering in SpeedSense

```
TrafficHotpots(segmentId, avgSpeed, confidence) (
tableInput(TrafficSpeed)
    classify<AggregateSpeed in,Hotspot out>
        (in.count > COUNT_THRESHOLD),
        (out.confidence = min(1, in.count/COUNT_THRESHOLD * 0.5)
        if(in.avgSpeed <= SPEED_THRESHOLD * maxSpeed(in.segmentId)) out.state = RED
        else out.state = GREEN)
    filter<Hotspot in>(
        in.confidence > 0.7 and changePercent(avgSpeed, 10))
)
```

2.3 Distribution Model

The data processing model has been discussed assuming a centralized infrastructure. This section describes a generalization of the presented concepts to a distributed architecture, where the fixed infrastructure involves several interconnected nodes. Nodes are structured in a peer-to-peer model, each performing a similar role - as data acquisition, storage and processing elements. The particular issues that stem from this setting are essentially related to how relevant data can be located, how the pipeline processing model can be effectively distributed, and this complexity can be hidden from application developers. Concrete approaches will be presented later in Section 2.4, while this section sets the grounds for a generic distribution model.

Assuming the set of nodes with relevant data for a specific query is previously known, it is now necessary to extend the pipeline concept so that data from independent nodes can be merged together to produce the query results. Conceptually, pipeline processing can be broken down into two stages, or roles - *data sourcing* and *global aggregation*. Data sourcing refers to the process through which each node produces partial state tuples from the data it knows, while global aggregation refers to the production of an aggregate result by merging the partial state from several nodes. The data sourcing stage receives continuous sensor input (or stored data), acquired by (or stored by) the node that is executing the pipeline. Typically, data sourcing involves mapping, partitioning and local aggregation, as discussed in 2.2.2, where local aggregation is used to summarize data sourced by a particular node. On the global aggregation stage, pipelines receive the partial data from data sources producing overall aggregated results. Listing 2.5 presents the decomposition of the TrafficSpeed originally presented in 2.3. The data source stage applies the same operators as before, the only difference being that aggregation now produces partial state - to produce an average, a partial state is composed by sum and a count values. Global aggregation applies the *set* operator before aggregation, using a node identifier (part of the tuple metadata) as the partitioning condition - this will produce the set of

last *AggregateSpeed* tuples received from each node. The average operation uses an extended syntax to represent averaging from a partial state (sum and count values). Figure 2.3 illustrates the *TrafficSpeed* data source and global aggregation stages.

Listing 2.5 Distributed pipeline

```
TrafficSpeed(segmentId, avgSpeed, density) (
    sensorInput(GPSReading)
    dataSource(
        process<GPSReading in, MappedGPSReading out>(
            out.segmentId = segmentId(in.lat, in.lon) out.extent = segmentExtent(out.segmentId))
        timeWindow(size: 15s, slide: 10s)
        groupBy(segmentId) (
            aggregate<MappedGPSReading, AggregateSpeed>(
                sum(speed, sumSpeed)
                count(count)
    ) ) )

    globalAggregation(
        groupBy(segmentId) (
            set<AggregateSpeed>(peerId, mode: change)
            aggregate<AggregateSpeed, AggregateSpeed>(
                avg(sumSpeed, count, avgSpeed)
) ) ) )
```

Given this decomposition, it is now possible to decouple data sources from aggregation. A possible distribution model, would be to have the querying node performing the global aggregation role, and nodes with relevant data performing the data source role. A problem with this approach would be the excessive burden carried by the querying node. Distributed query processing can be achieved, for instance, through an aggregation tree spanning all relevant nodes, and rooted on the querying node. In this scenario, each tree node performs the global aggregation role, combining the partial state from the lower levels, and forwarding the result downstream if necessary [1]. Forwarding an aggregate value is not necessary when a global aggregator knows that there is no more information available downstream for the aggregate spatial extent, i.e., when the aggregate is *complete* or *bounded*. In this case the tuple can be forwarded to the root without further processing. How a global aggregator knows when an aggregate is complete depends on how data is partitioned among nodes. This issue will be explored further in Section

[1] throughout the document, the direction towards the root at a given point of the aggregation tree is referred as *downstream*, while the opposite direction is referred as *upstream*

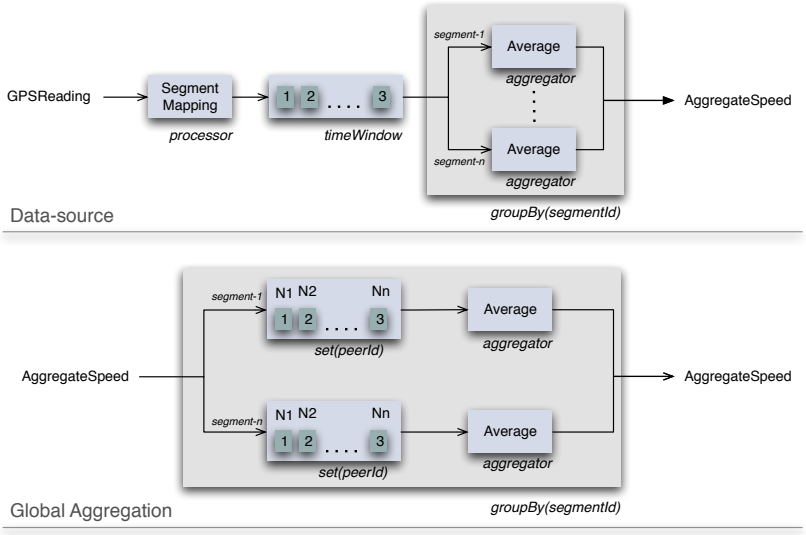


Figure 2.3: TrafficSpeed distributed pipeline

2.4 when distribution strategies are discussed. The classifier operator has to be reconsidered in this distributed setting, a necessary condition for the classifier to produce an inference is that the aggregate it receives is complete, so that classifications are not produced over partial values. If the input is not complete, it has to be forwarded so that aggregation of partial states can continue downstream.

2.3.1 Persistence

In order for applications to access past data using the *stored stream* construct, pipeline outputs resulting from continuous queries have to be stored in a persistent medium. Since pipeline processing is distributed among independent nodes, it is now necessary to define how storage is partitioned between the nodes that participate in a query. The candidates for storage are *bounded* tuples output by a global aggregation pipeline at a particular node. Since no further information regarding the tuple spatial extent is available downstream, the tuple can be stored knowing that it represents a final result. The actual location where a tuple is stored and replicated depends on the adopted distribution strategy.

2.3.2 Continuous Processing

Running a continuous query, over an aggregation virtual table, produces a data stream that conveys the evolution of the monitored variables over time. Querying NoiseMap produces current noise levels, where each tuple represents the average noise level for a particular grid cell, given the most recent acquired data. The set containing the last tuple received for each grid cell,

at a particular moment in time *t*, represents the query result at *t* - where the relation between this moment *t* and real-time, depends essentially on the acquisition latency and additional latency introduced by the aggregation tree. This result set is updated over time as new data is acquired and input into data sourcing pipelines - when new data flows into the aggregation structure, it triggers the recomputation of partial states downstream, possibly resulting in a change to the result set.

Partial state in a global aggregation pipeline refers to the set of data tuples representing the most recent partial aggregates at that particular tree level. State can be maintained by the *timeWindow* and *set* operators. While a *timeWindow* has an implicit state expiration policy given by the window size, this is not the case for the *set* operator. State can be managed according to a *soft-state* or *hard-state* approach, where each option places different requirements on the underlying communication substrate. In a hard-state approach, *set* considers a tuple as valid until a new one is received, or until a failure on the originating node is detected. Since a node can distinguish absence of communication from failure, the upstream nodes can restrain from communicating their values, as long as they do not change - an opportunity to cut on communication and computation costs, depending on the dynamism of the monitored variables. In a soft-state approach, the *set* is parametrized by a *time-to-live*. Soft-state has the advantage of simplicity, since node failure is automatically handled - the values received from a failed node will remain only until the time-to-live expires. But, in this approach, upstream nodes cannot restrain from sending regular updates even if the data has not changed significantly. The two state management approaches also impact the semantic of the query result set. If soft-state is used, tuples in the result set also have an assigned time-to-live and are evicted after expiration.

2.3.3 Historical Processing

In historical processing a pipeline propagates aggregated values only when all data has been received from all descending nodes. This can be achieved by extending the *end-of-stream* semantic so that when a global aggregation pipeline receives an *end-of-stream* from each of the child nodes, it will produce an *end-of-stream* locally - resulting on the propagation of the aggregation tuples upstream. In this setting, only complete aggregates reach the root, but node failure can prevent, or delay, completion of query processing. Alternatively, a *flush* control signal can reduce delay, in face of node failure, at the cost of producing progressive results. This signal can be triggered periodically or when pipeline input is idle for some time.

2.3.4 Virtual Table Derivation

When used without the *stored* modifier, the meaning of the *tableInput* declaration is the application of the stream transformation defined in the derived table to the output of the origin table. In a distributed setting, where stream transformation is broken down into data sourcing and global aggregation, this requires some redefinition - the pipeline defined by a derived table always concerns the global aggregation stage, since data sourcing has already been specified

in the origin. Conceptually, virtual table derivation results in a stream transformation where the data source stage is given by the origin table, and the global aggregation stage results from the concatenation of the origin global aggregation pipeline with the pipeline defined by the derived table. The same concept applies for subsequent table derivations i.e., each derived table represents the concatenation of the pipeline it defines to the origin global aggregation stage.

2.4 Distribution Strategy

As discussed earlier, the 4Sensing architecture involves a high number of mobile nodes and an infrastructure that comprises fixed nodes, owned by a particular user or institution, interconnected over a peer-to-peer overlay. A *distribution strategy* determines the structure of this fixed network and defined the roles (or functions) the elements assume in this architecture and how they interrelate in order to provide acquisition, storage and query processing services. In particular, it defines the role of the *homebase*, how data is acquired by the system, how queries are distributed to relevant nodes, and how data is aggregated. This dissertation proposes and analyses three strategies - RTree, NTree and QTree - described in sections 2.4.5 to 2.4.7.

Each strategy essentially proposes a specific data partitioning approach, specifying which fixed nodes acquire and store which data. Data partitioning is the basis for determining which nodes are relevant to a particular query, and how data can be effectively aggregated. In a *personal store* model, data partition is based on ownership. This is the case in RTree, where a user's *homebase* acquires and stores all data produced by his mobile node. Alternatively, in a *spatial partitioning* approach (as in QTree and NTree), fixed nodes divide physical space into possibly overlapping areas of responsibility .

Common to both approaches is the fact that each fixed node is associated with a physical space coordinate (given by a latitude and longitude) - a geographic address - which reflects its actual physical location. The next three sections will present the main functions involved in a distribution strategy.

2.4.1 Homebase

A *homebase* is the point of access to the 4Sensing architecture for a given user, supporting application clients and mobile connectivity. From the standpoint of the mobile node, it is the primary point of contact with the fixed infrastructure (a well-known node). The exact role of this homebase depends on the actual distribution strategy but, as a common denominator, it will act as a boostrap agent, creating the pre-conditions for running applications by installing the associated contexts on the mobile client. Additionally, the homebase acts as a mobile client delegate, by handling queries on its behalf.

2.4.2 Data Acquisition

Besides their homebase, mobile nodes interact with *acquiring nodes*, responsible for the reception and storage of data acquisition streams. Determining which node plays the acquiring role is a homebase function that depends on the data partitioning strategy. In the *personal store* model, the homebase assumes this role for the associated mobile, while in spatial partitioning, the acquirer is determined in function of the mobile node's physical location. In this case, the data delivery process depends on the particular quality of service model. In online data delivery, the homebase assists the mobile node in finding the acquiring node, a process that has to be repeated regularly due to node mobility. The mobile and fixed acquisition end-points can then interact directly by forwarding the standing queries and delivering data. In deferred mode, the homebase acts as a proxy, initially receiving acquired data and routing it to the appropriate acquisition node. The model is open in terms of how mobile nodes link to the fixed infrastructure. It assumes an abstract network layer whose quality of service characteristics set the bounds of the supported acquisition modalities.

2.4.3 Query Dissemination and Control

Queries are issued by application clients through the homebase and are distributed among the set of relevant nodes (the data sources) using a query dissemination and control function of the overlay network. Generically, nodes support query routing based on the target area of interest and build a dissemination tree rooted on the node that issued the query, reaching all (or a majority of) nodes that hold relevant data. This same tree can be used for query control operations, such as query maintenance and termination.

2.4.4 Query Processing

Following the query dissemination process, a query processing tree is established and specific nodes assume the global aggregation role. A processing tree is more effective for a particular data partitioning if it leverages this partitioning in order to reduce computation and communication. This reduction is possible if a node can determine if it has all the relevant information regarding a particular spatial extent, i.e., that aggregate values produced for that extent are complete, in this case avoiding the propagation of partial state downstream. How close to the data sources this decision can be taken, determines the computation and communication gains.

Overlapping queries refers to the occurrence of two (or more) queries over the same virtual table, whose area of interest intersect. If those queries are continuous, and the target table uses a consistent *mapping* operation - i.e., one that does not depend of the particular query, as is the case of a spatial partition using a regular grid - then the queries will produce the same results for the partitions that are contained by both areas of interest. A relevant characteristic of a distribution strategy is how it can explore this redundancy to reduce computation and communication effort.

2.4.5 RTree

The basic rationale behind RTree (Random Tree) is to have the homebase as a personal data repository. Storing own data in a personal store is a natural choice for personal archiving and tracking applications, such as CarTel [24] and BikeNet [10]. Besides, it allows the implementation of privacy protection mechanisms - such as the privacy firewall advocated in [17] - since data streams initially flow to a machine trusted by the user. Personal stores can be private - a strong incentive for users to contribute fixed resources to the system - or shared by a group of users.

2.4.5.1 Data Acquisition

In this model, all data acquired by a mobile node belonging to a particular user is sent to (and stored on) his homebase. Given the mobility patterns of a particular users, the homebase will receive, process and store data from a potentially large geographic area. This data will typically cover at least a city-wide area, with most of the data being confined to the region around his/her home and workplace.

2.4.5.2 Query Dissemination

The determining characteristic of this model is that every homebase potentially hosts data that is relevant for a particular query. To reach all the relevant data it would be necessary to distribute each query to all the homebases participating in the system. Given that a node in New York or even Porto will probably not have much information about Lisbon, query distribution can be restricted according to a simple geographic filter. For this purpose, a homebase defines the geographical area it is willing to serve - its *query filter*. This filter can be expressed as a circular area around its geographical address. Query dissemination is then achieved by delivering the query to the nodes whose *query filter* intercepts the query's *area of interest*. By definition, this model does not guarantee that all relevant data is reachable by a given query. If a user, whose homebase geographic address is located in Lisboa, travels to Porto and its query filter defines a radius of 10 km, then the acquired data will be out of reach - in the query initiated acquisition model, data beyond a user's query filter will not be acquired at all. Since the data distribution is essentially random, any dissemination tree is equally effective according to the definition given above - thus the name Random Tree - whichever the tree structure, an intermediate node has no way to determine if it has complete information regarding a particular spatial extent, so partial results have to be propagated downstream until reaching the root.

2.4.5.3 Query Processing

In RTree every node in the aggregation tree has the same role i.e., all nodes participate as data sources and global aggregation nodes. Each node will then instantiate and run both stages. The pipeline output at a particular node is the result of merging partial state data, produced by the

descendants of that node, and its own data source. Each node propagates its pipeline output downstream to its parent that, in turn, uses it as input to its global aggregation pipeline instance. Since a node cannot determine if it has complete information for a particular spatial extent, this process has to be repeated until the root is reached. The query processing tree in RTree is illustrated in Figure 2.4.

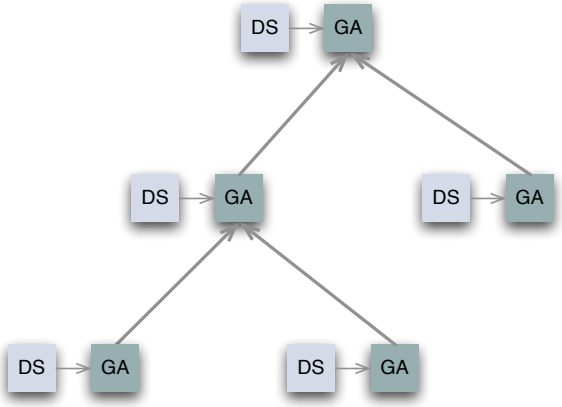


Figure 2.4: RTree aggregation

2.4.5.4 Persistence

Persistence of acquired data is simplified in RTree, since data partition is based on ownership - the acquisition streams of a particular user is stored on his homebase. Regarding derived data, RTree has the particularity that tuples are never bounded before reaching the root, so only at this point a decision can be taken regarding where the tuple is stored and replicated. In order to be reachable by future queries, a tuple can be stored on any node whose query filter contains its spatial extent.

2.4.5.5 Network Model

Regarding data acquisition, given the personal store approach, nodes do not need any information about their peers - data acquired by a node is simply delivered to, and stored on, the respective homebase. Query dissemination, on the other hand, assumes that each node has sufficient knowledge about their peer's query filters, so that a query can be routed to all, or a majority, of nodes that intercept the query area. This calls for a content-based, or geographic, routing substract capable of assembling a tree routed at the query issuer. Additional routing state has to be managed in order to use the reverse path of this tree for aggregation.

2.4.5.6 Network Dynamism

A defining characteristic of the peer-to-peer network model is the inherent dynamism caused by new nodes entering the network and node exiting, either graceful or by failure. This network dynamism has to be taken into account in terms of its impact on data availability and query evaluation. Given the personal store model, acquired data in RTree is not replicated, meaning that node exits will result in data loss. But in this particular context, where value is obtained essentially from aggregated data, the absence of particular raw data points does not necessarily impact the quality of application inferences, provided there is sufficient coverage for the area of interest. There is a natural resilience in the application domain that makes replication of acquired data unnecessary. Derived data, on the other hand has to be replicated in order to prevent data loss.

The aggregation tree structure is sensible to node failures, possibly resulting in a degradation of the quality of results. The severity of this degradation is higher if the failure occurs closer to the root, as more data becomes unreachable. Given that aggregation trees are independent for each query, the impact of a failure is limited to queries running on the failed node, and can be handled by closing and re-issuing the query. Node entries during query execution can result in additional data becoming available, but not being reachable by running queries. This is particular relevant in long standing, continuous queries. To address both node failures and entries, continuous queries can be reissued periodically.

2.4.5.7 Overlapping Queries

The arbitrary structure of the aggregation trees in RTree reduces the opportunity to share computations among continuous queries. A possible approach would be to perform an initial step to discover standing queries that enclose a new query that is about to be issued. In this particular case, the same tree could be used without adding additional effort. The fact that independent trees are used to process overlapping queries has consequences on persistence - since different trees will independently produce data for the same spatial extent and time, it is necessary to have a global persistence strategy that controls the placement and number of replicas. These issues are left open for future work.

2.4.6 QTree

In QTree (where Q stands for quadrant) data partitioning is based on the successive subdivision (or *splitting*) of geographic space into quadrants. Subdivision occurs for a given region if at least one of its quadrants holds a minimum number of nodes (the *minimum occupancy*). This region will then be divided into those quadrants that hold at least the minimum number of nodes. Figure 2.5 depicts spatial subdivision with a minimum occupancy of two. Each node belongs simultaneously to all the quadrants that contain it, down to the smallest - called its *maximum division quadrant*. Partitioning, both for point data and data with a spatial extent, follows the principle that tuples are affected to the maximum division quadrant that fully bounds

it. Minimum occupancy assures that load can be shared, and data replicated, among nodes that fall into the same quadrant. A problem with this approach is that an extents that crosses a first level quadrant can be stored in any node on the system; the same can happen for acquired data in regions with low node density. This means that in some cases a query has to be flooded to every node. In order to reduce query scope it is necessary to establish a *minimum division* level. Assuming a minimum division level d_{min} means that geographic space is assumed to be totally divided up to d_{min}, i.e. all quadrants at this level have minimum occupancy. In this scheme, a tuple is never stored above d_{min}, if it crosses more that one quadrant at that level it has to be replicated among the quadrants it intercepts. For areas where this assumption does not hold (e.g., bottom left in 2.5), QTree cannot guarantee that all relevant data can be found - an issue that could be address by assigning more than one coordinate to particular nodes using a load balancing criteria.

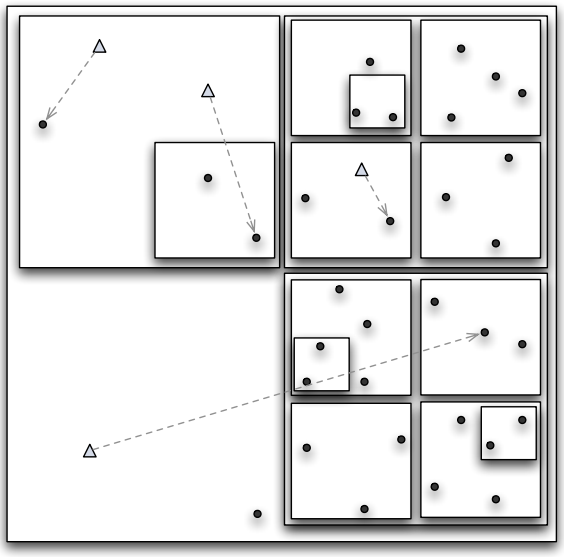


Figure 2.5: Spatial subdivision in QTree with minimum occupancy of 2 nodes

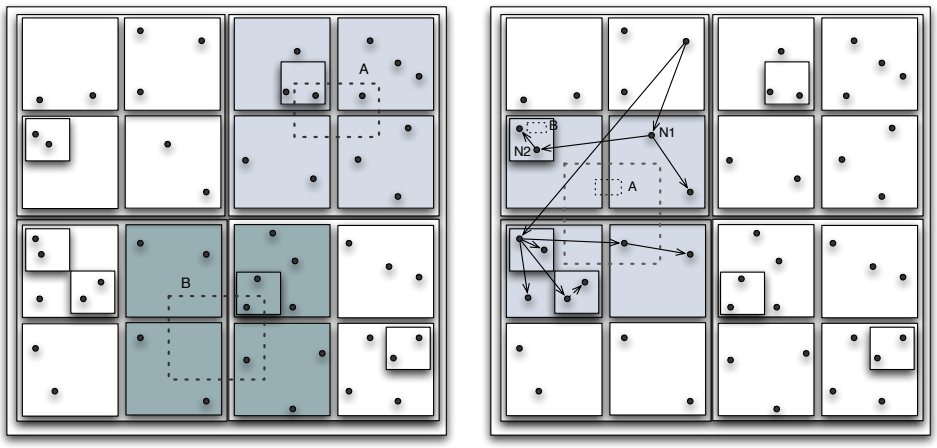
2.4.6.1 Data Acquisition

A mobile node contacts its homebase in order to bind to an acquisition node in the fixed infrastructure. In QTree, a suitable acquisition node is any node belonging to the maximum division quadrant that includes the mobile node's physical location. Figure 2.5 illustrates four mobile nodes, represented as triangles, connected with the respective acquisition nodes.

In delayed acquisition, data is fed initially to the homebase and, from this point, forwarded to acquisition nodes according to the same principle, i.e., each tuple has to be forwarded to any node belonging to the maximum quadrant division that bounds it.

2.4.6.2 Query Dissemination

In order to reach all relevant data, a query has to be distributed to all nodes within its *search area*, where this area is defined as the union of the quadrants, at the minimum division level, that completely cover the query area. This is illustrated in Figure 2.6a, where shadowed quadrants represent the search areas for queries A and B.



(a) Search area (b) Dissemination tree

Figure 2.6: Query dissemination and processing

A query can now be distributed by recursively subdividing the search area until all nodes are reached, building a distribution tree in the process as depicted in Figure 2.6b. Initially, the node that issues a query divides the world into four quadrants and finds the interceptions between the search area and each quadrant. For those quadrants that have minimum occupancy - and have non-empty interceptions, it chooses an aggregation node and forwards the query together with the quadrant area. For quadrants that do not have minimum occupancy, the node assumes the aggregation role and forwards the query to all peers in that area - these nodes become the tree leaves and will act as data sources. The chosen aggregation nodes, in turn, repeat the procedure by subdividing the assigned quadrant and forwarding the query according to the same logic.

2.4.6.3 Query Processing

Aggregation is performed using the reverse path of the query dissemination tree. The aggregation tree structure is depicted in Figure 2.7 - note that global aggregation nodes also join the tree as data sources. Tuples are bound at the tree level where the assigned quadrant completely encloses its spatial extent, i.e., upper tree levels will not hold additional data regarding that extent. Bound tuples can then be forwarded to the query root without further processing. In Figure

2.6b, extents *A* and *B* are bound at the nodes *N1* and *N2* respectively. Another relevant characteristic in QTree is that although nodes closer to the root cover wider areas, they only aggregate data for spatial extents that are not completely bounded at lower levels of the aggregation tree.

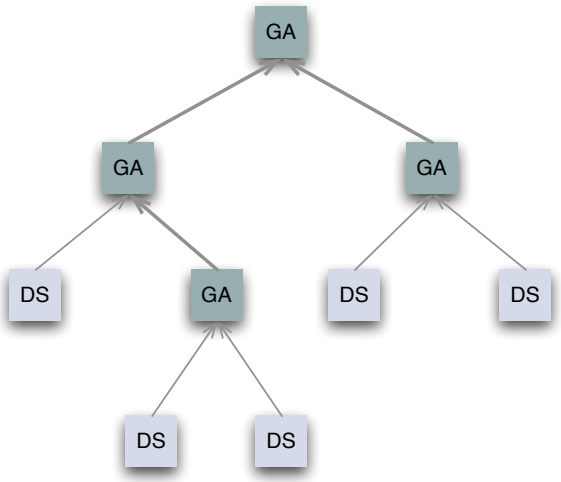


Figure 2.7: QTree aggregation

2.4.6.4 Persistence

A node can decide where to persist a tuple when it becomes bounded. If this occurs at a quadrant below the minimum division level, it can be stored in any peers belonging to that quadrant. Below this level, tuples have to be stored in all the minimum division quadrants they intersect.

2.4.6.5 Network Model

QTree assumes that nodes have sufficient information about network membership in order to support data acquisition bindings and to build query dissemination trees. An approach similar to what is proposed in P2PR-tree [39] to build a distributed spatial index, can be used for this effect. In this approach, a node needs to know at least one peer on the neighboring quadrants for each division level, together with the respective quadrant bounds, and all peers in his maximum division quadrant - as illustrated in Figure 2.8. This way a node will hold more information regarding neighboring nodes than distant ones, and there is a possible path between any two peers. For acquisition binding, a node can use its routing information to find the smallest quadrant it knowns that includes the mobile node location. The target node will then repeat the procedure until an indivisible quadrant is reached. The same routing information is sufficient to build a query dissemination tree. As in P2PR-tree, routing information can be complemented, as peers interact, to maintain redundant paths and reduce routing costs.

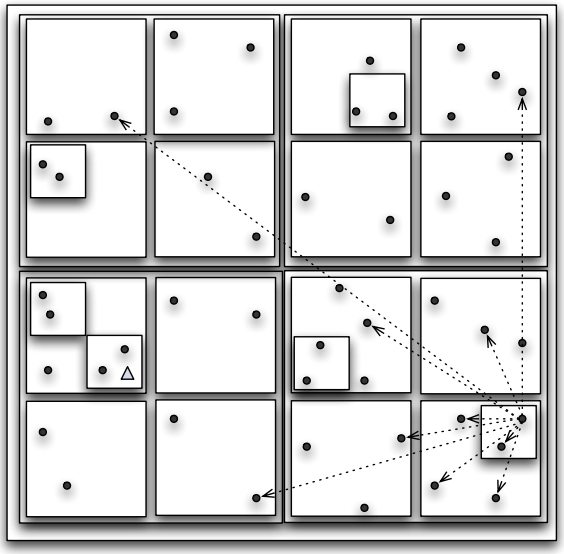


Figure 2.8: Routing information in QTree

2.4.6.6 Network Dynamism

Concerning network dynamism, two scenarios have to be considered when a node exits the system. In the simplest case, the exit has no impact on its maximum division quadrant; this happens if the minimum occupancy still holds. If this is not the case, the maximum division quadrant will have to merge with the upper level quadrant. Since data bounded at one level is also bounded at the upper level, there is no need to migrate stored data between nodes. Assuming data stored by a node is replicated in peers on its maximum division quadrant, exits will not cause data loss. Similarly, a node entry might result in one of two scenarios. In the simplest case, the entry does not result in splitting and so has no impact on data partitioning. If splitting occurs, stored data has to be repartitioned so that each tuple is stored at the smallest quadrant that completely bounds it.

As in RTree, node failures in an aggregation tree impacts query results and the severity of this impact depends on how close to the root the failure occurs. But in this case, failure at a tree node requires rebuilding the tree only for the affected quadrant, after any necessary merging. When a node joins the network during continuous query execution, the aggregation tree can be adapted to reflect the change. If no subdivision occurs, the node responsible for aggregation on the maximum division quadrant where the node lies can simply forward the query to the new node. If splitting occurs, any node bounded by the new maximum division quadrant can now assume the aggregation role for that quadrant and forward the query to the new node.

2.4.6.7 Overlapping Queries

As presented, QTree will generate different query dissemination and aggregation trees over the same area, depending on the routing information each node holds, resulting in repeated computation of the same query over a given area. Query dissemination in QTree can be modified so that the same tree is used for different queries over the same quadrant. As before, query dissemination proceeds by choosing a node as the aggregator for a given quadrant, and forwarding the query to this target node. If a query covering the same quadrant has been issued previously, the target will be already engaged on an aggregation tree, although possibly at a different level. The target can then forward the query downstream or upstream until the correct tree level is reached. The new tree and the existing tree can then be joined at this point. The aggregation node at the junction point, will keep the context for both installed queries and forward data toward each query root according to the respective query constraints.

2.4.7 NTree

NTree (where N stands for neighbor) partitions data according to a *geographic hashing* approach, where each data tuple is *hashed* to a two dimensional coordinate pair - its *key* - and assigned to the fixed node whose geographic address minimizes the euclidian distance to this key - the tuple's nearest neighbor. The key for point data is simply given by the associated coordinate, while for data with spatial extent any hashing function that preserves data locality can be used - for instance, the centroid. Relevant data for a query, in NTree, is defined as the set of tuples whose keys are bounded by the query area. In order to find all relevant data for a given query, it would be necessary to determine the intersection of the query area with the cells of the *Voronoi* diagram obtained from the geographic coordinates of the fixed nodes. NTree assumes an approximation to this spatial decomposition, for instance using a *query filter* as in RTree, but where the filter covers an area that approximates the corresponding *Voronoi* cell. Figure 2.9 illustrates data partitioning in NTree. Tuple A (spatial extent) and B (point data) are stored in nodes N1 and N2 respectively. The circle around node N4 represents its approximation to the *Voronoi* cell used for query dissemination.

2.4.7.1 Data Acquisition

As in QTree, in order to support data acquisition a mobile node has to bind to a fixed node. In this case, the mobile's homebase has to find the closest fixed node, given its physical location. Due no node mobility, this process has to be repeated regularly. In delayed acquisition mode, the homebase will initially receive the acquired data and proxy it to the destination nodes.

2.4.7.2 Query Dissemination

In NTree query dissemination and aggregation follow independent paths, so there is no specific requirement regarding the structure of the dissemination tree. The only requirement being that

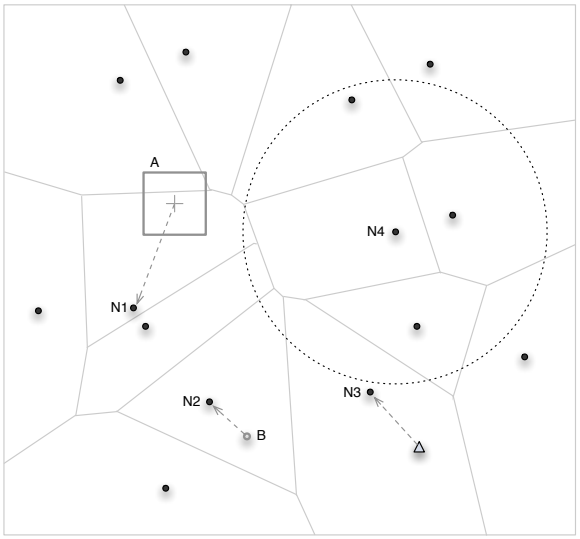


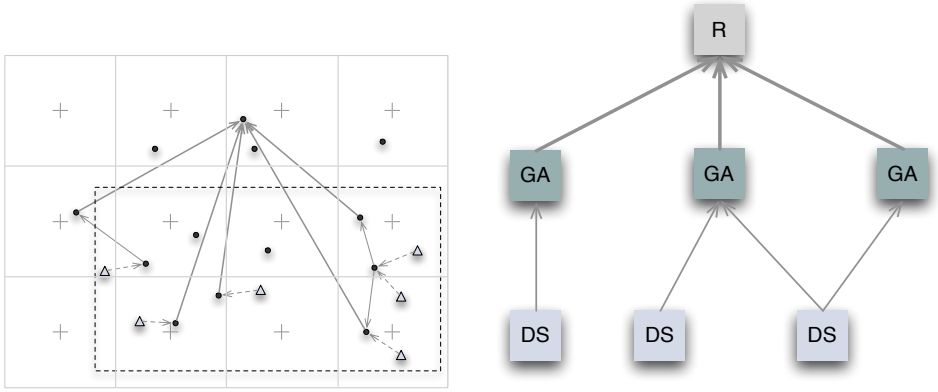
Figure 2.9: NTree data space partitioning

a query reaches the majority of relevant nodes. Using the filter approximation described above, this is accomplished by routing a query to all nodes whose filter intersects the query area.

2.4.7.3 Data Aggregation

NTree uses a bottom up approach to build the aggregation structure. When a node receives a query it will act as a data source and instantiate the associated pipeline. Data tuples produced by this pipeline are forwarded to the respective nearest neighbors, where they are processed by global aggregation pipelines. Since data can be re-maped at the global aggregation stage, unbounded data is again routed to the nearest neighbor - thus creating a multi-level aggregation path (multi-level aggregation is discussed in Section 2.4.7.4). Finally, bounded data is forwarded to the query issuer. Global aggregation nodes might not have been contacted during the initial query dissemination stage; in this case the data origin has to propagate the query to the tuple's destination.

Assuming all nodes have a consistent view of the network membership, global aggregation for a particular spatial extent occurs at a single point, and consequently data for that particular extent is bounded at that point. Figure 2.10 illustrates a (single-level) aggregation scenario where data is mapped to a regular grid. Data acquired from mobile nodes is forwarded to the data source pipeline running on the closes node, which maps data to the respective grid cell. The output of this pipeline is then forwarded to an aggregation pipeline running on the node that is nearest to the cell centroid. The global aggregation pipeline, in turn, merges all data concerning a grid cell and forwards the result to the root. It can be seen that the resulting aggregation structure in NTree is not a tree but a set of connected trees, since nodes can produce



(a) (b)

Figure 2.10: NTree aggregation

data concerning different spatial extents.

This approach is akin to *MapReduce*, as presented in [8], where the *map* and *reduce* functions are performed by the data source and global aggregation pipeline respectively. A data source typically applies a *mapping* operator to data tuples that results in assigning a key and associated spatial extent, thus producing *intermediate* key/value pairs. Data sources aggregate local data with the same key before sending to global aggregators, a logic similar to that of a *combiner* function in *map reduce*. Finally the global aggregation pipeline implements a *reduce* function by merging data with the same intermediate key from several data sources and producing a final result.

2.4.7.4 Multi-Level Aggregation

In NTree, the number of aggregation levels is not determined by the query dissemination process. Instead this structure is determined by the data mapping operators. If the data source stage maps tuples into street segments, it implicitly creates a per-segment aggregation level; and if the global aggregation stage re-maps data into streets, it implicitly creates a per-street aggregation level.

Applications may benefit from additional levels in order to deal with queries over large spatial regions. Consider a country level query to retrieve the cities where traffic congestion occurs. This could be handled by a data sourcing stage that maps GPS readings to street segments and produces local averages per segment. The global aggregation stage merges all averages for a given segment and, if traffic congestion is detected, re-maps the data to the city level. In this case, the nodes responsible for city level aggregation would receive data relative to a potentially large number of street segments. The application would benefit from additional street, council

and region data mapping levels. A global aggregation pipeline can achieve this using a *multi-level mapping operator* as the last step of the aggregations stage. This operator would map data successively to street, council and region. As data is mapped to the next level it is forwarded to the corresponding nearest neighbor where a new aggregation stage occurs. The disadvantage of using this approach is that the intermediate mapping levels are not accessible to queries and other tables. An alternative is to use virtual table derivation by defining a table for each required mapping level. While in RTree and QTree, table derivation results in a single global aggregation pipeline that is replicated in each aggregation level, in NTree each table derivation results in a global aggregation sub-stage that together determine an aggregation path. Data routing in NTree with multiple tables proceeds as follows: If the output from a stage is bounded at the current node, it proceeds to the next stage on the same node; if not bounded, data is routed to the same stage on the nearest neighbor.

2.4.7.5 Persistence

Given its design, in NTree tuples are bounded at their nearest neighbor - the single point in the aggregation tree where all data with a common spatial extent flows into. In order to be found by future queries, this is also the point where data has to be stored. Replication can be performed in neighboring nodes.

2.4.7.6 Network Model

NTree communication requirements can be supported by a geographic routing substract , where nodes hold information about peers in their neighborhood, enabling them to route data to the closest peer to a given geographic address. Query dissemination can be based on this principle by initially targeting a query to its nearest neighbor, according to the geographic hashing function, and then forwarding to neighboring nodes whose filters intercept the query area. Routing of query data and results can be enhanced by grouping messages with a common destination in intermediate nodes to reduce network overhead.

2.4.7.7 Network Dynamism

As in QTree, changes in network membership caused by node exits and entries may require that data is repartitioned among nodes. When a node enters the network, it has to reclaim ownership over data in surrounding nodes for which it is the now the nearest neighbor. Conversely, when a graceful exit occurs, data owned by the exiting node is delegated to nodes in the vicinity. To prevent data loss due to node failure, data has to be replicated among neighboring nodes. Regarding continuous query processing, NTree is the most resilient of the three approaches since it does not require a fixed aggregation tree. When a global aggregation node exits (either gracefully or by failure), and after a period of consolidation where nodes regain a consistent view of the network, other nodes will assume its role as they become the nearest neighbors for tuples previously assign to it.

2.4.7.8 Overlapping Queries

Overlapping queries can be naturally handled by NTree since mapping partitions are consistently handled by the same nodes. At the end of the processing path, pipeline results have to be forwarded to each of the query issuers. This can be accomplished by applying, at this point, the constraints associated with the standing queries to the pipeline output tuples, in order to decide where data should be forwarded.

3. Prototype

This chapter presents the 4Sensing prototype designed and implemented for the validation and analysis of the models presented in Chapter 2. The first two sections describe the development platform and prototype scope. Section 3.3 presents the fixed node architecture and describes the implemented services, while the subsequent sections discuss these services in detail.

3.1 Development Platform

The prototype was implemented in Groovy [20], an object oriented dynamic programming language that runs on top of the Java Virtual Machine. The main motivation for using Groovy is the fact that its dynamic features and closure support make it particularly well suited to implement Domain-Specific Languages, thus simplifying the prototyping of a Virtual Table Definition Language. Closures are anonymous functions that can be passed as arguments and stored as variables, and are extensively used for the definition of pipelines in VTLD. The prototype also takes advantage of Groovy's dynamic features to build a set of generic stream operators, such as the possibility of adding behavior to objects in runtime and intercepting method calls. Being based on the JVM, Groovy shares with Java the advantage of being abstracted from heterogeneous deployment environments.

3.2 Scope

The main goal of the prototype is to support the comparative analysis of the three proposed strategies using a case-study application. The prototype focuses on the virtual table concept and the distribution model. To that end, it implements a Virtual Table Definition Language used by applications to describe virtual tables in terms of stream inputs and processing pipelines as presented in the Chapter 2. Additionally, it creates a common framework that supports particular distribution strategies and implements the proposed RTree, QTree and NTree approaches.

Some aspects of the model that fall outside this scope have been simplified and others have been left for future work. In particular, the prototype takes for granted a one-hop overlay network model, where each node knows the complete network membership, and focuses on the fixed infrastructure services, taking a simplified approach to data acquisition and delivery support. Experimental work has been centered around real-time data acquisition and continuous queries, while data persistence, replication and historical queries have been left for future work.

3.3 Fixed Node Architecture

The prototype implements a framework supporting a fixed infrastructure for participatory sensing applications. This framework is structured as a set of services hosted by a *node runtime* running independently on each element of the fixed infrastructure.

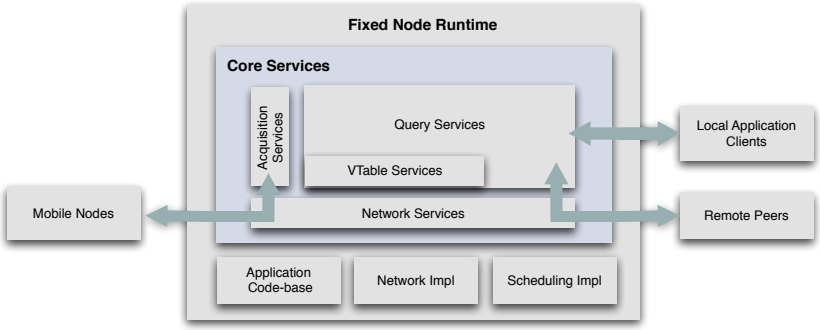


Figure 3.1: Fixed Node Runtime

The node runtime - represented in Figure 3.1 - provides the interfaces that allow external components to interact with the core services - local application clients, mobile node and remote peers. Internally, the runtime provides low level service implementations supporting the core services - in particular, it provides overlay network management and task scheduling implementations. The prototype focuses on the core services, described below, while other aspects of the node runtime, application clients and mobile nodes have been adapted for a simulation setting.

Virtual Table Services Implements the set of services related to virtual table definition and pipeline assembly. Virtual Table Services provide the library of components implementing the supported stream operators.

Query Services Implements the main logic regarding query setup and operation. Supports the query interface exposed to application clients and handles query distribution and control. The Query Services manage the active queries, and associated pipelines, and control the data flow between local and remote pipelines.

Acquisition Services Controls the delivery of data from homebase to acquiring nodes. In acquiring nodes, the Acquisition Services manage sensor data delivery to registered pipelines.

Network Services Hosts a peer database modeling a one-hop overlay network, provides interfaces for membership querying and supports the construction of random dissemination trees. Additionally, Network Services support message exchange, abstracting other service layers from the particular runtime network implementation.

Figure 3.2 illustrates the main data flows described above, which will be detailed in the sections covering each service.

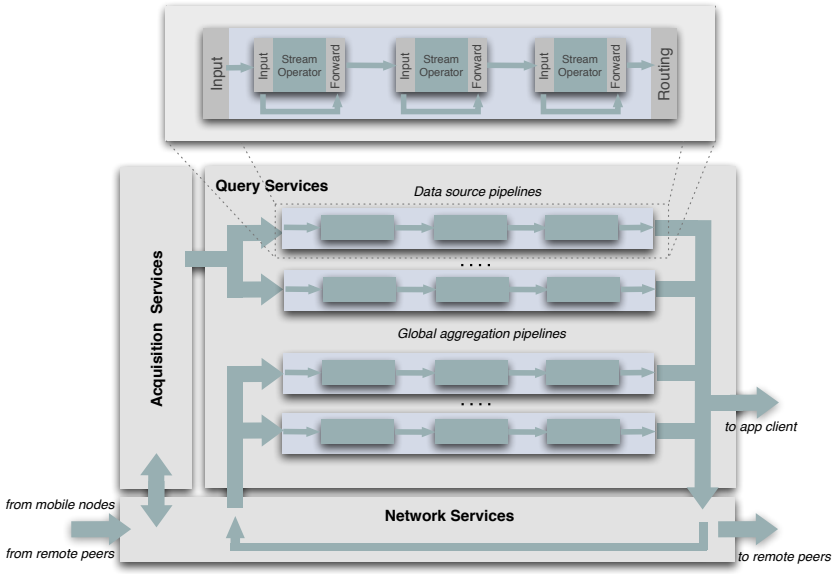


Figure 3.2: Main Data Flows

3.4 Virtual Table Services

Virtual Table Services comprehend the set of services and components supporting virtual table definition and operation. Domain application use these services to specify virtual tables; internally, they are used by Query Services for table lookup and materialization when a query is received.

This section focuses on the aspects of the prototype that are relevant from the standpoint of application development. In particular, how application services are defined in terms of virtual tables and data representation, how components - the building blocks used to implement pipelines - work and can be extended to support domain-specific requirements.

3.4.1 Data Representation

Tuples are represented by Groovy bean classes, where each bean property corresponds to a tuple attribute. Applications define specialized types by extension of the *Tuple* base class - presented in Listing 3.1 - which defines the common metadata properties described below:

mNodeId The identifier of the mobile node that originated the tuple. This property is present in sensor data.

Listing 3.1 Tuple class

```
class Tuple {
    UUID mNodeId
    UUID peerId
    double time
    double lat
    double lon
    Rectangle2D boundingBox

    public Tuple derive(Class tupleClass)
    public Tuple derive(Class tupleClass, propertyMap)
}
```

time A timestamp, assigned by mobile nodes to sensor data, representing the moment of acquisition.

peerId Identifies the origin of the tuple in the fixed infrastructure; *peerId* is present in tuples resulting from an aggregation operator.

lat, lon The geographical coordinates associated to point data.

boundingBox The spatial extent for data associated with a geographic region. A *boundingBox* represents a rectangular geographic area defined by the minimum latitude and longitude values, a width and a height. The *boundingBox* can be assigned by the acquiring node or by the query processing pipeline as part of a *mapping* operation.

Additionally, the tuple base class defines a *derive* operation, used to create new instances of the tuple classed passed as argument. Attributes of the derived tuple, that are also present in the source (including meta-data), will be assigned the source values. Optionally, a property map can be used to assign values to attributes of the derived instance.

3.4.2 Virtual Table Definition

A domain application is modeled through the definition of a set of virtual tables. For this effect, developers use a virtual table definition language (VTDL) that will be detailed in this section. A virtual table is a derived stream specified in terms of its inputs and stream transformations. Stream transformations are structured in a data sourcing stage, that feeds the query processing tree with local data, and a global aggregation stage that merges data from descendent nodes.

Virtual tables can be defined by direct derivation from sensor data or from another table. The construct for the former case is shown in Definition 3.1, which follows closely the pseudo-code examples presented in Chapter 2, the difference being the possibility of breaking down the global aggregation stage into sub-stages. These sub-stages are optional and developers can still specify a unique global aggregation transformation. The rationale behind sub-stages has to do with the construction of the bottom-up aggregation tree in NTree as explained in 2.4.7.4. It is a convenience construct that introduces an alternative to the usage of multi-level mapping operators and table derivation for multi-level aggregation. This approach can be used when exposing the intermediate levels as tables is not relevant. For QTree and RTree, it has no practical effect and is equivalent to the definition of a single global aggregation stage.

Definition 3.1 Base virtual table definition

base-virtual-table-definition ::=

[**sensorInput** '(' <sensor-definition-name >')']+

dataSource {<*pipeline-definition* >}

[**globalAggregation** { <*pipeline-definition* >}]*

The VTDL construct for table derivation is shown in Definition 3.2, where <*virtual-table-name*> is the fully qualified name of the base table [1]. In this case, only global aggregation is specified, since data sourcing is defined in the base table.

Definition 3.2 Derived virtual table definition

derived-virtual-table-definition ::=

tableInput '(' <virtual-table-name>')'

[**globalAggregation** { <*pipeline-definition* >}]+

Each stage is assigned to a pipeline definition - a sequence of VTDL instructions that specifies how to assemble a pipeline from individual components. Section 3.4.3 will provide additional detail regarding pipeline components, while Section 3.4.4. details the particular component types supported by the prototype and the corresponding VTDL constructs.

3.4.2.1 Virtual Table Manager

The Virtual Table Manager is the local repository of virtual table definitions. It is responsible for executing the VTDL scripts, when a table lookup is requested, producing the corresponding *VTableDefinition* instance that holds the list of input sensors and the pipeline definitions

[1] The fully qualified name of a table is given by the name of the VTDL script file preceded by the Groovy package name defined in the script.

for each processing stage. This definition - whose structure is presented in Listing 3.2 - provides the VTDL operations and logic for table definition and, in particular, for table derivation. The *tableInput* operator results in importing the definition of the source table. The resulting *VTableDefinition* holds the definitions for all tables chained by *tableInput*, as if this clause represented a macro that expands to the referenced table.

Listing 3.2 VTableDefinition class

```
public abstract class VTableDefinition extends Script {
    public void sensorInput(Class sensorClass) {...}
    public void tableInput(String tableName) {...}
    public VTableDefinition dataSource(Closure pipelineDefinition) {..}
    public VTableDefinition globalAggregation(Closure pipelineDefinition) {...}

    public getStage(String stage) {...}
    public getSensorInputs() {...}
}
```

Pipeline definitions are represented as Groovy closures - a code block containing the sequence of VTDL assembly method calls that can be invoked to assemble a pipeline - a process detailed in Section 3.4.5.1.

3.4.3 Pipeline Component

Pipelines are structured as a sequence of components, each implementing a particular stream operator. These components share a common structure and behavior as defined by the base *Component* class - presented in Listing 3.3.

The basic logic of a component is to receive a tuple through the *input* method, operate over the data and possibly *forward* a result tuple to the next component in the pipeline sequence. Particular component types implement specific input logic, determining which tuple types (bean classes) they accept, and forwarding the tuples they do not handle - this is illustrated in the pipeline inset in Figure 3.2. Consequently, data inserted into the pipeline, or forwarded by a component, is dispatched to the next component in the linked sequence that can handle that particular tuple type.

3.4.4 Component Library

The prototype implements a library of specialized components covering the stream operators presented in Chaper 2. The VTDL constructs for these specialized components are shown in Definition 3.3 and the subsequent sections detail their logic and usage.

Listing 3.3 Component class

```
public abstract class Component {
    Component next

    public void init() {}
    public boolean canDispose() { return true }
    public void dispose() {}

    public void streamInit() {...}
    public boolean streamCanDispose(){...}
    public void streamDispose() {...}

    public abstract void input(input) { forward(input) }
    protected final void forward(input) { next?.input(input) }
}
```

Definition 3.3 VTDL pipeline definition

```
pipeline-definition ::= [
    ( process (<processor >| <closure>) ) |
    ( filter <closure>) |
    ( classify <closure>) |
    ( groupBy '(' (<attribute-list >| <closure>) ')' { <pipeline-definition>}) |
    ( timeWindow '(' size: <integer>, slide: <float>')' ) |
    ( aggregate '(' <class>') <closure>)
]+

attribute-list ::=
    '[' [<string>]+ ']'
```

3.4.4.1 Processor

A processor is a generic operator and the basis for any application specific transformations. The *Processor* class does not implement any pre-defined transformation, it is the domain application that provides this logic, by extension or method injection as explained below. What *Processor* provides is a generic dispatching mechanism to abstract the developer from the details of pipeline processing.

Applications add behavior to a processor by providing any number of *process* methods, each with a particular input type. When a tuple is received by the *input* method, the processor determines if an appropriate method exists based on the tuple type. If a suitable *process* method is available then the tuple is delivered to and handled by this method, otherwise the tuple is forwarded to the next component. The result of a *process* method is forwarded to the next component in sequence or, if the transformation requires forwarding more than one data tuple, the *forward* method can be explicitly called within *process* for this purpose.

Two VTDL constructs, shown in Definition 3.3, can be used to specify a processor depending on the flexibility requirements of the domain application. If a processor has to handle several input types, developers must extend the *Processor* class with the required process methods and use the first construct, as shown below.

```
class PotholeInterpolator extends Processor {
        void process(GPSReading r) { /* store last GPSReadings */}
        PotholeReading process(AccelerometerReading r) {
                 /* use GPS interpolation, return combined reading */
                 GPSAccelReading interpolation = new GPSAccelReading();
                 ...
                return interpolation
         }
}

process new PotholeInterpolator()
```

When a unique process method is required, there is no need to create an extended class. In this case, the operation is specified directly in the process declaration as a Groovy closure and injected into the generic *Process* instance.

```
process { Temperature t -> t.degrees = (t.degrees - 32) * 5/9; return t}
```

The example above results in a Process component with one *process* method that converts Temperature readings from Fahrenheit to Celsius.

3.4.4.2 Triggered and Periodic Processors

A Triggered processor is implemented by extending the *TProcessor* class. As in a standard processor, *process* methods are responsible for handling data input. The particularity of the triggered processor is that data output can be decoupled from input by implementing an *output* method. The *output* method is scheduled for execution, when input is received, using the delay parameter passed on instantiation. Output will only be re-scheduled after the previous call terminates, thus occurring at most once per period. The *output* method can perform arbitrary processing and forward any number of tuples. If the *output* method has a boolean output type, the return value determines if an immediate reschedule is required.

The example below illustrates the definition of a triggered processor that forwards data at a maximum rate of one tuple every 10 seconds.

```
class Throttle extends TProcessor {
        Tuple last
        void process(Tuple input) {last = input }
        void output() {forward(last)}
}

process new Throttle(10)
```

A periodic processor is implemented by extending the *PProcessor* class. As in a triggered processor, input and output are decoupled; in this case the *output* method is called periodically, independently of the input flow. The parameter passed on instantiation defines the output periodicity in seconds.

3.4.4.3 Filter

A filter is a specialized processor, implemented by the *Filter* class, where the *process* methods implement boolean filter conditions. An input is forwarded to the next component if this condition evaluates to *true*, and is dropped otherwise. Filter conditions can access the *last* input that passed the filter and auxiliary functions *changePercent* and *changeAbsolute* support conditions based on changes on particular attributes.

Filters can be defined using the *filter* construct, specifying the filter condition as a closure. In the first example below, the filter will drop tuples with out-of-order timestamps, while the second filter only forwards input tuples if *avgSpeed* attribute changed more than 5% relative to a previously forwarded tuple.

```
filter {Tuple input ->  input.timestamp > last.timestamp}
filter {AggregateSpeed a -> changePercent('avgSpeed',5)}
```

3.4.4.4 Classifier

Classifiers are also specialized processors. A classifier is defined by extension of the *Classifier* class or by method injection. The difference relative to a standard processor is that the *process* call occurs only if the input tuple is bounded. If this is not the case, the input is simply forwarded to the next component.

3.4.4.5 GroupBy

A *groupBy* operator, is a specialized component implemented by the *GroupedComponent* class. It splits an input stream according to a condition, given by a closure, or a list of attribute names. It maintains a set of active partition pipelines and dispatches input to the appropriate partition. Partition pipelines are handled dynamically, i.e. they are created lazily when an input is received for a partition that is not yet instantiated and they are disposed when all components agree on the *canDispose* call.

3.4.4.6 TimeWindow

The class *PTimeWindow* implements a time-window, as presented in the Chapter 2. Placing of a window on a pipeline is flexible. A windows placed inside the *groupBy* declaration is instantiated independently for each partition; given that partitions are instantiated on-demand, partition windows will not be synchronized. A time-window forwards a tuple sequence followed by an *EOS* instance - used as the *end-of-stream* indicator defined in Chapter 2.

3.4.4.7 Set

A *set* operator is a specialized component, implemented by the *Set* class. As in *groupBy*, it is configured by a partitioning condition, given by a closure or a list of attribute names. The latest tuple received for each partition is stored internally and the complete set is forwarded when an *EOS* instance is received (in *eos* mode) or whenever a change occurs (*change* mode).

The example below illustrates the usage of the set operation in *eos* mode, resulting in the output of (at most) one tuple per mobile node, every 10 seconds - the latest tuple received within a 30 seconds time window.

```
timeWindow(mode: periodic, size:30, slide:10)
set(['mNodeId'], mode: eos)
```

In change mode, the *ttl* parameter defines the time to live of stored tuples, where the value zero specifies a hard-state approach, as discussed in Section 2.3.2. In *eos* mode, the set state is reset whenever *EOS* is received.

3.4.4.8 Aggregator

An aggregator is a specialized component, implemented by the *Aggregator* class. The *aggregate* construct receives the class of the tuples that will be produced by the operator. The second argument is a closure that determines the aggregation operations that will be applied on the input data. This closure specifies a sequence of *count*, *sum* or *avg* operations, as presented in Definition 3.4.

Definition 3.4 VTDL aggregation operator definition

aggregation-operator ::=

count(<tuple>, <string>) |

sum(<tuple>, <string>, <string>) |

avg(<tuple>, <string>, <string>) |

avg(<tuple>, <string>, <string>, <string>)

Each operator receives the input tuple as the first argument, followed by string arguments that specify the input and output tuple attributes that the operator will act upon:

count The string argument in *count* defines the attribute name of the output tuple, where the result will be stored.

sum The first string argument is the name of the input attribute to be summed; the second string argument is the name of the output attribute where the result will be stored

avg The *avg* operator provides two distinct forms; the first form, used to produce local averages, has a similar structure to the *sum* operator. It produces an average by summing the values of the input attribute over the input set and dividing by the set cardinality. The second form is used for global aggregation and produces an average by combining partial values; the first two arguments specify the input attributes that contain partial sum an count values. In this form the operator outputs an average, produced from the partial sum an count values, together with the total sum and count values.

The aggregator operates over tuple sets, producing output when a stream termination indicator is received from the set or time-window components. Output metadata is inferred from the input set - the timestamp is given by the maximum timestamp received; spatial extent is given by the extent of the input set, if it is consistent across all tuples, otherwise it is given by the query area of interest. Finally, the local peer identifier is assigned to the output.

If *aggregate* is inside a *groupBy* operator, the partition key attributes - whose values are constant for each partition - will also be assigned to the aggregate output (if they are defined in the output tuple class).

3.4.5 Pipeline Structure and Operation

Heading the component sequence, a front-end Pipeline class - presented in Listing 3.4 - mediates the interactions between services and the components regarding life-cycle management and data input.

In response to a query, a pipeline is assembled, operated and disposed. Its life-cycle is managed through the set of operations described below.

init Initialization is triggered immediately after pipeline assembly. The *init* operation results in the equivalent call in each of the linked Components. During this stage, components can run any initialization procedures such as task scheduling. After initialization all components must be ready to receive *input* calls for data processing until the reception of a *dispose* call.

canDispose Disposal of a pipeline can occur as a consequence of query termination or, for the partition pipelines managed by a *groupBy* component, during the query lifetime, in order to reduce resource consumption. In the latter case, the dispose process has two stages. First *canDispose* is called to determine if all components are ready for disposal. Components should return *true* if they do not store relevant state information.

Listing 3.4 Pipeline class

```
public class Pipeline {
    protected Component head

    public Pipeline addComponent(Component c) {...}

    /* life-cycle methods */
    public void init() { head?.streamInit() }
    public boolean canDispose() { return head?.streamCanDispose() }
    public void dispose() { head?.streamDispose() }

    /* pipeline input */
    public void input(tuple) { head?.input(tuple) }
}
```

dispose A dispose call occurs when all components in a pipeline agree on a *canDispose* call or when the query is terminated. The *dispose* operation results in the equivalent call in each of the linked components. During this stage, components should perform cleanup tasks such as terminating any scheduled tasks.

During query operation data is delivered to a pipeline through the *input* method, which delegates to the head of the component sequence. The tuple is subject to the dispatch flow described in Section 3.4.3, being consumed by the first component that handles the tuple type.

3.4.5.1 Pipeline Assembly

The pipeline assembly logic is implemented by the *Assembler* class - presented in Listing 3.5 - that supports the VTDL constructs related to pipeline definition. The assembly process is the result of executing a pipeline definition in the assembler context (using the *Assembler* as the closure delegate, in the Groovy terminology).

Assembly can occur on query setup, and during query operation. In the latter case it is triggered by the *groupBy* operator when a new partition pipeline is required. The *Assembler* hierarchy defines which VTLD constructs are available for each case - *AssemblerBase* defines the common VTDL constructs for both top-level and partition pipelines, while *Assembler* adds the *groupBy* construct available only for top-level pipelines.

Listing 3.5 Assembler class

```
class Assembler extends AssemblerBase {
    public Assembler(Pipeline target)

    /* from AssemblerBase */
    public Pipeline addComponent(Component c) {...}
    public Pipeline process(Processor p) {...}
    public Pipeline process(Closure processor) {...}
    public Pipeline aggregate(Class outputClass, Closure aggregator) {...}
    public Pipeline filter(Closure filter) {...}
    public Pipeline classify(Closure classifier) {...}
    public Pipeline set(parameters, partitionKey) {...}
    public Pipeline timeWindow(parameters) {...}

    public void assemble(Closure pipelineDefinition) {...}

    /* from Assembler */
    public Pipeline groupBy(partitionKey, Closure partitionDefinition) {...}
}
```

3.5 Query Services

The Query Services implement all aspects related to query setup and processing. It provides a query interface for application clients and implements the logic supporting query dissemination, local configuration, distributed processing and result routing. The base *QueryServices* class supports the common behavior to all distribution strategies, while RTree, QTree and NTree are implemented as specializations of this common base.

Section 3.5.1 presents the query interface, while the subsequent sections detail the main processes and architecture aspects supporting query operation.

3.5.1 Query Interface

Application clients use the querying interface provided by the Query Services to start and close queries and receive results. To process a query, a client instantiates a *Query* object, parameterized with the target virtual table, and configures an area of interest specified by the latitude and longitude boundaries.

```
def q = new Query(<virtual-table-name>).area(minLat: <min-lat>,
        minLon: <min-lon>, maxLat: <max-lat>, maxLon: <max-lon>)
```

Query results are received and processed by the closure provided to the *runQuery* call. In the example below, results are displayed in the application console as they arrive to the query issuing node. Finally, the *closeQuery* call terminates a query.

```
services.query.runQuery(q) {
        result -> println result        /* result handling code */
}
...
services.query.closeQuery(q)
```

3.5.2 Query Setup and Processing

Query setup is initiated when a node receives a query from a local application client, or from a remote peer - during the query dissemination process, resulting in the following setup procedures:

Pipeline instantiation
: Pipeline instantiation involves accessing the Virtual Table Services to obtain a virtual table definition. Depending on the particular distribution strategy and the point in the dissemination tree, data source and/or global aggregation pipelines are instantiated and assembled.

Input and output setup
: Data source pipelines are assigned to sensor inputs in the Acquisition Services according to the virtual table definition. A data routing component is appended to the pipeline to handle output.

Query Registry update
: Query setup proceeds by updating the Query Registry - a structure that holds runtime information concerning the standing queries. For each stage, a query *context* references the associated pipeline and the aggregation tree information used to assist output data routing. From this point, the pipeline is connected to the global aggregation structure.

In NTree, given the bottom-up approach used to build the aggregation tree, query setup can also occur when data for a previously unknown query is received. For this purpose, NTree encapsulates the query specification in the data message forwarded to remote nodes.

3.5.3 Query Execution

During query execution, the Query Services handle data flow between local and remote pipelines. Data tuples received from the network are dispatched to the associated pipeline. For this purpose, the query and stage information provided in the query data message is used to lookup the appropriate context in the Query Registry. At the end of each pipeline, routing components determine the destination of the output data, which is encapsulated into a query data or query

result message; tuples that require further processing are encapsulated in query data messages and routed to a target pipeline, while query result messages are used to propagate a finalized tuple to the query root without further processing.

3.6 Network Services

Network services provides an abstraction layer supporting inter-peer communication. For the prototype implementation, the Network Services hold information regarding the complete network membership, modeling a one-hop overlay. The actual logic for query and data routing, which depends on the distribution strategy, is implemented by the Query Services, while Network Services supports membership query and message forwarding. Regarding message reception, it allows services to register message handlers for specific message types and handles delivery accordingly.

In RTree and NTree, the Network Services assists query distribution and control by building a dissemination tree, given a spatial range (the area of interest of a query) and a root node. The process used to build this tree is a simplification, based on the assumption that all peers have a consistent view of the network. Peers are initially filtered by matching the node *filter* to the spatial range. In both RTree and NTree, this filter is implemented as a circular area, centered on the node's geographical coordinates, with a fixed radius that is established by configuration. For NTree this is a simplification of the proposed approximation of the Voronoi cell that does not depend on the network topology.

Conceptually, peers are arranged in a circular structure ordered by an identifier. Starting from the query root, each node is assigned the next n peers in the circular order that haven't been assigned to a preceding node - this is illustrated in Figure 3.3 for a branching factor of 2.

Figure 3.3: Query Dissemination tree in RTree and NTree

Assuming a consistent membership view, this process can be replicated in each peer resulting in the same distribution tree. Although this assumption is plausible in a simulation setting, a consistent membership view does not always hold in a real-world implementation, given the network dynamism. A realistic implementation could be supported by a tailored peer-to-peer content-based routing, or geographic routing substrate.

A KD-Tree data structure, holding peer geographic coordinates, is used to support membership queries[2]. Range queries are used in QTree to obtain the node set for a particular quadrant;

[2]The prototype uses a KD-Tree implementation in Java developed by Simon D. Levy [26]

this information is used to enable query distribution and data acquisition binding. NTree uses nearest neighbor queries to find the destination for a give data tuple.

3.7 Acquisition Services

In the prototype implementation, mobile nodes run black box application clients that acquire data and deliver to the fixed infrastructure. Mobile nodes do not connect directly to acquiring nodes, instead data is forwarded to a homebase, which acts as a proxy to the destination node. The Acquisition Services hosted on homebase nodes are responsible for binding and data delivery. Binding to a target acquisition node is a function of the particular distribution strategy and, for that purpose, the Query Services are used to determine the possible targets for a particular data tuple, based on its geographic coordinates. Given the one-hop overlay model, this process is simplified, since a homebase holds the information required to determine the possible target nodes. The validity of the current binding is verified for each acquired data tuple; an update occurs when the acquiring node is no longer part of the target list for the last sample received.

Regarding data consumption, Acquisition Services are used by acquiring nodes to register for particular sensor inputs when data source pipelines are instantiated. The sensor definition concept, allowing applications to configure specific acquisition requirements, has been left out of the prototype implementation. Instead applications name a sensor type directly in the *sensorInput* declaration. Presently, only online acquisition of GPS data is supported; the sampling modality (periodic or event driven) and rate are dictated by the application clients running on mobile nodes.

4. Case Study and Experimental Evaluation

This chapter presents the evaluation of the 4Sensing participatory sensing framework, supported by the SpeedSense case-study application. Section 4.1 discusses the SpeedSense application, while the simulation setting used for the evaluation work is presented in Section 4.2. Finally Section 4.3 presents the comparative evaluation of the three distribution strategies.

4.1 SpeedSense Application

SpeedSense continuously monitors the current traffic status in an urban setting, allowing client applications to access the current traffic speed per road and information about congested roads. For this purpose, users collect real-time data while driving, using GPS equipped mobile devices. Possible usage scenarios for SpeedSense data include the visualization of the current traffic conditions and route planning.

SpeedSense does not intend to be a realistic implementation of road traffic estimation. The application uses a simplified model to infer traffic conditions that does not consider the full complexity of the problem, including the effect of location error and road features, such as traffic signals and intersections. The goal of this case study is to provide a benchmark application to support a comparative analysis of the proposed distribution models, and other aspects of the participatory sensing framework.

Two virtual tables have been implemented: TrafficSpeed - described in Section 4.1.3 - supports querying for average speed per segment, while TraficHotspots - described in Section 4.1.4 supports querying for congestion detections.

4.1.1 Road Network Model

SpeedSense requires a map representation of the application's geographic area of coverage, providing the common reference data that is shared among the fixed infrastructure nodes and application clients. SpeedSense uses the Open Street Map (OSM) [43] vectorial representation of the road network [1]. This map data is used to map geographic coordinates to road segments and to determine the spatial extent, and associated road type, of segments. Segmentation is specially relevant for large roads, such as highways, that can extend for several kilometers. Although the extent between two road intersections can be used for this purpose, the size of these segments is not homogeneous, being significantly larger in highways than other road types. For this reason, roads are divided into 1 km extents, resulting in one segment for each driving direction. OSM classifies roads into types, such as highways, primary to tertiary and residential (see Figure 4.1 for a rendering of the Lisbon urban area). This classification is used

[1] The OSM project provides free maps created collaboratively by the user community. OSM maps are edited, in a similar spirit to Wikipedia, using base data gathered from GPS devices or satellite images. SpeedSense also uses part of the codebase of the JOSM [28] editor for parsing, querying and rendering OMS data.

to assist the detection of congested roads, by assigning an expected driving speed to each road type.

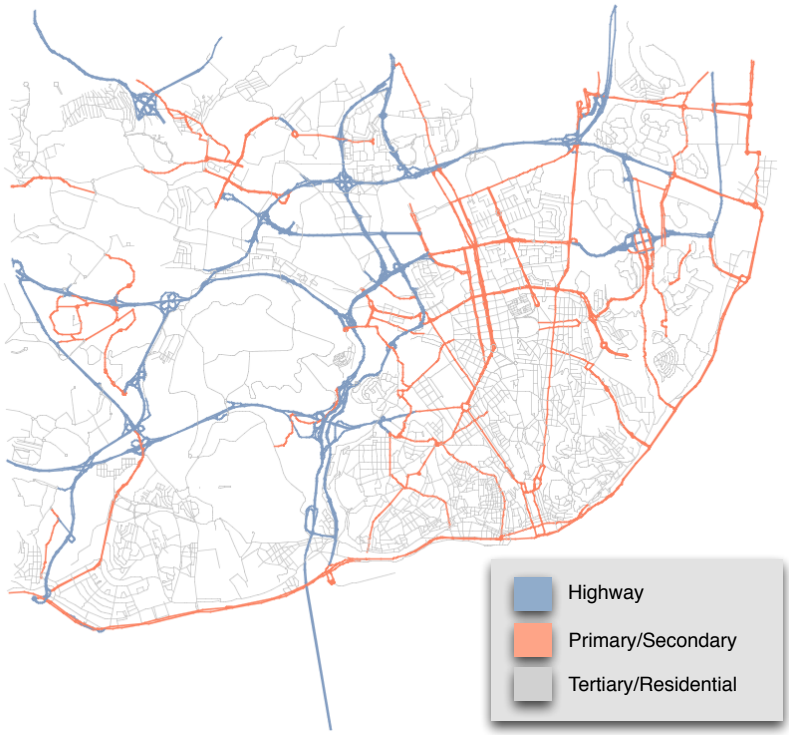


Figure 4.1: Lisbon map rendering from OSM data

4.1.2 Data Acquisition and Mapping

GPS readings are produced periodically by mobile nodes (every 5 seconds in the simulation setup) and delivered in real-time to the fixed infrastructure. GPS samples convey the vehicle's location (given by a latitude/longitude pair), its current speed and a timestamp. For simulation purposes, the segment identifier is also provided by the mobile node as part of the raw GPS samples. This is a simplification that leverages the fact that the OSM map model is also used for traffic simulation. Consequently, data mapping is 100% accurate. SpeedSense expects that sampled data reaches the acquiring nodes with small delay variations compared with the size of the time windows used.

4.1.3 TrafficSpeed

TrafficSpeed is used to query the average speed for road segments enclosed by the area of interest. This table derives directly from the *GPSReading* sensors input and produces an output

stream of *AggregateSpeed* tuples that convey a segment, sample count, and total and average speed. Average speed is computed using a time-window to break down the continuous acquisition stream into finite time intervals. 4.1 shows the VTDL for this table.

Listing 4.1 TrafficSpeed virtual table specification

```
sensorInput( GPSReading )
dataSource {
    process{ GPSReading r ->
        r.derive( MappedSpeed, [boundingBox: model.getSegmentExtent(r.segmentId) ])
    }
    timeWindow(size:15, slide:10)
    groupBy(['segmentId']){
        aggregate( AggregateSpeed ) { MappedSpeed m ->
            sum(m, 'speed', 'sumSpeed')
            count(m, 'count')
} } }
globalAggregation {
    timeWindow(size:10, slide:10)
    groupBy(['segmentId']){
        aggregate( AggregateSpeed ) { AggregateSpeed a ->
            avg(a, 'sumSpeed', 'count', 'avgSpeed')
} } }
```

The data sourcing stage handles data mapping and local data aggregation, using the following stream operator sequence:

process Maps raw GPS data to a road segments. For the simulation scenario, the segment identifier is present in the raw GPS reading; the process operator simply assigns the spatial extent of the segment to a *MappedSpeed* tuple, derived from *GPSReading*, that holds a segment identifier and a speed attribute.

timeWindow Breaks down the continuous *MappedSpeed* stream into discrete window extents. The simulation setup usess a 15 seconds windows with 10 seconds slide.

groupBy Partitions the mapped data stream, creating an independent stream for each road segment.

aggregate Outputs an *AggregateSpeed* for each window extent, with the sample count and total speed - the partial state used to compute the speed average.

Global aggregation receives aggregate speed data from child pipelines and produces the overall average speeds by merging the partial records.

timeWindow Accumulates the *AggregateSpeed* tuples from child pipelines. Since all pipelines output results at most every 10 seconds, this window will contain the most recent partial aggregate (for each segment) from each child pipeline.

groupBy Partitions the stream of *AggregateSpeed* tuples by road segment.

aggregate For each partition, outputs an *AggregateSpeed* per window extent, conveying the average speed per segment (and the aggregate sum and count) computed from the partial sum and count values received from each child pipeline.

4.1.4 TrafficHotspots

The *TrafficHotspots* table is used to query for road segments enclosed by the area of interest where congestion has been detected. Congestion detection is based on the current average speed, the number of samples - as an indicator of the relevance of the average - and a threshold given as a fraction of the maximum speed for a particular road. The following formula is used in the simulation setup to determine if a segment is congested:

$$(avg_{speed} \leq SPEED\_THRESHOLD \times max_{speed}) \wedge (n_{samples} > COUNT\_THRESHOLD)$$

Roads are classified into types (highway, primary, etc.) that are used to assign maximum speed values. Detections are assigned a confidence value; for simulation this value grows linearly with the number of samples according to the following formula:

$$confidence = min(1, \frac{n_{samples}}{COUNT\_THRESHOLD} \times 0.5)$$

The TrafficHotspots table outputs a stream of *Hotspot* tuples representing real-time detections of congested road segments. Definition 4.2 shows the VTDL for this table which derives from *TrafficSpeed* and extends its global aggregation stage with the hotspot detection classifier. The classifier receives an aggregate speed stream and produces a *Hotspot* tuple if the computed average is complete, reliable and below the congestion threshold.

4.2 Simulation

The SpeedSense application is implemented in a simulation environment. This environment comprises a single computer running a simulator that emulates the execution of multiple nodes on a single Java virtual machine, providing inter-node communication and task scheduling services.

Listing 4.2 TrafficHotspots virtual table specification

```
tableInput("speedsense.TrafficSpeed")
globalAggregation {
    classify( AggregateSpeed ) { AggregateSpeed a ->
        if(a.count > COUNT_THRESHOLD && a.avgSpeed <= SPEED_THRESHOLD * model.maxSpeed(a.segmentId))
            a.derive( Hotspot, [confidence: Math.min(1, a.count/COUNT_THRESHOLD*0.5) ])
} }
```

4.2.1 Traffic Model

The same OSM map model used for the SpeedSense application implementation is used for the simulation of traffic patterns in the Lisbon urban area. Traffic is modeled by emulating a fleet of vehicles driving through random routes. An average speed, for each road segment at a given time, is determined by its car density (averaged over a 10 second period), according to the following formula:

$$\frac{max_{speed}(segment)}{1+\frac{avg_{density}(segment)}{5}}$$

The maximum speed for a given segment is the same value used for congestion detection and depends on the road type; given the formula used to detect congested segments, congestion occurs is segments with a density of at least 5 vehicles.

This average speed per segment is used to obtain the random speed individually for each vehicle, according to a normal distribution. Vehicle paths are determined by choosing a random start position and sequence of road intersections. In order to induce traffic confluence, higher probability is given to highways and primary roads; a new path is computed whenever a vehicle reaches its destination. Figure 4.2 shows a rendering of the traffic simulation.

4.2.2 Infrastructure

The simulation comprises a fixed and mobile node infrastructure. Communication among fixed nodes is supported by the simulator network layer that defines an addressing scheme and exposes communication primitives to send and receive messages. Fixed node are distributed randomly across the urban space with a minimum inter-node distance of 250 meters (see Figure 4.2). A one-hop overlay network connecting the fixed infrastructure is simulated through a common peer database that is shared among all nodes - consequently, nodes share a consistent view of the network membership. The network is static i.e., membership is determined before the simulation and there are no node entries or exits during execution.

Mobile nodes do not have a network address and reference their homebase counterpart directly - the link between mobile nodes and acquiring nodes is simulated, resulting in the delivery

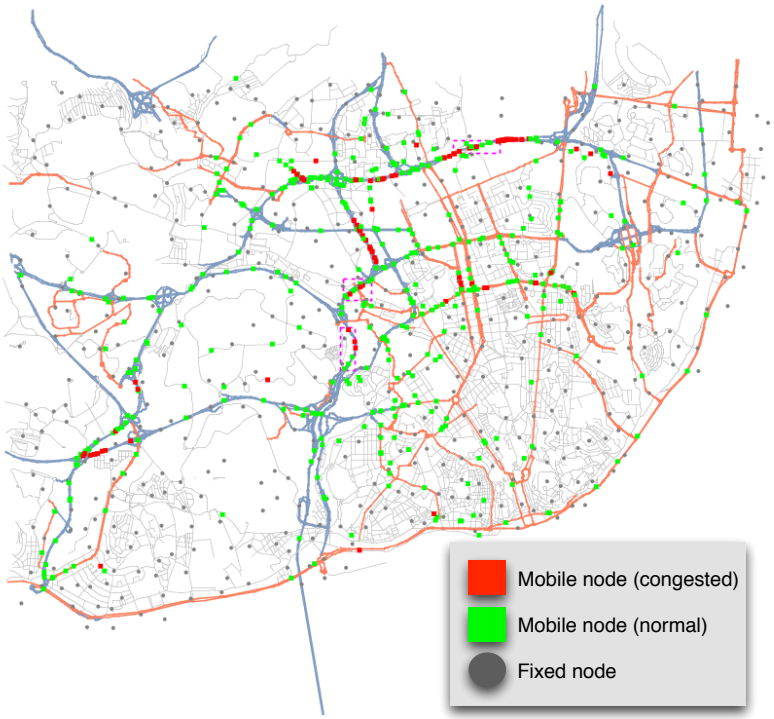


Figure 4.2: Traffic simulation

of raw data with no latency. Each mobile node simulates a vehicle, according to the presented traffic model, and reports its GPS reading every 5 seconds as it follows the assigned path.

4.3 Experimental Evaluation

Simulations have been run for two different node densities; in a low density scenario, the Lisbon urban area is served by 50 fixed nodes, while in the high density scenario, the same area is served by 500 nodes. In both cases, the mobile infrastructure comprises 500 mobile nodes. In RTree, mobile nodes are evenly distributed among the fixed counterparts.

Two queries - Q1 and Q2 - have been tested for each distribution strategy and node density, where Q1 covers 1/16th of the overall simulation area and Q2 covers four times this area (25% of the total simulation area). The two queries are placed on a high mobile node density area.

A set of metrics, presented in the following sections, have been captured for each simulation scenario and averaged over 10 runs of 15 minutes of simulated time each. Each run of the simulator resets the distribution of fixed nodes, consequently the metrics represent the average result expected for running the same query with different node distributions. The first three sections discuss metrics that capture how the work required to evaluate a query is distributed among the

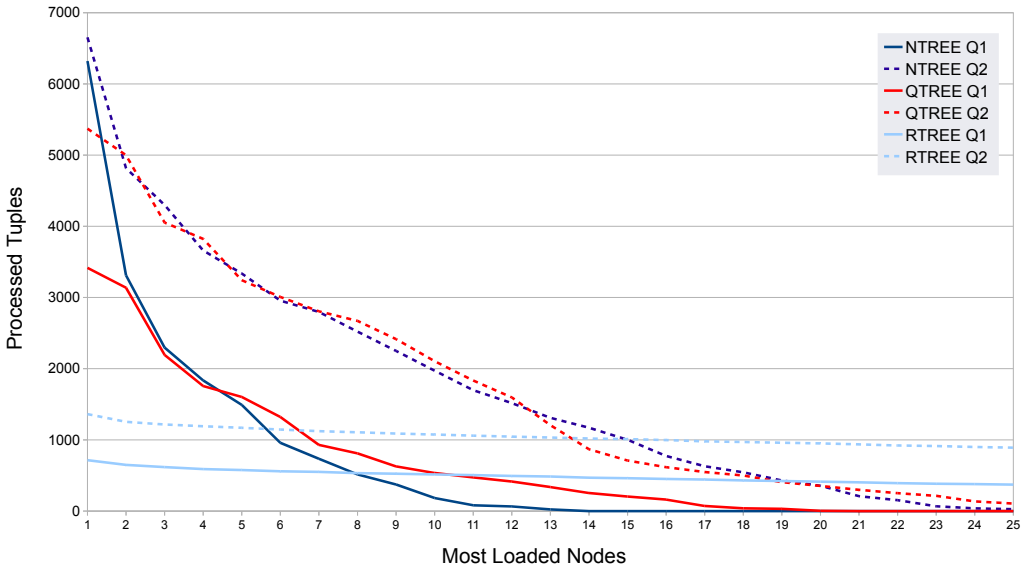


Figure 4.3: Acquisition Workload - 50 Nodes (Q1 and Q2)

participating fixed nodes. Work has been broken down into data acquisition and aggregation. The resulting metrics (discussed in sections 4.3.1 and 4.3.2) measure the corresponding events occurring in each fixed node, while an a total workload metric (discussed in 4.3.3) reflects the combined acquisition and aggregation work. Section 4.3.4 evaluates the latency and accuracy of query results and Section 4.3.5 discusses the network usage measured for each strategy. Finally, Section 4.3.6 summarizes the main characteristics of the three distribution strategies.

4.3.1 Acquisition Workload

The acquisition workload measures the number of GPS sensor readings received by each acquiring node - which is equivalent to the number of tuples processed by the data sourcing pipeline hosted in each node. Figure 4.3 shows how the query area affects acquisition workload in the low density scenario, while Figure 4.4 show the acquisition workload for low and high density scenarios (first 25 nodes). Both graphs plot absolute values from the most loaded to less loaded node, averaging the values obtained from all simulation runs.

RTree is effective in balancing the acquisition load evenly. A wider query area, or a lower node density, results in a higher load per node but, in all cases, the load is well distributed.

In both NTree and QTree - where acquisition binding is based on geographic criteria - the distribution reflects the data skew inherent to the application domain, given that some areas have significant higher mobile node density then others. Although having different binding strategies, the resulting distributions are similar. A lower node density degrades significantly the balance of the distribution, as can be infered from Figure 4.4. This is specially the case in

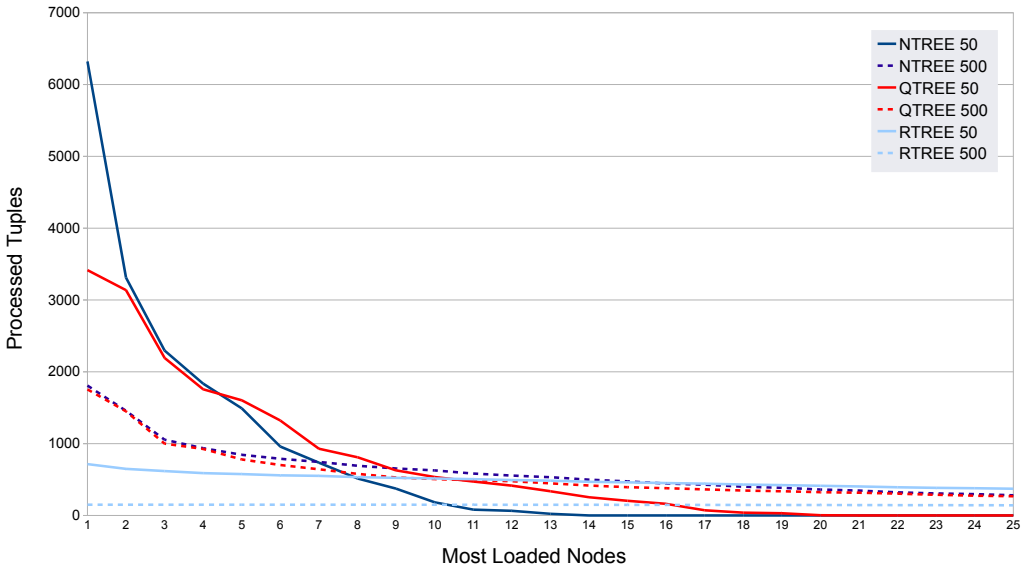


Figure 4.4: Acquisition Workload - Q1 (50 and 500 nodes)

NTree, since QTree guarantees that any geographic point is served by a number of acquisition nodes given the minimum occupancy (2 nodes in the simulation scenario) - this is evident in Figure 4.4, where the top load in QTree for Q1 is about half that of NTree.

Overall, RTree balances load more effectively than NTree and QTree with the downside of affecting more fixed nodes. NTree and QTree acquisition strategies result in skewed load distributions, specially for a lower node density.

4.3.2 Aggregation Load

The aggregation workload refers to the number of inputs received by the global aggregation stage in each node. This value corresponds to the sum of the number of updates received for each segment aggregated by the node. Thus, the load distribution depends on how well the strategy balances the responsibility over road segments, and the amount of updates per segment. Figure 4.5 shows the effect of the query area in the acquisition workload, in the low density scenario. Figure 4.6 shows the aggregation workload for low and high density scenarios.

RTree clearly stands out in terms of total aggregation work and distribution, specially for a wider query. This is essentially given to the fact that the responsibility over segments grows cumulatively towards the root. Consequently, although all fixed nodes in the simulation participate in aggregation tree, the upper level nodes handle a significant share of the load. Ultimately, the query root has to aggregate over all the segments covered by the query area. This cumulative effect is the main drawback in RTree and results in a significantly higher total work than other strategies. Besides, it results in RTree having a high sensitivity to the area of the query, as can

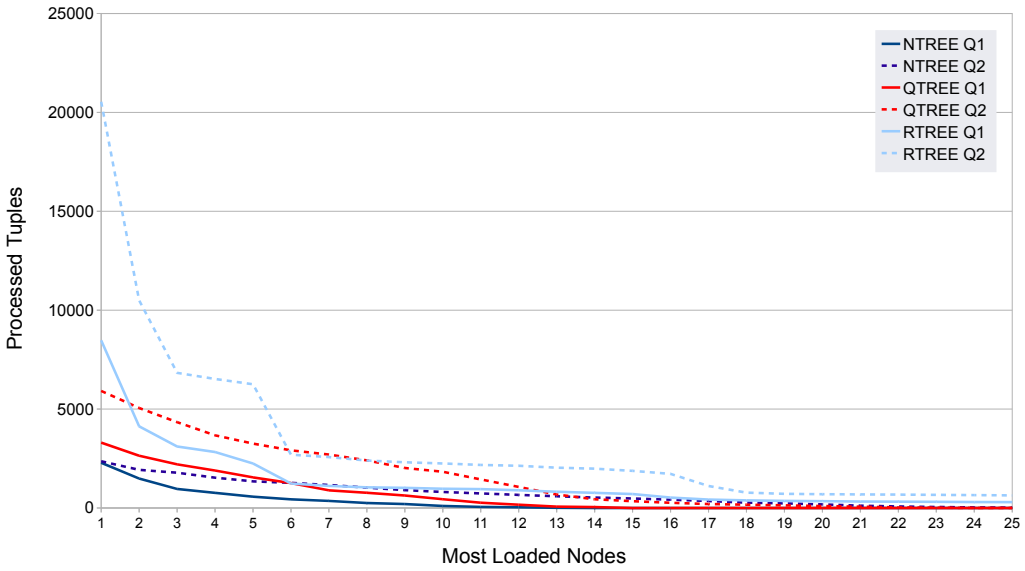


Figure 4.5: Aggregation Workload - 50 Nodes (Q1 and Q2)

be seen in Figure 4.5, while a higher number of nodes does not result in better balancing.

Another characteristic of RTree contributes for the higher workload: the number of updates, produced by data sourcing pipelines for a segment per window period, depends on the number of mobile nodes that cross that segment. In contrast, NTree and QTree produce a limited number of updates for every window period for a give segment, where the limit depends on the node density and the segment geography - in NTree it is given by the number of nodes that are the closest to any point of the segment; in QTree it is given by the number of nodes enclosed by the maximum division quadrants crossed by the segment. Using a time-window in the aggregation stage reduces the number of updates by aggregating data from upstream nodes, still, this reduction is limited by the fact that information for a given segment is spread throughout the aggregation tree.

QTree distributes load more effectively than RTree due to the fact that each node in the aggregation tree is responsible for a limited geographic area and, although this area grows towards the root (each level encloses the areas of its descendants), the cumulative effect is reduced by the fact that a node at a given level aggregates only those segments that are not fully enclosed in a lower level. In this case the load will not be necessarily higher in the root. Contrary to RTree, load distribution in QTree is significantly more sensitive to the node density, balancing load more effectively with high node densities.

In the high density setting, NTree shows a balanced aggregation distribution since the nearest neighbor approach results in a balanced split of segment responsibilities. NTree is also characterized by having lower total aggregation work; this is due to the fact that only one aggregation level is used for the evaluation scenario - updates do not have to be aggregated at

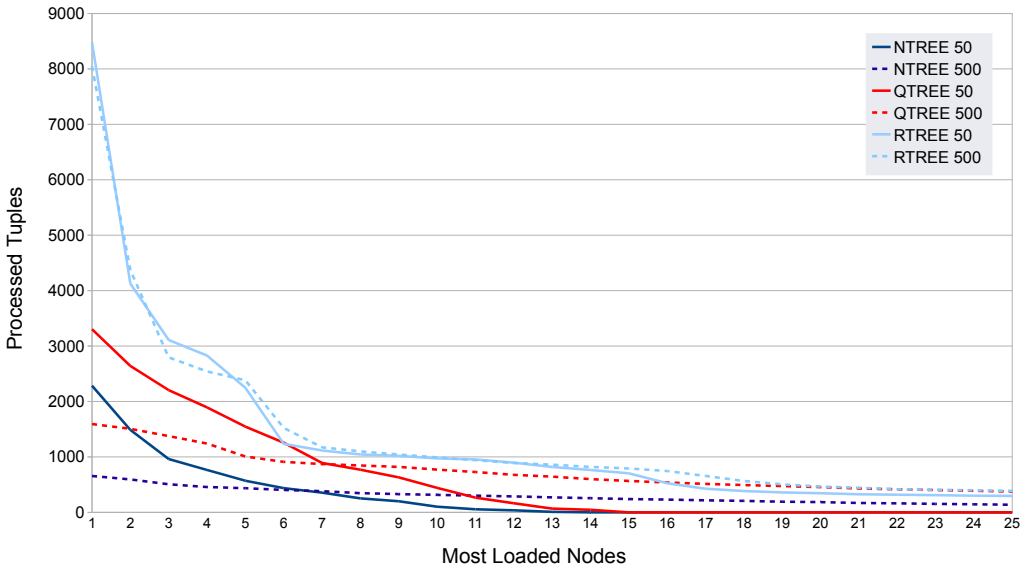


Figure 4.6: Aggregation Workload - Q1 (50 and 500 nodes)

multiple levels. NTree is also less affected by the query area, while (as in QTree) a lower node density degrades the distribution significantly.

In a high density scenario, NTree has the best aggregation performance, considering both load balancing and total work. RTree shows higher total work and a skewed distribution that reflects its tree structure - where load increments in direction to the root. QTree stands between the two - although it uses also a multi-level aggregation tree it is able to reduce effectively the propagation of updates towards the root. With lower node density, QTree and NTree degrade significantly but still show better results than RTree, for which the levels closer to the root are heavily penalized, although the distribution is skewed in all cases. Structured strategies based on geography are more efficient for aggregation in the studied application, although work balance with lower node densities is similar to the one observed in an unstructured approach, the overall aggregation load is effectively reduced.

4.3.3 Total Workload

Total workload refers to the overall fixed node input, including both raw data originating from mobile nodes and aggregation data produced by peers, and is obtained by adding the acquisition and aggregation loads with equal weights in each node. Figure 4.7 shows the effect of the query area in the total workload, in the low density scenario. Figure 4.8 show the total workload for low and high density scenarios.

Total load in RTree is dominated by the aggregation effort, and thus shows the same characteristics as the aggregation load - a high load skew and total work that grows significantly with

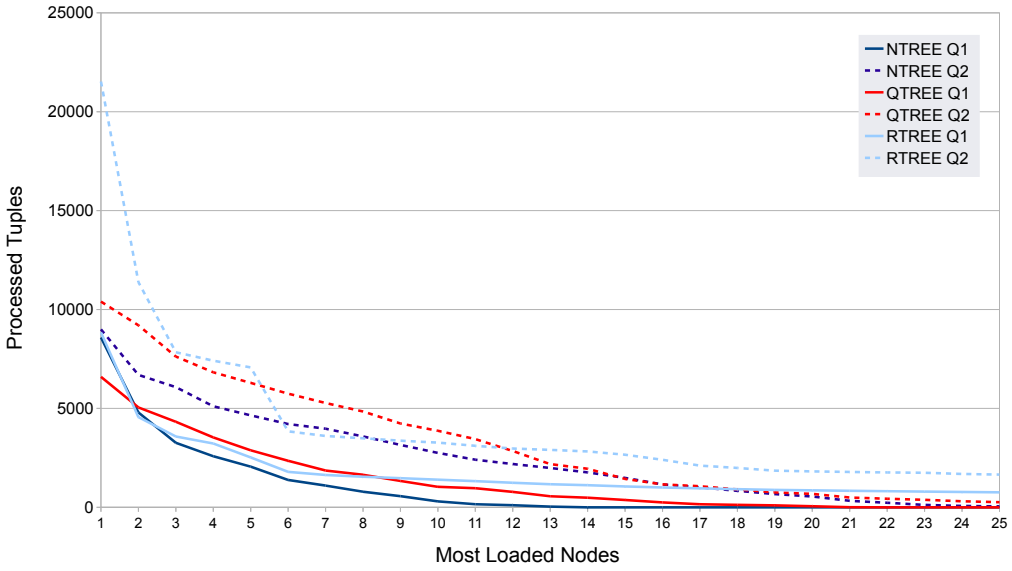


Figure 4.7: Total Workload - 50 Nodes (Q1 and Q2)

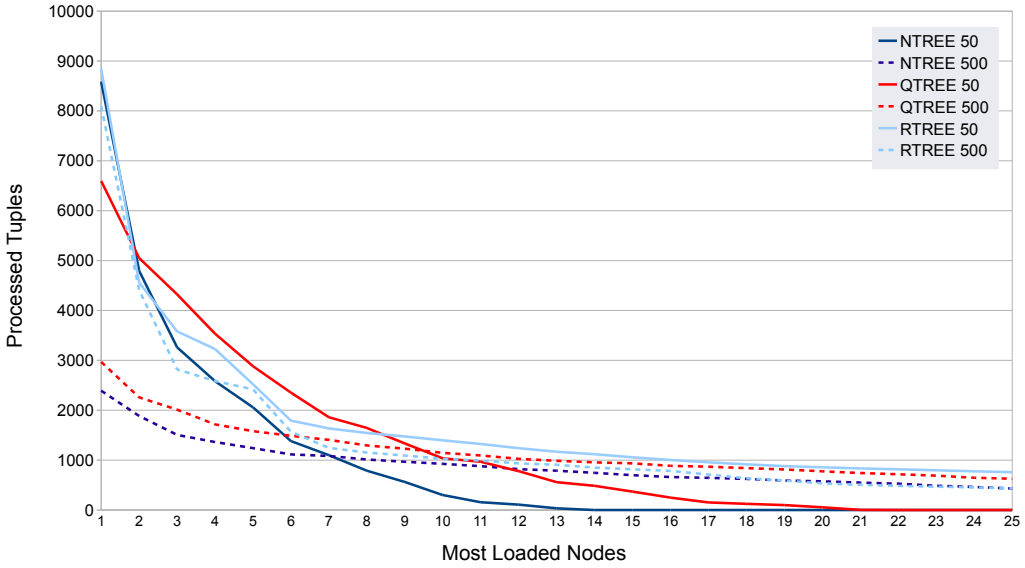


Figure 4.8: Total Workload - Q1 (50 and 500 nodes)

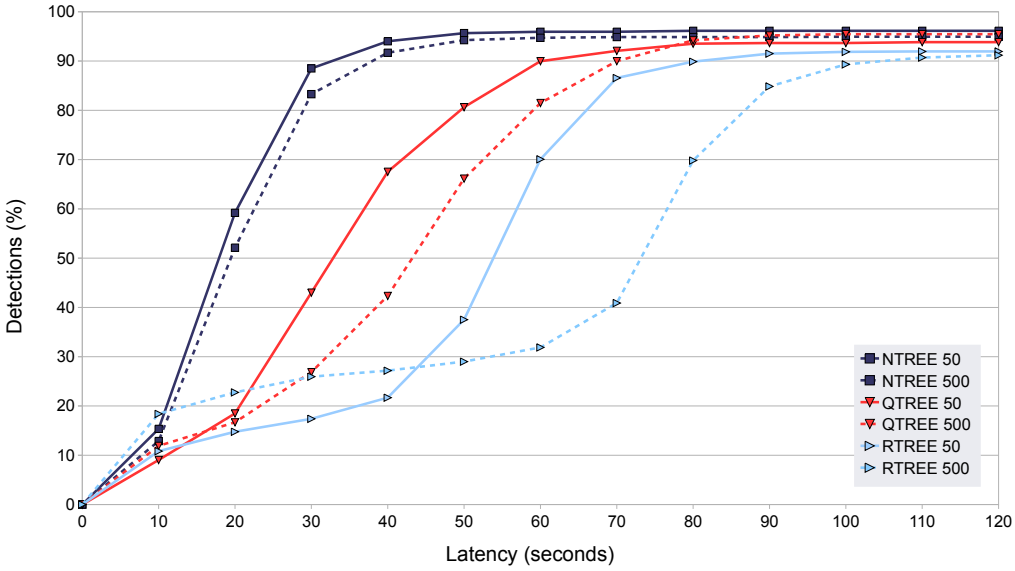


Figure 4.9: Detection Latency

the query area, and low sensitivity to the node density.

In the high density scenario, both QTree and NTree show a good total work distribution, relative to RTree, but with low density both suffer from the high skew in the geographic distribution of mobile nodes.

Overall, NTree shows the best results for high density, followed by QTree, while RTree is highly penalized by the aggregation effort. For low densities, the distance to RTree is smaller - NTree and QTree actually show a higher skew for Q1 than RTree, but both limit the total work more effectively in wider queries.

4.3.4 Query Success and Latency

Query success is given by the percentage of accurate detections, including both false negatives and false positives - where a false negative occurs when no detection is received within 120 seconds of occurrence. Detection latency times the lag between the occurrence of a segment congestion and the arrival of the respective detection at the query root. Transient congestions (lasting less than 20 seconds) are not considered for the evaluation. It should be noted that the traffic patterns in the simulation are highly dynamic compared to the real world, with several short lived congestions occurring during query evaluation. Figure 4.9 plots the detection latency, where the x-axis refers to latency categories, in seconds (where, for instance, the category 60 groups all detections arriving in up-to 60 seconds from the congestion occurrence), and the y-axis represents the percentage of detections.

RTree shows an average detection latency of 57 seconds for the high density setting and 47

seconds for low density, which is justified by the different depths of the aggregation tree (6 and 4 levels respectively). The wide range of latency values, with some detections arriving in 20 seconds and other arriving in 90 seconds reflects the structure of the aggregation tree where data sources are connected at different tree levels. This fact possibly contributes to detection error, when using an aggregation time window. Aggregation averages data sourced at different levels, and thus sampled in different moments in time; for more transient phenomena this can result in higher error, which is visible in RTree results where the overall detection accuracy is around 91%.

Average latency in QTree (33 and 41 seconds for high low and high densities, respectively) also reflects the aggregation tree depth. Data sources, in this case, are connected at the tree leaves; although not all tree branches have the same depth, the discrepancy in time of averaged samples, and its contribution to error, is lower.

Detection latency in NTree is lower, reflecting the fact that only one aggregation level is used. There is a small difference in latency between the low and high density scenarios (18.7 and 20.3 seconds respectively) which can result from the fact that more fixed nodes tend to result in the separation of data sources and global aggregation pipelines, for a given segment, into different nodes.

NTree shows the best overall results. It has to be taken into account that the case study application does not use multi-level aggregation, which would rise the latency values. RTree lags in terms of detection accuracy, due to the particular structure of the aggregation tree. Removing the time window in the aggregation phase - and using the *set* operator instead - solves this issue but reduces scalability with the growth of the mobile node base. Finally QTree performance lies between the other two strategies, showing good accuracy but higher detection latency that NTree.

4.3.5 Communication Load

The communication load measures the amount of messages exchanged during the execution of a query, providing an indication of the efficiency of the different distribution strategies in terms of network usage. Communication load includes data messages and binding events. The former accounts for tuples exchanged between peers, relative to query data (incomplete aggregations that are forwarded up the aggregation tree) and query results (complete aggregations that are forwarded to the query root, not applicable to RTree), not including raw data messages from mobile nodes. Query data messages are forwarded according to the aggregation time-window periodicity, and group the tuple set produced for each window extent. Binding events capture the overhead introduced in NTree and QTree by the separation of the homebase and acquiring roles. Although in the prototype, all sensor data is proxied through the homebase, this is a simplification of the proposed model, where mobile nodes bind to an acquisition node according to their geographical location and send acquired data directly. Still, this process represents a network overhead since at least one message has to be forwarded when a node switches between acquisition nodes. Figure 4.10 shows the average number of messages sent for each simulation

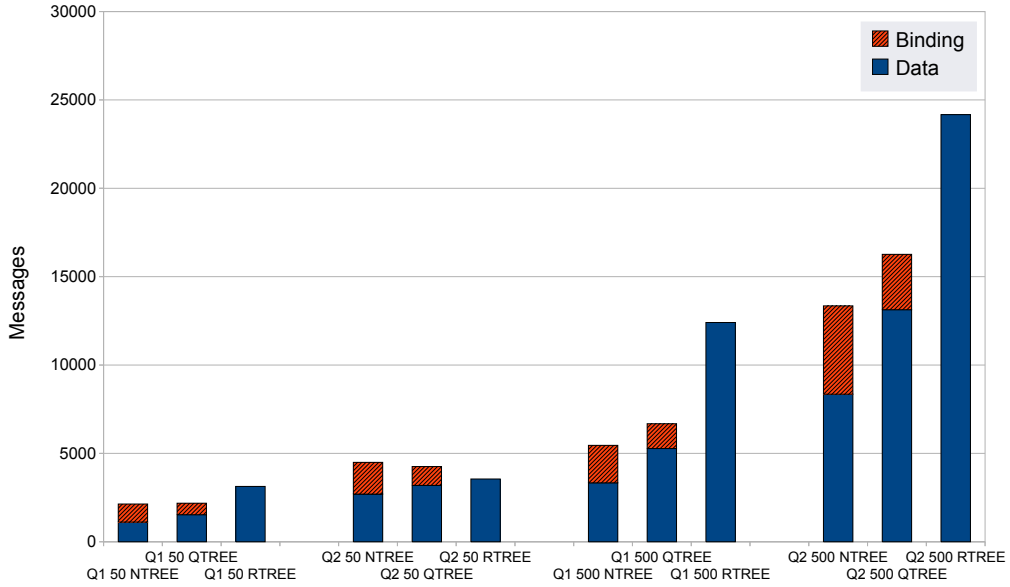


Figure 4.10: Communication Load

scenario and distribution strategy, for both query data and binding events.

RTree does not scale efficiently with the growth of the fixed node base, reflecting the fact that more nodes result in data being more spread throughout the network, resulting in a higher communication effort necessary to aggregate data. Running the same query in low and high density settings represents a growth of around 300% for Q1 and almost 580% times for Q2. In a low density scenario, the number of messages from Q1 to Q2 grows 13.5%, while with higher density, the number of messages grows 95%.

In QTree, executing the same query in low and high density settings result in a significant growth in number of messages of around 200% for Q1 and 280% for Q2, but not so accentuate as in RTree. The impact of the query area is high in both low and high density settings, resulting in a growth of 95% for low density and around 140% for high density.

Finally, in NTree, changing from low to high density results in a growth of around 150% and 197%. The impact of the query area results in a growth of around 110% (for low density) and 145% (for high density) .

NTree shows the best performance in most cases, in terms of the absolute number of messages exchanged. It also shows a higher resilience to the growth of node density and the consequent data dispersion, given that it requires only one hop to group all related data. RTree is less sensitive to the query area, specially in low node densities, but the total number of messages is generally higher. Overall, the three strategies show comparable performances in the low density scenario, with RTree showing an advantage in Q2, due to the binding overhead introduced by NTree and QTree, but RTree falls behind in a higher density setting, due to its unstructured approach to aggregation and the fact that it affects the entire fixed node base. QTree performance

is close to NTree; although it shows a lower binding overhead, in high density settings this does not compensate for the higher data traffic.

4.3.6 Summary

RTree produces the best acquisition balancing compared to the other strategies, with any node density; although sensors are unequally distributed in space, since there is no relation between the geographic location of the mobile node and the data destination, data is equally dispersed. This property becomes a weakness in the aggregation stage, given that related data is spread throughout the network and has to be forwarded towards the query root, resulting in a heavy load on the higher level nodes. A higher number of nodes degrades the RTree efficiency, since data is more spread, resulting in more communication load without changing the skew towards the root. RTree requires an adequate number of fixed nodes that is sufficient to offload acquisition work but not too high that communication costs start to dominate. Due to data sourcing occurring at different tree levels, RTree has a higher error rate which can impact scenarios where the monitored variables are too dynamic for the latency introduced by aggregation windows.

QTree and NTree have similar characteristics regarding both data acquisition and aggregation. Both are affected by the unbalanced distribution of mobile nodes, specially for lower node densities. The strength of the two strategies resides in the ability to gather related data closer to the data sources - the less hops data has to traverse in the aggregation path, the less overall work. NTree requires at most one hope from data source to complete aggregation, while in QTree the number of hops depends on the geography of each particular spatial extent. This characteristic gives an advantage to NTree in both total workload and communication costs.

Overall, for low densities and small queries the three approaches have comparable performance. For, higher node densities and wider queries, RTree degrades significantly and NTree is more effective in limiting the total work.

A higher node density has significantly different impact in unstructured and structured approaches: in RTree, more nodes result in data being more dispersed, requiring more communication effort to gather related data. In NTree and QTree, more nodes allow better balancing by distributing more evenly the responsibility over spatial extents.

Regarding communication load, it should be noted that the performance difference in RTree, using higher node densities, is accentuated for queries targeting the TrafficHotspots table. The same evaluation over TrafficSpeed would shorten the distance between all strategies, since the average for every segment within the query area would have to be forwarded to the root.

5 . Related Work

This chapter presents an overview of previous work related to data management in *participatory sensing* applications. The relevant body of work spans several areas of research besides *participatory sensing*, including wireless sensor networks, internet scale sensor networks, ubiquitous computing, overlay networks and spatial queries.

The first section presents an overview of the main issues regarding data management and persistence in *participatory sensing* applications. Section 5.2 discusses representative works that focus on one or more of the aspects covered on the overview.

5.1 Data Management in Participatory Sensing

5.1.1 Infrastructure

Several works in the participatory sensing domain propose a centralized infrastructure (e.g., [24], [11] and [38]), where the mobile node base is served by a central back-end that handles data storage, processing and querying. This centralized infrastructure suits the particular goals pursued in these works, focused on data acquisition and delivery [24], and application specific models [11, 38] and has the advantage of simplifying all aspects of data management. However, centralized solutions have relevant drawbacks, such as the implications of having privacy sensitive data, for the entire user base, managed by a single administrative entity. Additionally, the high data volume that participatory sensing applications may generate can imply excessive infrastructural costs, making this approach a conceptual mismatch with community oriented efforts. The infrastructure for these people-centric projects should be fully supported by the community of users since no economical or institutional backing can be assumed. A peer-to-peer architectural model, where interested parties provide resources according to a system of incentives, is more adjusted to this domain of grassroots applications.

One approach to deal with decentralization is to rely entirely in the mobile nodes and ad-hoc coordination to support applications. This is the approach explored in the mobile middleware solutions presented in [45] and [30]. A limitation of this approach is that mobile nodes have to spend computation, communication and energy resources in coordination efforts, regarding node and service discovery and context dissemination. This is specially relevant given that communication can be expensive, energy is a scarce resource and the middleware should have a minimal impact on the primary phone functions.

An alternative approach, also followed by this dissertation, is to integrate an hybrid architecture, composed of mobile and fixed components (e.g.,[12] and [33]). Hybrid infrastructures can serve different purposes, such as the integration of heterogeneous sensor networks and the offload of the most resource intensive functions to fixed components. Fixed nodes act as proxies, receiving and processing data from devices, hiding device mobility from the network, and coordinating with other nodes to service data requests. In this hybrid architecture, mobile nodes

interact with a limited number of fixed nodes and are abstracted from coordination efforts. In the Partisans architecture [12], this proxy role is assumed by the *mediators*. A mediator is a fixed node, geographically close to sensors, that provides a set of in-networks functions over sensor data streams, such as enhancing data with verified context information, data validation, anonymization and stream replication to serve multiple clients. Partisans relies on a structured registry, similar to the DNS, to facilitate discovery of sensor data by *subscibers*. In the 4Sensing architecture, data discovery is supported by an overlay network model used to direct queries to relevant nodes, according to geographic criteria.

5.1.2 Heterogeneous Sensors

In order to support a large domain of *participatory sensing* applications, a data management model has to take into account the heterogeneous nature of the collected data. From streams of scalar sensor readings, to multimedia content, an appropriate level of flexibility has to be built into the underlaying platform. A distinctive characteristic of participatory sensing application is that the mobile sensor base is necessarily heterogeneous, in terms of both hardware and software, and changes with time as new devices are introduced. This represents a challenge to application developers that has to be addressed by the supporting middleware.

In CarTel [24], sensor *adaptors* abstract the details of specific sensors from the data acquisition system. Meta-data conveyed in an *adaptor* is mapped to a relational schema that is then used to process queries submitted by applications. Application data requirements can thus be abstracted from the details of interfacing with specific sensors, and new sensors can be integrated simply by programming a new *adaptor*.

In SensorMap [40], the purpose built Sensor Description Markup Language (SGML) is used to describe sensors interfaces in terms of attributes such as sensing type, data type and scalar data units. The authors advocate the creation of standard ontologies for sensor description as the basis for interoperability.

In this dissertation, applications express acquisition requirements using sensor definitions. Matching a sensor definition to a suitable sensor is left open and could be facilitated by the approaches described above. Adaptors abstract the acquisition system from the low-level sensor integration, while descriptive metadata can assist the selection of matching sensors for a given sensor definition. Data in 4Sensing, referenced through sensor definitions and virtual tables, is represented as tuples, thus abstracting from the heterogeneous representations produced by particular sensors.

5.1.3 Data Placement

A successful implementation of a distributed data management system has to deal with the, possibly conflicting, goals of promoting the fair sharing of computing resources among users, and guaranteeing efficient query computation, for instance by involving only relevant nodes and

grouping data according to some notion of locality. Regarding the first goal, the amount of computing resources contributed by a particular user, such as storage, bandwidth and processing, should be proportional to the perceived benefit he takes from the system, and each user should be able to select the applications they are willing to share his resources with. These aspects, characteristic of a peer-to-peer system, set the boundaries for the options regarding where data can be placed in the system.

Data placement, in this context, refers to the problem of deciding where to store persisted data in a distributed system in a way that enhances some characteristics, such as the fair sharing of resources, load balancing, resilience and query efficiency. In IrisNet [19], a hierarchical data schema partitions the application data space, and specific hosts in the network can be assigned to subsets of this hierarchy for data storage and query processing. The assignment of hosts to the nodes of the data hierarchy can be determined dynamically in order to balance the load, reduce network traffic and average query response time . Although the hierarchical partitioning provides a natural solution for data placement, not all classes of applications fit easily within this model.

Other approaches resort to structured peer-to-peer overlay networks, such as Distributed Hash Tables (DHT). DHTs have proven to be efficient for the retrieval of data objects by identifier, but a limitation of this approach is that data objects are opaque entities, typically characterized by name, unrelated to each other. Several efforts have recently been conducted to devise structured peer-to-peer architectures that support complex data types. These works focus on the issue of flexible querying based on multi-dimensional data attributes, instead of simple retrieval by object identifier. One approach is to map multi-dimensional data spaces into one-dimensional peer identifier space over a DHT, as in SCRAP [16]. Alternatively, the data space can be partitioned and the responsibility over partitions assigned to particular nodes, as in MURK [16]. Another approach is to create distributed versions of centralized spatial indexes - P2PRtree [39], detailed in section 5.2.6, is one example.

The data placement logic, in the proposed model, is embodied in the distribution strategy concept. The distribution strategy determines how data is distributed in the system, how it can be located, and how related data is gathered and summarized. The vast body of research in distributed databases, including the works described above, is a valuable input for the conception of suitable distribution strategies.

5.1.4 Social Domain and Data Granularity

An important aspect of urban sensing systems relates to the social scope of the intended application. Three basic domains are referred in the literature [4, 12]. Personal sensing applications focus on archiving and personal monitoring, for instance for health monitoring or fitness applications. Social sensing applications involve sharing among a specific community group or social network. In public sensing, data is open to the public at large. Different scopes can coexist in the same application, e.g., BikeNet [10] envisions the use of personal data, gathered from bicycle rides, to extract meaning relevant to the community, such as aggregated pollution maps,

available to the public.

In a broad spectrum of applications, particular data points are of low relevance to the desired output, and data aggregation is an effective tool to reduce data volume and preserve privacy. Data aggregation allows the summarization of data or extraction of relevant features by the application of aggregation operators - such as average, sum, maximum or histograms - over a time interval or geographic region. Aggregation is also necessary to allow client applications to perform efficient queries, for instance it would not be reasonable for a mobile application to fetch raw velocity readings in a area of interest to infer the traffic conditions.

In the personal sensing domain, aggregate computations might not be appropriate given the loss of detail they represent. In CarTel [24] the raw data sets (traces) are stored in a central database and users can view particular traces overlaid in a map. In this case, the loss of information represented by data aggregation might impact the relevance of the application to the user. But it can be questioned if this raw personal data should be stored in the common persistence system. Alternatively, a personal store can complement a common substrate and only summarized, or anonymized data (see Section 5.1.8) would be stored in the latter. In our system, the homebase role concept, hosted on a node owned and trusted by a particular user, can assume this personal store role and control the granularity of published data.

5.1.5 Querying

A querying model should provide a unified view of the data management system, hiding the complexity of its distributed nature. Different abstractions have been proposed, including declarative SQL queries (e.g., [35], [34] and [3]) publish/subscribe (e.g., [23]) and resource discovery (e.g., [12] and [40]).

Periodic, or continuous, queries in TinyDB [35], TAG [34] and CQL [3], extend the standard SQL syntax with a continuous query semantic. Queries, in these systems, are highly flexible and can specify predicates, for data selection, and data operations, for summarization and event detection.

In content-based publish/subscribe, consumers express data requirements by issuing a subscription, composed of attribute constraints over the data domain. Content production is decoupled from consumption, and the delivery of events to interested parties is handled by a content-based routing infrastructure. The adaptability to dynamic environments, and the ability to serve multiple clients with one publish operation, makes this approach a good option for mobile environments [23].

In the discovery model [12, 40], data sources register in a directory meta-information describing the data streams provided, including sensor and data types, and context information such as their location. Content consumers query this directory for data sources observing a set of attribute constraints and obtain a handle to one or more candidates. An example exchange between a sensor source and consumer is depicted in 5.1. While in declarative queries, content producers and consumers are completely decoupled, in this case the infrastructure mediates a discovery process, enabling a direct connection between the communication endpoints.

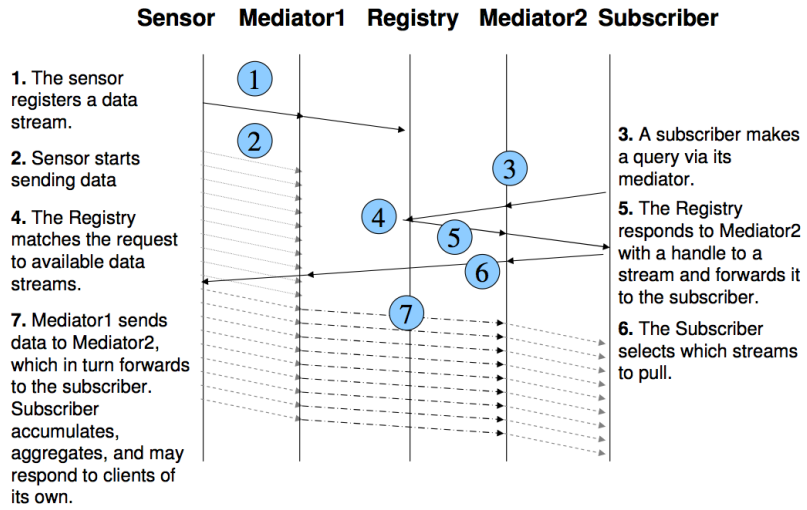


Figure 5.1: Example exchange between a sensor and content consumer in Partisans [12]

In 4Sensing, the sensor definition, virtual table and pipeline models separates querying from acquisition and data operations. This approach embodies characteristics of topic-based publish-subscribe systems and SQL based queries. Queries can be conceptualized as subscriptions with spatial restrictions, targeting specific topics represented by the virtual tables, while the expressivity of the SQL based queries, in terms of data operations, is embodied in the pipeline concept. Although clients cannot issue arbitrary queries over the available sensors, this approach has the advantage of providing a way to control the granularity of published data - a characteristic that is relevant in the participatory sensing domain.

5.1.6 Processing

Data processing refers to the computation of events, metrics or pattern recognition from raw sensor data. Events can be highly summarized information that represents relevant transitions in the state of the monitored system. Depending on the application domain, raw data tends to be more ephemeral than higher order inferences. For instance, in the context of traffic monitoring, raw data such as position and velocity readings are less significant than a transition from a blocked to unblocked state in a specific location.

Events can be triggered by individual sensors or from aggregate readings and can be derived directly from sensor readings (e.g., if velocity rises above a threshold) or from the more or less complex processing. In Pothole Patrol [11] a number of filters define a pothole detection algorithm. Additionally, several similar detections are used to improve accuracy - clustering a minimum number of detections in a particular location assists the filtering of false positives. The cluster location itself is used to summarize the contained events and thus represents an additional opportunity for data reduction.

Events can also be used as sensor triggers. In Nericell [38], this strategy, called triggered sensing, is used to enhance energy efficiency by using a relatively inexpensive sensor (in terms of energy consumption), such as an accelerometer, to trigger more expensive devices, for example a GPS or microphone. Triggered sensing is used in BikeNet [10] in a similar way, although with a different purpose, allowing, for instance, the system to activate the camera when specific conditions are satisfied by distributed sensors in a Body Area Network.

Processing can be localized on the mobile device or on the back end. The decision on where data processing takes place is mainly a tradeoff between computational, energy and communication costs on the mobile device. Heavy computations on the mobile side can be too slow to keep with the sampling rate and will drain the battery. If all computations are to be performed on the back-end, raw data has to be delivered, increasing communication costs and possibly straining a centralized infrastructure. An additional consideration relates to privacy. Nericell [38] processes raw audio feeds on the mobile node to reduce energy and communication costs, but also to avoid revealing sensitive information.

Processing, in the proposed framework, is modeled through the combination of standard and application specific stream operators structured in a processing pipeline. Mobile processing, proposed for future work, can be attained using the same pipeline approach by extending the sensor definition with the specification of a stream transformation. Similarly, triggered sensing can be modeled by the inclusion of trigger conditions in a sensor definition.

5.1.7 Data Aging

Aggregation and processing are a partial answer to face the limits in data storage capacity, still at some point data has to be discarded according to an aging strategy. Aging can be based in a set of generic or application specific data features such as age, relevance, reliability or precision. For instance, the different layers of data managed by an application - raw data, summary representations and higher order inferences - can be handled within different time scales - where more summarized information can outlive raw sensor data. This is the basis of the multi-resolution storage proposed in [15] for wireless sensor networks, where sensor data is spatially summarized in a hierarchical overlay. The higher levels in the hierarchy store coarse views of the stored data while lower levels store increasingly detailed summaries. The aging of a particular summary has to balance its space requirements, its relevance to query quality and the available resources.

The major challenge regarding data aging is to provide a balance between the necessary data reduction and the preservation of rich detailed information. Metadata can convey relevant data features that assist an aging strategy. In CarTel [24], a prioritization scheme is used to determine data transmission order in an opportunistic and intermittent connection setting. This prioritization is application specific and specified using a declarative query scheme. Although it is essentially concerned with transmission order, this data prioritization can become a relevant asset to assist an aging strategy.

Although data relevance is essentially application dependent, a generic scheme could be

devised to associate a relevance metric to specific data points. This could be based on the coverage density in a particular location, or the number of corroborating readings in a clustering scheme, as used in Pothole Patrol [11] for the validation of pothole detections.

Data aging is an open subject in 4Sensing. Assuming the sensor definition and virtual table constructs specify categories of data with different priorities, this categorization could be explored to apply differentiated aging policies. Another possible research direction would be the explicit association of custom data aging policies to sensor definitions and virtual tables.

5.1.8 Privacy

The *participatory sensing* paradigm depends on the willingness of participants to share information for a common goal. This shared data could potentially be traced back to individual users and disclose personal information beyond what the user is willing to reveal, and the problem can be aggravated by the fact that mobile sensors can be tasked to provide data without explicit user intervention. In particular, sensor readings can be annotated with time and location information, possibly revealing user identity and activity. Another potential issue could arise from the accidental invasion of privacy that sensing tasks can pose if, for instance, someone unwillingly gets caught on camera or sound recording - the concerns raised over Google Street View are a good illustration of this point [48]. These issues can undermine confidence in *participatory sensing* applications and so privacy must be at the center-stage from the beginning.

Clearly people are more and more willing to share private information, something that can be testified from services such as Facebook [13], Twitter [52] and Google Latitude [32], but together with this tendency, privacy concerns are being raised and it is not clear where to draw the line. Unlike social networks and services, where personal identity is central, *participatory sensing* essentially deals with aggregated data and anonymity is expected by potential users. Confidence must be supported by appropriate policies and privacy measures that guarantee some degree of anonymity and control over what data users are willing to submit. As a baseline, sensor readings should not be traceable to individual users.

The suppression of user identity (or mobile node identity) is not by itself sufficient to guarantee anonymity since the location and movement patterns can be enough to infer identity [29]. Different mechanisms would be appropriate for specific application domains, in some instances user control over data granularity can be well suited - Fire Eagle allows users to set up specific privacy levels for each application they decide to share location with, from exact address to country level; In Partisans [12], content providers define privacy policies, applied by mediators, that control the granularity of shared data. But ideally an automated process would determine granularity balancing anonymity and precision requirements. In [29], space is divided into *tiles*, based on historical data of user location, so that *k-anonymity* is attained with high probability - a sensor data report is associated with a time interval and a region in space, instead of a precise location, that is determined by the system so that k-1 reports from other users will probably coexist in the same space at the same time - thus preventing tracing back to a particular user.

In [22], the *Virtual Trip Line* (VTL) concept is introduced in the context of traffic monitoring. A VTL is a virtual marker in space used to trigger data sampling, allowing sampling in predefined geographic locations instead of at periodic time intervals. The motivation for this approach stems from the assumption that data is more relevant in specific areas (higher traffic roads) and more privacy sensitive on others. Careful placement of VTLs can thus improve privacy and data quality.

In PoolView [17], data perturbation is proposed as a privacy protection mechanism. Raw sensor data is collected and stored in a private storage server, and noise is added to data as it flows through a privacy firewall to an aggregation service. Knowledge of the noise model used can then be used to reveal real statistical trends from the aggregation of perturbed data.

The described privacy policies can be implemented in a user's homebase, acting as his personal store, or in mobile nodes, when the acquiring nodes are not trusted (as in QTree and NTRee). In the latter case, VTLs, tiles or data perturbation parameters can be pre-loaded into mobile nodes, and used to preserve privacy in data acquisition.

5.2 Selected Works

The following sections present a detailed discussion of selected works. This selection does not intend to be a comprehensive survey, instead it focus on works that illustrate some of the aspects introduced in section 5.1.

5.2.1 MetroSense

In MetroSense [4], the authors discuss their concept of a people-centric sensing system structured around the three stages of sensing, learning (data analysis and inference) and sharing. An abstract framework is presented supporting different applications categorized according to social domains - personal, community and public.

MetoSense presents a flexible architecture where sensing, learning and visualization functions can be hosted on the sensing device, remote servers or split between the two depending on factors such as resource requirements, communication constraints and social domain.

The main focus of the MetroSense project is on the design of a robust opportunistic sensing model. The authors formalize two people-centric sensing approaches, opportunistic and participatory, as extremes of a continuous spectrum regarding the role of the user in the sensing process. On one extreme, the participatory model requires that users consciously decide which application requests to service. The MetroSense vision focuses on the opportunistic model, where application requests are serviced by sensing nodes, matching the application's context requirements, without user awareness of which applications they are servicing at a given time.

One of the main concerns in MetroSense is to counter the limitations imposed by this unassisted model and by device heterogeneity both having consequences on data accuracy. The proposed framework leverages device context information and collaboration between nearby

devices to better match sensing request requirements. On the data analysis (learning) stage relations between data sensed by different devices are established to enrich collected data.

The construction of inference models (for higher-level semantic information e.g., activity classification) is also central to the MetroSense project. To lighten the burden on individual users, the framework proposes the distribution of model training activities such as labeling and the exploration of community commonalities to benefit the model accuracy.

Regarding information visualization and sharing, MetroSense explores several strategies for specific domain applications, such as integration with social network services, virtual worlds and community websites.

The model proposed in this dissertation, is closer to the participatory approach, since application requests are serviced by the community of users that have opted in to a particular application. However, servicing particular requests does not necessarily require user awareness, which would result in an excessive user burden. Since the sensor definition and virtual table constructs, which can be subject to user review, pre-define the application data usage, users are still in control of the kind of requests they are willing to serve. Still, the discussion in MetroSense regarding sensor context and heterogeneity, inference and sharing are relevant for the proposed model.

5.2.2 CarTel

CarTel [24] is a research project developed by the MIT Computer Science and Artificial Intelligence Laboratory, concerned with the design and evaluation of a software platform that supports the implementation of mobile sensing applications. The main design goal is to hide the complexities of a mobile sensor network from specific applications by providing a simple programming model and handling data heterogeneity and intermittent network connectivity in a transparent way. CarTel focus on vehicle based sensing applications and bases communication on opportunistic wireless connection (Wi-Fi, Bluetooth or cellular) and has been tested in domains such as traffic analysis, monitoring of road surface conditions (Pothole Patrol [11]) and mapping of WiFi coverage.

To simplify the programming model CarTel opts for a centralized architecture where applications are hosted in a central server (the portal). Applications specify data acquisition requirements through a modified SQL language that allows the definition of continuous queries that regularly sample sensor readings. Queries are issued by applications and forwarded to mobile devices and sensor readings data streams are sent back to the central portal and stored in a relational database. The query language allows for the specification of data acquisition requirements such as specific sensors, sampling rates, filtering and summarization in an application independent way.

Transport is enabled by a custom built network stack (CafNet [5]) both for the delivery of queries to mobile nodes and the streaming of data back to the portal. Central to CafNet is the assumption of opportunistic wireless access resulting in great variability of network conditions and long periods of disconnection. This assumption determines the carry-and-forward design

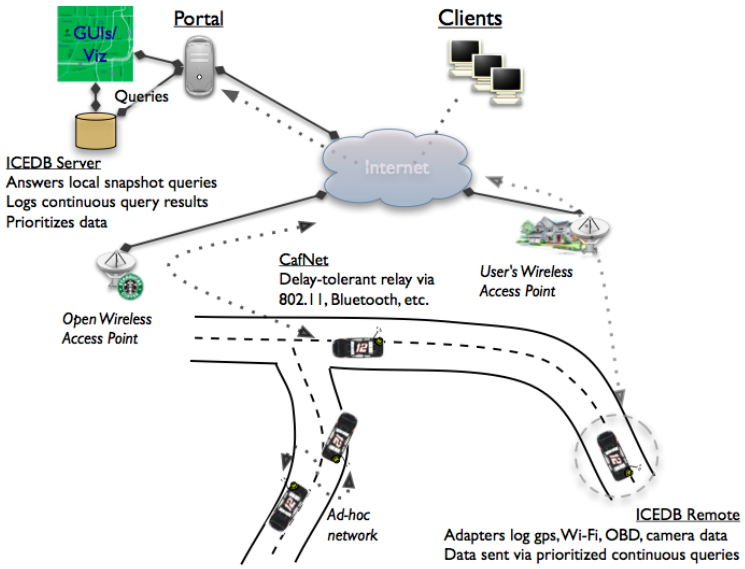


Figure 5.2: Cartel system architecture [24]

of CafNet, where mobile nodes store query results in a local buffer while network connection is unavailable, and the design of a rich framework for data prioritization. The data priority determines the delivery order and eventual discard of lower priority data in case of overflow on the mobile node. Data management is thus dictated by the delivery network conditions. The domain application can control local prioritization by declaring strict query priorities and data ordering requirements (for instance to deliver first the endpoints of a GPS trace and then the recursive bisection of that trace). Priorities can also be established based on a global system view, for instance to give higher priority to data in areas with low coverage. For this effect, applications submit special high priority summarization queries and get back aggregate data. This aggregate data is then used to compute, and forward to remote nodes, the assigned priorities for the underlaying detailed data.

CarTel does not implement any generic provision for data aging and no metadata is collected that could aid data management decisions - information is centrally stored as long as relevant for the specific application domain. Although prioritization and summarization data, computed in the remote nodes, is used exclusively to support the carry and forward model, this ancillary data could be used to assist data aging policies.

Continuous queries, in CarTel, are used to control data acquisition and assume the role of sensor definitions in the proposed model. The focus on opportunistic data delivery and the abstraction from heterogeneous sensors are relevant in a distributed setting and are complementary to the work developed in this dissertation.

5.2.3 BikeNet

BikeNet [10] is a mobile sensing application that provides cyclists with context regarding their cycling experience in terms of sensed data and derived information, such as performance and health metrics. A community portal (BikeView) adds a social dimension allowing user groups to visualize shared data. The project also intends to explore the cooperation between personal goals (through storage and analysis of personal data) and communal goals (e.g., CO_2 level mapping from aggregate data).

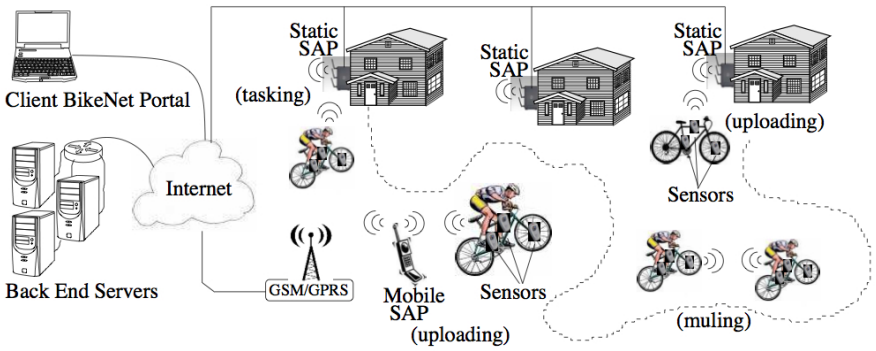


Figure 5.3: BikeNet system overview [10]

A three tier architecture separates mobile sensing elements, fixed sensor access points (SAPs) and a centralized back-end service. Mobile sensing elements deployed on a bicycle or weared by the user, such as acelerometers, magnetometers, CO_2 readers, etc., are connected through short range radio creating a body-area/bycicle-area network (BAN). BikeNet explores different query modes - besides continuous sensing, BikeNet supports triggered sensing and back-end querying. In triggered sensing, sensor devices in the BAN are tasked to notify specific conditions, and a sensing action, such as video or audio capture, is triggered, by a coordinating node, when the aggregate condition is satisfied by the participating sensors. Back-end queries can be configured in the BikeView portal and delivered to a SAP, which coordinates with in-range sensors that qualify for the query requirements and delivers the sensed data.

Another aspect explored by BikeNet is the usage of different delivery modes, according to specific timing requirements. The system defaults to a delayed delivery mode, where BAN data upload is performed at the end of the ride. An opportunistic delivery mode allows sensing nodes to deliver data directly through nearby fixed SAPs or indirectly through an inter-BAN muling scheme to optimize delivery time. Real-time delivery, essentially used to support back-end querying, is accomplished by integrating a mobile phone on the BAN, acting as a mobile SAP.

The BikeNet vision is to provide a set of tools supporting personal archiving, selective sharing among group members, and public sharing. With this purpose, the authors present a set of higher level inferences that characterize a ride in terms of topography, performance

and environment. Additionally, health and performance metrics provide the context for the interpretation of the cycling experience. Visualization tools, in the BikeView portal, allow a user to plot these metrics, and the subjacent data sets, and infer the best routes according to health and performance goals.

Salient features in BikeNet, from a data management perspective, include the flexible acquisition control, querying models and delivery modes. Additionally, the primary focus on the personal domain, while social and public goals can leverage the data collected - in part motivates the homebase concept proposed in this dissertation.

5.2.4 IrisNet

IrisNet [19] is a software platform designed to enable the development and deployment of wide-area sensor network applications. The authors envision a worldwide sensor web, consisting of common PCs connected to the internet and equipped with sensors. The framework intends to leverage this infrastructure and provide the necessary services for domain applications, such as service deployment, a common sensor host runtime environment, distributed storage and a rich query interface that provides a unified system view. Although the envisioned worldwide sensor web consists of fixed nodes, its architecture can be adapted to mobile sensing infrastructures. IrisNet has been tested with domain applications such as parking-space localization, network and host monitoring, and a coastal imaging service to identify nearshore phenomena.

The main concerns in IrisNet are related with the runtime environment on the sensor hosting node (*Sensing Agents* - SAs) and the overall architecture for data storage and retrieval. Regarding SAs, the design goal is to allow different services to share the same hosts and sensors, in a controlled and efficient way, for data collection and processing activities. The SA provides the interface and environment for applications to deploy and control the execution of *senselets* - that act as agents for specific applications, collecting, processing and uploading sensor data - and cater for security, privacy and sharing of heavy computation such as image processing.

Regarding data storage and retrieval, IrisNet abstracts the common requirements for sensor web applications and addresses the scalability issues imposed by high update rates and data volumes. To that end, a distributed architecture is proposed, supporting partition and replication, based on the hierarchical structuring of sensor data according to geographical or administrative domains (e.g., in the parking-space finder application data is structured according to block, neighborhood and city domains). Specific levels in this hierarchy are dynamically allocated to data management nodes (*Organizing Agents* - OAs) by a distributed algorithm, in order to optimize query response time, network traffic and load. For instance, the assumption that the geographical area of interest of a query is usually related to the location of the querier is explored by placing sensed data close to the origin.

Sensor data is represented according to an XML schema that naturally mirrors the data hierarchy, while data queries are specified using XPath. Distributed XPath queries are performed by determining the *lowest common ancestor* (LCA) of the potentially selected nodes and forwarding the query to the associated OA, thus avoiding overloading a global root. This OA will

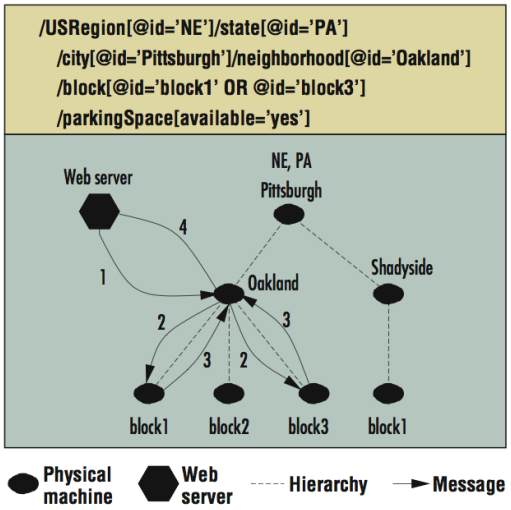


Figure 5.4: A query processing example in IrisNet [19]

then compose a query response from locally stored data or collaborating with other OAs down the data hierarchy. Partial results from lower levels of the data hierarchy can be cached locally to improve performance. An example of the query process is depicted in Figure 5.4 for the Parking-space-finder service.

A scheme based on DNS supports mapping a node in the data hierarchy to the OA responsible for that node. This scheme is also the base for the replication support. Clients can query DNS for a list of primary replicas maintaining strong consistency or fallback to query the list of secondary replicas with weaker consistency requirements.

Storage and query of historical data is approached in IrisLog, a test application deployed in PlanetLab that monitors hosts and networks, using multiresolution vectors to store data samples with lower sampling rates for older data.

In 4Sensing, multi-layer aggregation in the NTree distribution strategy can be performed by applying successively coarser mapping operators. These successive levels results in the hierarchical partitioning of the data space (e.g., from road segment to city level) and the assignment of levels of this hierarchy to particular nodes. This resembles the IrisNet approach, although in this case the levels of the hierarchy are not defined by the data schema of the application, instead being determined by the virtual tables involved in a particular query.

The hierarchical data scheme inherent to the IrisNet architecture might not be well suited for applications domains where the geographical or administrative domains are not easy to establish. However, the data placement approach, that takes into account a set of metrics to optimize resource usage, could be extrapolated for a non-hierarchical domain. For instance, in a placements scheme based on the geographic partition of space, the spatial partitioning can evolve over time in order to reduce load and storage requirements.

5.2.5 PoolView

In PoolView [17], the authors propose a mathematical model and architecture for privacy protection in *participatory sensing* applications. In people-centric sensing applications, where participants can deploy, by individual initiative, data collection activities for the public interest, a central authority entrusted with data protection cannot be assumed. Data perturbation is proposed as a privacy protection mechanism applied to individual data streams before uploading to an aggregation service. Although the original individual data cannot be reconstructed from the shared streams, aggregate computations obtained from the modified data will closely match the values for the original data. To test the proposed model, the PoolView architecture is applied to the collection of traffic speed statistics and the collection of aggregate weight loss for participants in a particular diet.

In the proposed architecture, an Information Distilation Layer is composed of data aggregation services, or *pools*, deployed by any interested party to compute aggregate data relevant to the community. The aggregation service acts as a centralized server and the architecture assumes its permanent availability, which, in a grassroots initiative, can be difficult to ensure. On the client side, a private storage server owned by the end user stores raw data from individual sensors. Between the private storage and the data *pools* a privacy firewall enforces privacy by intercepting requests and applying noise to data.

Since adding random noise is ineffective for data protection, the authors propose the use of application-specific noise generated from a mathematical model that approximates the measured data. The mathematical model, and the distribution probabilities of the model parameters, are proposed by the data *pool* owner and shared among the application users. An individual user picks model parameters according to the known distribution (using client software) to configure his privacy firewall. The firewall will then add noise data, obtained from the mathematical model and particular parameters, to the real data stream. To ensure trust on the noise model, a validation procedure based of curve-fitting is enforced on the initial data upload to guarantee that actual data reasonably fits the noise model. At the server side, knowledge of the shared model and parameter probability distribution allows the subtraction of the noise from the sum of community data series and thus reveals the real aggregate values.

The need for a mathematical model that approximates the phenomena that is being measured is probably a relevant weakness of the proposed approach. To mitigate this issue, the authors argue that an hypothesis is usually subjacent to scientific experimentation and thus a model is usually available; however, this view conflicts with the grassroots nature of some *participatory sensing*, limiting its applicability. The proposed solution also leaves aside what is referred as *context privacy* and its consequence is evident in the collection of traffic speed statistics, where actual speed values are obfuscated but user localization data is shared without privacy protection. Still, this does not invalidate the data perturbation approach; complementary privacy protection mechanisms, such as Virtual Trip Lines[22] or k-anonymity [29], could be used to solve this issue.

The concept for a personal firewall, implementing privacy preserving data transformations,

is an important aspect of a participatory sensing infrastructure that can be incorporated in the homebase role, proposed in this work, or in mobile nodes, depending on where the distribution strategy establishes a frontier of personal trust. In a personal store model, a homebase can assume the role of privacy firewall, while for the approaches where mobile nodes bind directly with an acquiring node, this firewall would be hosted on mobile nodes

5.2.6 Distributed Spatial Indexes

Most peer-to-peer networks are designed to support simple queries based on data object identifiers (e.g., [50], [46] and [44]). Distributed spatial indexes try to address this limitation, extending search in peer-to-peer networks to support spatial queries. This section describes two works that adapt centralized spatial indexes - R-tree and MX-CIF quadtree - to a distributed setting.

P2PR-tree builds on the centralized R-tree spatial index [21] and adapts it to peer-to-peer overlay networks. It uses a tree-based hierarchical decomposition of space, where each peer represents its stored data by a Minimum Bounding Rectangle (MBR). The main idea is that each peer stores a partial index view, so that the amount of information known concerning nearby peers is greater than that available for distant peers. When a node joins the network, it starts by contacting an arbitrary peer to get the top-level space subdivision (and a known peer contained in each subdivision). The node stores this information, determines which subdivision(s) encloses its own data and contacts the respective peer(s) to gather further subdivisions down the tree. This process is repeated until reaching the leaf level.

Static decomposition is used for the first two levels of the hierarchy, i.e., the regions at these levels do not depend on the peers participating in the network, and respective data. For lower levels, space is dynamically partitioned using a clustering technique. Partitioning occurs when the maximum number of children for a region has been reached. Partitioning implies contacting every node in the affected region - this is the main motivation for static decomposition at the upper levels, since dynamic partitioning would imply contacting an extremely large number of peers.

Query processing works by having a query traverse the distributed index until reaching relevant peers (the leaves of the tree) i.e., those peers whose MBRs intersect the query window. A query can be issued at an arbitrary node, which will start by contacting a peer at the enclosing level 0 region. The process is repeated down the tree until the leaves have been reached. To deal with network dynamism, redundant relations for each tree level are built from interaction between peers during querying.

P2PR-tree has some constraints, which restrain its general applicability. On one side, it assumes that nodes do not store data than spans separate regions in space. The index looses its efficiency if the node stores disperse data, as in this case the enclosing MBR will cover a wide and sparsely populated area, resulting in less effective pruning of the search space.

Another spatial index - the MX-CIF quadtree - is adapted to a peer-to-peer network in [51]. The MX-CIF quadtree uses recursive subdivision of space to assign spatial objects to the index nodes. To assign an object to a node, space is successively decomposed into regular blocks (or

quadrants) until reaching a block where the object intersects at least two of its child blocks.

To distribute the index, the authors propose the use of a key-based routing protocol. The responsibility over each block is assigned to a particular peer by mapping the block centroid to a peer address using a hash function - the prototype implementation uses the Chord [50] routing model and the SHA-1 hashing algorithm. To execute a query, the query object is propagated down the tree index; at each level, a peer test for intersection with local data and, if necessary, propagates the query down the index tree using the Chord lookup algorithm.

A drawback of the quadtree approach is that a single node is responsible for a particular spatial region. Given the hierarchical structure of the index, the upper level nodes are visited for every query execution. To mitigate this problem, the index is modified so that data can only be assigned at levels above a *fundamental minimum level*. In this modified version, queries are initially routed to the nodes responsible for blocks at this level that intersect the query bounds.

Both indexes described above can be explored to form distribution strategies. Using the mix of static and dynamic decomposition proposed in P2PR-tree, node responsibilities over spatial partitions can be configured dynamically, as nodes enter and leave the system, according to the (stored or live) data distribution. Localization of relevant data, in this case, is performed by traversing the tree index, and the resulting search tree can be used for multi-level aggregation. The quadtree distributed index suggests an approach for the implementation of a distribution strategy over a key-routing substrate such as Chord. As in P2PR-tree, the index tree can be used for multi-level aggregation. However this approach has the drawback that data locality is not preserved i.e., data relative to a particular geographic region is potentially dispersed across the globe, and, given that the index structure is based on a single tree, there is no opportunity to balance the query processing load.

QTree mixes some aspects of the described spatial indexes. It uses a regular spatial decomposition characteristic of quadtree indexes. As in the distributed MX-CIF quadtree, each level of the decomposition is assigned to a particular node. However, QTree builds a query specific tree during query dissemination instead of relying on a fixed index tree. This is possible because the responsibility over each partition is shared among the enclosed nodes, i.e., all nodes within a given partition have the same role, a characteristic shared with P2PR-tree.

6. Conclusions and Future Work

This dissertation proposes a model for the development of participatory sensing applications, using a distributed infrastructure based on personal computing resources, providing a set of constructs that support data acquisition, querying and data sharing between applications. Additionally, three distribution strategies are proposed and evaluated in terms of their usage of the infrastructure resources. The following section presents the conclusions regarding these two aspects.

6.1 Model

This section revisits the requirements expressed in the introduction section and discusses how they are met by the proposed model.

Data acquisition Applications express data acquisition requirements in terms of *sensor definitions* that allow applications to convey a wide range of requirements - delivery QoS, sampling modalities, sensor tasking and persistence - covering the common requirements found in the studied applications. Defining acquisition requirements as sensor definitions has the additional advantage that users can easily review the acquisition policies for a particular application.

Data processing and aggregation The pipeline processing model, together with a library of stream operators allows applications to implement common processing and aggregation tasks by composing out-of-the-box components. Generic components, such as the *processor* and *filter* support domain specific transformations. Additionally, applications can develop specific components by extension of the component library.

Querying The proposed querying model abstracts applications from the actual location of the target data, presenting an unified system view - queries are expressed as if data is centralized. A simple query model was chosen, where applications express a target table and interest area and the data operations are specified by pre-defined *virtual tables*. In contrast to a model where queries specify the data operations, this gives applications control over how data is used by clients and other applications, an important aspect in order to establish trust.

Persistence Through the *stored stream* abstraction, applications can access historical data using the same constructs used for continuous data processing. Controlling persistence through sensor definitions and virtual table definitions, applications can specify where persistence is relevant and where data has a volatile nature.

Data sharing Sharing of data between applications is enabled by virtual table composition, enabling application mashups and promoting the *data commons* principle. Exposing data through virtual tables allows applications, to control the granularity of published data.

6.2 Distribution Strategy

Three distribution strategies, RTree, QTree and NTree have been evaluated experimentally, using the SpeedSense case-study application. In RTree data is acquired by a personal store and aggregated using a random tree spanning the nodes with relevant data. QTree successively divides space into quadrants, to partition data among nodes, and uses a structured tree for aggregation where the tree levels towards the root are assigned over successively wider areas. NTree uses a nearest neighbor relation for data partitioning; a *geographic hashing* function is used to forward data to the nearest neighbor, thus building the aggregation structure from bottom up. The experimental evaluation resulted in the following conclusions:

RTree RTree has a clear advantage regarding the balanced distribution of acquisition work. For smaller queries and lower node densities, this advantage accumulates as more queries are issued, given RTree the best overall performance. The downside of RTree is its inability to effectively limit aggregation work; for larger query areas and higher fixed node densities, aggregation work dominates, degrading the performance relatively to the other strategies.

QTree With a structured approach, QTree can limit the aggregation work more effectively than RTree, resulting in better scaling regarding the query area and number of fixed nodes. The downside in QTree is that acquisition work is not balanced due to the uneven mobile node distribution, thus compromising performance in a low density setting.

NTree For higher node density, NTree showed the best overall performance, given its ability to limit aggregation work. As QTree, the acquisition work is unbalanced and, for lower node densities, performance degrades significantly.

The results show that choosing an effective distribution strategy depends on the characteristics of the fixed infrastructure and the application domain. For low density and small queries RTree is more effective, while with a high density fixed infrastructure, QTree and, specially, NTree have a significant advantage. A downside of NTree, is that only one node (and always the same node) can handle aggregation for a particular spatial extent. For applications that use coarser data aggregation - for instance, city level aggregation - load will be directed to specific nodes. QTree provides an inherent balancing mechanism - the wider the aggregation area, the larger the set of candidate nodes that can handle aggregation, and a different node is chosen each time the aggregation tree is built.

6.3 Future Work

This section presents the proposed future work, covering relevant extensions to the presented models and some identified aspects that require further research.

Multi-query Processing and Efficient Result Delivery The expected usage of the 4Sensing framework involves multiple simultaneous queries issued by mobile devices with overlapping areas of interest. Pipeline processing should avoid redundancy by merging computations for several queries and query results should be efficiently delivered to the mobile clients.

Persistence and Replication Some aspects of data persistence have been left open in the proposed model. The presented distribution strategies leave open the particular policy to distribute and replicate data among candidate nodes. Additionally, applications should be able to specify data aging policies, defining, for instance, storage limits for virtual tables so that older data is eliminated as new data becomes available.

6.4 Contributions

This dissertations explores the development of a distributed data management model for the generic support of *participatory sensing* applications, with the following main contributions:

- A generic framework supporting the data life-cycle of applications based on geo-referenced data, including: data acquisition, processing and aggregation, storage, querying and data sharing between applications;
- A flexible decentralized model for data processing, supporting different overlay network structures and data distribution strategies, that leverages the advantages of a hybrid fixed/-mobile infrastructure;
- The development and comparative evaluation of three alternative models for data distribution in an hybrid infrastructure - RTree, QTree and NTree.

Bibliography

[1] Guardian.co.uk Mapping a Crisis. http://www.guardian.co.uk/open-platform/blog/mapping-a-crisis, June 2010.

[2] MESSAGE Mobile Environmental Sensing System Across a Grid Environment. http://www.commsp.ee.ic.ac.uk/ wiser/message/, July 2009.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, VLDB Journal, 2003.

[4] A.T. Campbell, S.B. Eisenman, N.D. Lane, E. Miluzzo, R.A. Peterson, Hong Lu, Xiao Zheng, M. Musolesi, K. Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *Internet Computing, IEEE*, 12(4):12–21, July-Aug. 2008.

[5] Kevin W. Chen. Cafnet: A carry-and-forward delay-tolerant network. Master's thesis, Massachusetts Institute of Technology, February 2007.

[6] CitySense. http://www.sensenetworks.com/citysense.php, July 2009.

[7] Dana Cuff, Mark Hansen, and Jerry Kang. Urban sensing: out of the woods. *Commun. ACM*, 51(3):24–33, 2008.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI*, pages 137–150, December 2004.

[9] Walkin Paper Project Stamen Design. http://walking-papers.org, June 2010.

[10] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G-S. Ahn, and A. T. Campbell. The bikenet mobile sensing system for cyclist experience mapping. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 87–101, New York, NY, USA, 2007. ACM.

[11] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: using a mobile sensor network for road surface monitoring. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 29–39, New York, NY, USA, 2008. ACM.

[12] A. Parker et al. Network system challenges in selective sharing and verification for personal, social, and urban-scale sensing applications. In *Proc. 5th Workshop Hot Topics in Networks (HotNets-V)*, 2006.

[13] Facebook. http://www.facebook.com, June 2010.

[14] FixMyStreet. http://www.fixmystreet.com, July 2009.

[15] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An evaluation of multi-resolution storage for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 89–102, New York, NY, USA, 2003. ACM.

[16] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 19–24, New York, NY, USA, 2004. ACM.

[17] Raghu K. Ganti, Nam Pham, Yu-En Tsai, and Tarek F. Abdelzaher. Poolview: stream privacy for grassroots participatory sensing. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 281–294, New York, NY, USA, 2008. ACM.

[18] Shravan Gaonkar, Jack Li, Romit Roy Choudhury, Landon Cox, and Al Schmidt. Microblog: sharing and querying content through mobile phones and social participation. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 174–186, New York, NY, USA, 2008. ACM.

[19] P.B. Gibbons, B. Karp, Y. Ke, S. Nath, and Srinivasan Seshan. Irisnet: an architecture for a worldwide sensor web. *Pervasive Computing, IEEE*, 2(4):22–33, Oct.-Dec. 2003.

[20] Groovy. http://groovy.codehaus.org, April 2010.

[21] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, 1984.

[22] Baik Hoh, Marco Gruteser, Ryan Herring, Jeff Ban, Daniel Work, Juan-Carlos Herrera, Alexandre M. Bayen, Murali Annavaram, and Quinn Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 15–28, New York, NY, USA, 2008. ACM.

[23] Yongqiang Huang and Hector Garcia-Molina. Publishsubscribe in a mobile environment: Data engineering for mobile and wireless access (guest editors: Panos k. chrysanthis and evaggelia pitoura). *Wireless Networks*, 10(6):643+.

[24] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a distributed mobile sensor computing system. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 125–138, New York, NY, USA, 2006. ACM.

[25] Project Icarus. http://space.1337arts.com/, June 2010.

[26] KD-Tree Implementation in Java and C#. http://www.cs.wlu.edu/ levy/software/kd/, May 2010.

[27] Ushahidi Crowdsorcing Crisis Information. http://www.ushahidi.com/, June 2010.

[28] Java OpenStreetMap Editor (JOSM). http://www.openstreetmap.org/, May 2010.

[29] Apu Kapadia, Nikos Tri, Cory Cornelius, Daniel Peebles, and David Kotz. Anonysense: Opportunistic and privacy-preserving context collection. In *In Proc. of 6th Int'l Conf. on Pervasive Computing*, 2008.

[30] N. Kotilainen, M. Weber, M. Vapa, and J. Vuori. Mobile chedar - a peer-to-peer middleware for mobile devices. In *Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on*, pages 86–90, 2005.

[31] Nicholas D. Lane, Shane B. Eisenman, Mirco Musolesi, Emiliano Miluzzo, and Andrew T. Campbell. Urban sensing systems: opportunistic or participatory? In *HotMobile '08: Proceedings of the 9th workshop on Mobile computing systems and applications*, pages 11–16, New York, NY, USA, 2008. ACM.

[32] Google Latitude. http://www.google.com/intl/en_us/latitude/intro.html, June 2010.

[33] Liqian Luo, Aman Kansal, Suman Nath, and Feng Zhao. Sharing and exploring sensor streams over geocentric interfaces. In *GIS '08: Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, New York, NY, USA, 2008. ACM.

[34] Samuel Madden, Samuel Madden, Michael J. Franklin, Michael J. Franklin, Joseph Hellerstein, Joseph Hellerstein, Wei Hong, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *In OSDI*, 2002.

[35] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[36] Grassroots Mapping. http://grassrootsmapping.org, May 2010.

[37] Mobile Millennium. http://traffic.berkeley.edu, July 2009.

[38] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336, New York, NY, USA, 2008. ACM.

[39] Anirban Mondal and Yilifu Masaru Kitsuregawa. P2pr-tree: An r-tree-based spatial index for peer-to-peer environments. In *In Proceedings of the International Workshop on Peer-to-Peer Computing and Databases (held in conjunction with EDBT*, pages 516–525. Springer-Verlag, 2004.

[40] Suman Nath. Challenges in building a portal for sensors world-wide. In *In First Workshop on WorldSensor-Web, Boulder,CO*, pages 3–4. ACM, 2006.

[41] PEIR Personalized Estimates of Environmental Exposure and Impact. http://urban.cens.ucla.edu/projects/peir/, July 2009.

[42] Harry Wood Haiti Earhquake on OpenStreeMap. http://www.harrywood.co.uk/blog/2010/01/21/haiti-earthquake-on-openstreetmap/, June 2010.

[43] OpenStreeMap. http://www.openstreetmap.org/, April 2010.

[44] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[45] O. Riva and C. Borcea. The urbanet revolution: Sensor power to the people! *Pervasive Computing, IEEE*, 6(2):41–49, April-June 2007.

[46] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[47] SeeClickFix. http://www.seeclickfix.com, July 2009.

[48] Times Online All seeing Google Street View prompts privacy fears. http://technology.timesonline.co.uk/tol/news/tech_and_web/article1870995.ece, June 2010.

[49] Spacebits. http://spacebits.eu, June 2010.

[50] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[51] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *VLDB Journal*, 16:165–178, 2007.

[52] Twitter. http://www.twitter.com, June 2010.

[53] Cuidemos El Voto. http://www.cuidemoselvoto.org, July 2009.