2010 16th International Conference on Parallel and Distributed Systems

4Sensing - Decentralized Processing for Participatory Sensing Data

Heitor Ferreira, Sérgio Duarte, and Nuno Preguiça
CITI / DI-FCT-UNL
Quinta da Torre, 2829-516 Caparica, Portugal
Telephone: (+351) 212948536
Fax: (+351) 212948541
Email: heitorfr@gmail.com, {smd, nmp}@di.fct.unl.pt

***Abstract*—Participatory Sensing is an emerging application paradigm that leverages the growing ubiquity of sensor-capable smartphones to allow communities carry out wide-area sensing tasks, as a side-effect of people's everyday lives and movements. This paper proposes a decentralized infrastructure for supporting Participatory Sensing applications. It describes an architecture and a domain specific programming language for modeling, prototyping and developing the distributed processing of participatory sensing data with the goal of allowing faster and easier development of these applications. Moreover, a case-study application is also presented as the basis for an experimental evaluation.**

Index Terms - Participatory Sensing, decentralized processing, data streaming, mobile computing.

I. Introduction

Participatory Sensing [2], [3] is a new application paradigm that aims to turn personal mobile devices into advanced mobile sensing networks. Thanks to the willingness of the users and the inherent mobility of their daily routines, it is possible to assemble detailed views and interpretations of the physical world without the costs associated with the deployment of dense, wide-area, sensing infrastructures. Examples of the potential of this paradigm already exist, such as experiments combining accelerometer and GPS data to monitor road conservation [5] and traffic congestion [10], [14]. The expected outcome from this new movement is the creation of rich data sets, a data commons, supporting new services, discourses and interpretations.

The first wave of case-study applications already shows the potential of this paradigm, but their relatively confined specificity also exposes their exploratory stage. In general, these applications stand on top of adapted middleware architectures, in many cases using centralized entities [5], [10], [14], or limited distribution models. The more encompassing efforts at platform support tend to originate from fixed sensor networks backgrounds, with limited allowances for mobility [9], [12]. Centralized solutions raise several problems. On one hand, there are the implications of having a centralized repository hosting privacy sensitive information. On the other hand, a centralized model has financial costs that can discourage community-driven initiatives.

This paper focuses on the issue above and proposes a decentralized infrastructure for supporting participatory sensing applications, whose goal is to ease the prototyping and development of participatory sensing applications. The 4Sensing [1] system, includes a framework for modeling and carrying out the processing of participatory sensing data in a decentralized, fully distributed fashion.

The rest of the paper is organized as follows. Section II presents a review of related work. Section III describes the overall system model, including the proposed system architecture, processing language and execution environment. A case-study application and experimental results are presented and discussed in Section IV. Section V concludes the paper with our plans for future work and some final remarks.

II. Related Work

The availability of mobile devices with several sensors, such as smartphones, has lead to the creation of a large number of personal sensing applications, such as applications to record walks, routes, etc. These applications focus mostly on archiving and personal monitoring, for instance for health monitoring or fitness applications.

Some of these applications involve sharing among a specific community group or social network. In public sensing, data is open to the public at large. For example, in BikeNet [4], users can record their bike rides in their mobile phones and share this information with a community. The aggregation of this information, executed in a single server, allows community members to get information about the most popular bike routes. In CenceMe [13], an application running in the users smartphone infers the presence of individuals and shares this information through social networking applications such as Facebook and MySpace.

In [1], users collect information about the existence of flowers in some given place using their GPS-equipped mobile phones. This information is aggregated in a single server, allowing the exploration of the bio-diversity of plants in specific areas.

CarTel [10] focus on vehicle based sensing applications and bases communication on opportunistic wireless connection. The system has a centralized architecture, where applications

[1]This work was partially supported by FCT/MCTES, project PTDC/EIA/76114/2006 and CITI.

1521-9097/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPADS.2010.20

are hosted in a central server (the portal). Mobile nodes, executing in vehicles, are used to collect the information need by the running applications. Example applications include hot spot detection, querying popular car routes between two points, and monitoring of road surface conditions [5].

Our approach differs from these systems in a number of ways, the most important being its decentralized architecture. A decentralized solution helps scalability by spreading the load and storage requirements by the participants in the system. Lacking the need for having a powerful server infrastructure also makes this approach more suitable for community-based sensing, with the needed resources being contributed by the community.

Some sensing systems present an architecture built entirely in the mobile nodes and ad-hoc coordination to support applications (e.g. [16] and [11]). 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.

Other systems, such as IrisNet [7] target a worlwide sensor web consisting of common PCs connected to the internet and equipped with sensors. In this system, nodes are hierarchically organized, and data is stored and processed according to this organization. Unlike IrisNet, 4Sensing supports the integration of mobile sensor nodes.

In the Partisans architecture [6], 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.

Other systems (e.g., [6], [9], [12]) have an architecture similar to ours, based on the combination of fixed and mobile nodes. For example, in SensorWeb [9], a set of fixed nodes act as gateways for sensor network and proxies for mobile devices. In this system, a coordinator node mediates and coordinates the access of applications to the different gateways and proxies. In Partisan [6], sensor nodes generate information that is consumed by subscribers. Mediators perform limited in-network processing of the data streams. Unlike these systems, in 4Sensing, a network of nodes is used to process the information in a decentralized way, thus allowing to more evenly distributed the load among the participants in the system.

III. System Model

A. Architecture

4Sensing is a distributed system for running participatory sensing applications 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 have limited resources, in particular in terms of computing power and battery life. Mobile nodes are connected to a fixed infrastructure, composed by a set of nodes organized as an overlay network, which 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 the 4Sensing service middleware running in both mobile and fixed nodes. On the fixed infrastructure, an *application context* defines data acquisition, processing and storage requirements, while on the mobile side, contexts support the data acquisition tasks. Application clients - hosted on mobile or fixed nodes - issue queries and receive result streams through the service middleware, thus supporting the interaction between user and service. Figure 1 provides an illustration of the 4Sensing high-level architecture.

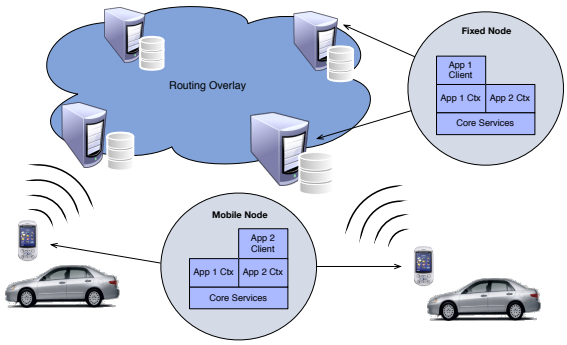

Figure 1. 4Sensing high-level architecture

B. Data Model and Processing Language

1) Streams: Participatory sensing 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, represented as a set of named attributes. Regarding its origin, a *data acquisition stream* is a tuple sequence produced by mobile nodes according to the acquisition requirements expressed by an application. This *raw* data has a spatial and temporal nature - a timestamp records the time when the sensor reading was sampled, while physical coordinates, or a spatial extent, convey the location in space that the sample refers to. A *derived stream* results from the application of a transformation operation over a source stream.

As an example, an application dealing with detecting congested roads in an urban setting would process a raw stream of GPS readings, obtained from *in situ* mobile nodes, and produce a derived stream of average speeds values to feed a hotspot detection logic component.

2) Virtual Tables: Similarly to the relational paradigm, where a data set with a common schema is represented

by a relation, or table, in 4Sensing, *participatory sensing* applications model data by defining *virtual tables*. A virtual table specifies a derived stream in terms of one or more inputs - either a set of data acquisition streams or another virtual table - and a sequence of stream operators structured in a processing *pipeline*. The term virtual is used here because tables do not necessarily have associated storage - although the same abstraction can be extended to support persistence by storing and replaying a previous *live* stream. A *query* expresses a spatial constraint over a virtual table - a bounding box, for instance. The result of a query is a continuous tuple stream produced by the target virtual table, resulting from the application of the spatial restriction over its base stream.

In the above example, the derived average speeds stream would be modeled as a *virtual table*, using GPS readings for its acquisition input, and could be used by some monitoring application instance (running on a mobile device) to issue a query to monitor a limited area around its general location.

3) Stream Transformations: A stream transformation can be modeled as a sequence of *stream operators* structured in a processing *pipeline*. Data tuples flow in a pipeline, being processed in sequence by each operator - transforming, aggregating, filtering and classifying data. Following is a description of the stream operators considered in the context of this work.

a) Processing and Filtering: The *processor* operator is the main extension basis for the implementation of domain specific processing, such as interpolation of sensor readings, unit conversion and data *mapping*. Mapping classifies data according to an uniform representation - e.g., a grid over the geographical space or the buckets in a histogram - thus establishing relations between data in order to support aggregation operations. Spatial mapping operations assign a spatial extent to data tuples, such as a bounding box, or physical coordinates representing the centroid of the extent.

In the given example, individual traffic speed samples have little use, so the aggregation spatial granularity could be controlled by the application via the use of a *processor* operator to map raw GPS readings to actual roads or road segments, in order to produce an aggregate speed value per road or road segment.

b) Partitioning: The *groupBy* operator partitions data by specific tuple attributes or an arbitrary partitioning condition, producing independent data streams. Each independent stream is processed by a sub-pipeline specified by the operator - for simplicity, each sub-pipeline cannot include itself a *groupBy* operator. Partitioning is used to group related data, for instance by creating independent streams for data in particular cells, according to an uniform grid introduced by a mapping operator.

In the running example, partitioning would refer to a *groupBy* operator devoted to the partition of incoming data into separate substreams referring to the same road or road segment, so that an aggregate average speed is generated separately for each of them.

c) Stream decomposition: To aggregate data, continuous streams have to be broken down into discrete tuple sequences. A *timeWindow* divides the stream into possibly overlapping time periods using a sliding window.

For instance, a *timeWindow* operator could be used to generate aggregate speed averages taking into account data received in the last 30 or 60 seconds, depending on the desired level of temporal detail.

d) Aggregation: An *aggregator* operates over a finite tuple sequence applying operations, such as maximum, minimum, count, sum and average, over one or more input attributes of the input tuples. An aggregator can be used together with mapping, partitioning and stream decomposition in order to continuously produce independent aggregate values over the spatial decomposition defined by the mapping operator.

In the road congestion example, the required *aggregator* operators would be *average* and *count*. Respectively, they would be used to produce the desired result of averaging the individual speed samples and, also, to provide a measure of significance of the result.

e) Classifier: A *classifier* is a specialized processor used to generate inferences from aggregated data, such as detecting an event. A classification tuple is forwarded whenever an input aggregate is *complete* and satisfies an application defined condition. An aggregate tuple is complete when it takes into account all the information bounded by its spatial extent - see section III-C.

Completing the example, a *classifier* operator would be used to implement the road congestion detection logic according to some model, for instance, taking into account the observed average speeds in regards to expected values for each given road and time of day, and using the number of available samples to attach a confidence value to each detection.

The use of these operators, in the context of the road congestion example scenario, is also illustrated in Figure 2. The figure actually shows the example stream processing *pipeline* decomposed in its two stages: one devoted to the processing of data obtained locally (*data sourcing*) and the other dealing with the processing of data obtained from remote nodes (*global aggregation*). This is intimately related to distributed processing and is explained in the next section.

The actual pipeline programming is performed using a domain specific language, whose specification is given in Figure 3. The implementation prototype uses Groovy [8] [2] and takes advantage of its dynamic dispatching/invocation features and support for closures for embeding additional application code into virtual table definitions. An example of the use of this DSL language is given in Section sec:evaluation.

C. Distribution Strategy

A stream transformation pipeline can be broken down into two stages, or roles - *data sourcing* and *global aggregation*, as shown in Figure 4. Data sourcing refers to the process through which each node produces partial state tuples from the continuous sensor input received from mobile devices, while global aggregation refers to the production of an aggregate

[2]Groovy is an object-oriented, dynamic programming language that runs on top of the Java Virtual Machine.

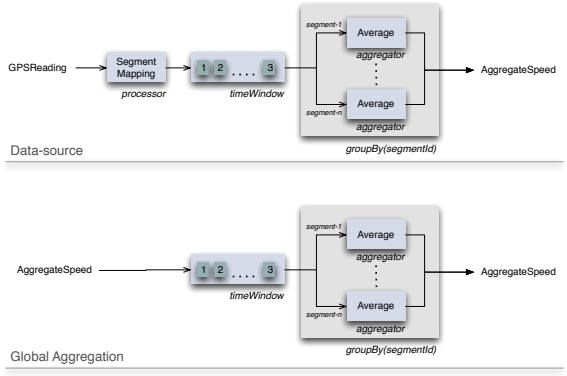

Figure 2. Example of the use of operators within the two node processing pipeline stages

base-virtual-table-definition ::=
[**sensorInput** '(' <sensor-definition-name >')']+
dataSource {<*pipeline-definition* >}
[**globalAggregation** { <*pipeline-definition* >}]*

derived-virtual-table-definition ::=
tableInput '(' <virtual-table-name>')'
[**globalAggregation** { <*pipeline-definition* >}]+

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

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

Figure 3. 4Sensing DSL specification

result by merging the partial states from several nodes. One key aspect of 4Sensing is that processing of these stream transformation operations is performed in a fully distributed way, using a strategy called QTree, as explained next.

In QTree, each fixed node is assigned a set of physical space coordinates (given by a latitude and longitude). Data partitioning is based on subdivision (or *splitting*) of geographic space into quadrants that hold at least a minimum number of nodes (the *minimum occupancy*). Each node belongs simultaneously to all the quadrants that contain it, down to the smallest - called its *maximum division quadrant*. Mobile nodes upload sensor readings to an acquisition node whose physical coordinates lie in the same maximum division quadrant (*M1* and *M2* in figure 5). Areas with low node density can result in data dispersion across the entire node base; to reduce query scope, QTree assumes a *minimum division level* - meaning that geographic space is fully divided at this level i.e., all partitions have minimum occupancy.

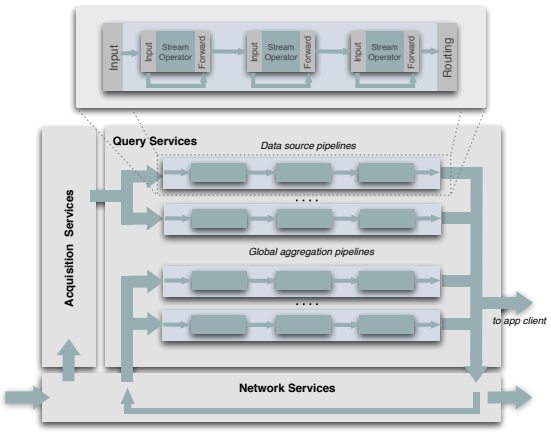

Figure 4. The data flow within a 4Sensing fixed infrastructured node, showing the two stream transformation stages: *data sourcing* and *global aggregation*

1) Query Distribution: To reach all relevant data, a query has to be distributed to all nodes within its *search area*, defined as the union of quadrants, at the minimum division level, that completely cover the query area. This is illustrated in figure 5, where the shadowed quadrant represents the search area for query Q.

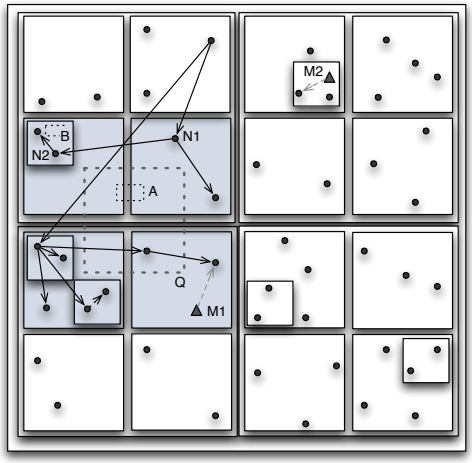

Figure 5. QTree spatial partitioning with a minimum occupancy of 2 nodes and minimum division level of 2

A query is disseminated by recursively subdividing the search area until all nodes are reached, building a distribution tree in the process as depicted in figure 5. Initially, the root node divides the world into four quadrants, finds their interceptions with the search area and forwards the query to a randomly selected peer in each interception. Target nodes repeat the same procedure; for quadrants that do not have minimum occupancy, the node assumes the aggregation role

and forwards the query to all peers in the area - these nodes become the tree leaves, acting as data sources. QTree provides an inherent balancing mechanism, given that a different node is chosen each time the aggregation tree is built and, the wider the aggregation area, the larger the set of candidate nodes.

2) Query Processing: Aggregation is performed using the reverse path of the query dissemination tree. An important aspect of QTree, that allows the reduction of computation and communication costs, is that aggregate tuples are *complete* 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 - and can be forwarded to the query root without further processing. In figure 5, extents *A* and *B* are complete 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 enclosed at lower levels of the aggregation tree.

3) Network Dynamism: Node failures in an aggregation tree impact query results, the severity of this impact depending on how close to the root the failure occurs. Failure requires rebuilding the tree for the affected quadrant, possibly after merging imposed by the minimum node occupancy requirement. When a node joins the network during query execution it can be incorporated into the aggregation tree after any necessary subdivision of space and consequent tree restructuring.

IV. Evaluation

The system presented in the previous sections has been evaluated with two goals in mind. Firstly, to obtain a initial assessment of the expressivity of the proposed programming language abstractions and, secondly, to evaluate in quantitative terms the performance of the distribution strategy that has been adopted. To this end, we modeled and implemented a case-study application, in a simulation setting, focused towards realtime participatory sensing data processing. Namely, this application, called SpeedSense, continuously monitors the current traffic status in an urban setting to allow client applications to access the current traffic speed per road and information about congested roads. For this purpose, simulated users collect real-time data while driving, using GPS equipped mobile devices. While, at this point, this case-study does not intend to be a realistic implementation of road traffic estimation, the scenario involved is a paradigmatic one for Participatory Sensing and is featured in some of the most referenced works in the area [10], [14].

A. Case study - SpeedSense

SpeedSense infers the current average speed, and congestion detections in a given area, based on GPS data sampled by in-transit vehicles at periodic intervals. For this purpose, two virtual tables have been designed: TrafficSpeed, which supports querying for average speed per road segment, while the other, TraficHotspots, allows for querying for congestion detections.

For the road network (and traffic) model, SpeedSense requires a map representation of the application's geographic area of coverage. The Open Street Map (OSM) [15] vectorial representation of the road network is used for that purpose. This map data is used to map geographic coordinates to road segments, to determine the spatial extent of segments and their associated road type. The network model used in the evaluation divides roads, as needed, into segments with a maximum of 1 km and uses separate segments for each driving direction. Each road is assigned an expected (uncongested) driving speed according to its type: highway, primary to tertiary and residential.

1) TrafficSpeed Virtual Table: 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 (cf. Definition 1). Specifically, the data sourcing pipeline stage handles local aggregation of raw GPS input data, by mapping incoming samples to road segment using the *process* operator (for simplification, raw GPS readings reference the segment identifier); a *timeWindow* accumulates data samples for the given time period, then *groupBy* partitions the resulting data into independent substreams for each segment. For each of these, *aggregate* accumulates data samples for the given time period to compute the intermediate sum and count results that are used to produce the actual (moving) average speed for that segment as an *AggregateSpeed* tuple.

The global aggregation stage receives *AggregateSpeed* values from descendent peers and produces the overall average speeds by merging the partial records. A *timeWindow* and operator *groupBy* are again used to accumulate data and partition the stream. For each of the resulting partitions (or substreams) *aggregate* is once more used for summing and counting all partial contributions and produce the actual *AggregateSpeed* value at this peer.

2) TrafficHotspots Virtual Table: This table outputs a stream of *Hotspot* tuples representing real-time detections of congested road segments. It is based on a simple model that compares the current average speed of a segment against a *congestion threshold*, given as a fraction of the maximum speed for that particular road, obtained from the road network model. It also takes into account the number of samples used to compute the average speed of the segment to provide a measure of the confidence or reliability in a detection result. Refer to Definition 2 for the actual specification of this table, where it can be seen that it derives from the TrafficSpeed table and, essentially, extends its global aggregation stage with the hotspot detection classifier. This *classifier* operator receives an *AggregateSpeeds* stream and produces a *Hotspot* tuple whenever the computed average is complete, reliable and below the congestion threshold.

Definition 1 TrafficSpeed virtual table specification

```
sensorInput( GPSReading )
dataSource {
    process{ GPSReading r ->
        r.derive(MappedSpeed, [boundingBox: model.getExtent(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')
        }
    }
}
```

Definition 2 TrafficHotspots virtual table specification

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

Definition 3 Q1 and Q2 query specification

```
def q1 = new Query("TrafficHotspots").area(
    minLat: 38.7379878, minLon: -9.1821318,
        maxLat: 38.758213, maxLon: -9.145832)

def q2 = new Query("TrafficHotspots").area(
    minLat: 38.727875, minLon: -9.2002818,
        maxLat: 38.7683250, maxLon: -9.1276818)
```

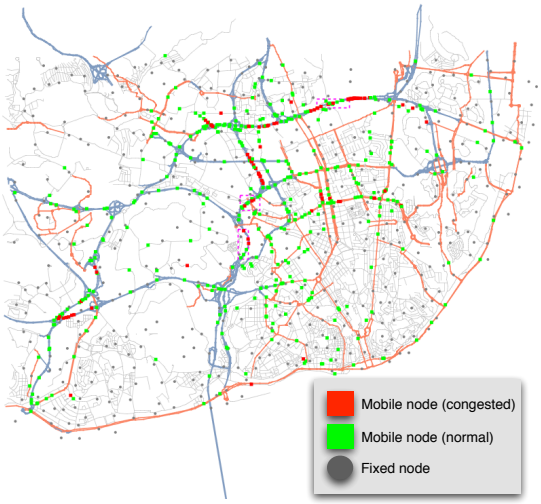

Figure 6. Snapshot of the traffic simulation showing congested mobile nodes in red

B. Evaluation

The SpeedSense evaluation was performed in a custom simulation environment of a fixed and mobile node infrastructure. Fixed nodes are distributed randomly across the urban space with a minimum inter-node distance of 250 meters. A one-hop overlay network connecting the fixed infrastructure is simulated through a shared common peer database that provides a consistent view of the network membership. The network is static i.e., membership is determined on startup and there are no node entries or exits during execution, or node failures. Mobile nodes interact with a fixed-node homebase counterpart directly, resulting in the delivery of raw GPS data with no latency. Communication between fixed nodes experiences latency and jitter. Each mobile node simulates a vehicle, according to a traffic model, and reports its GPS reading every 5 seconds as it follows the assigned path. A common clock is used to timestamp readings; thus, any effects of clock desincronization are not considered.

Traffic is modeled by emulating a fleet of vehicles driving through random routes. The maximum speed for a given segment is the same value used for congestion detection and depends on the road type. An average speed, for each road segment at a given time, is determined by its current car density and used to generate the random speed individually for each vehicle, according to a normal distribution. In the experiments performed, congestion occurs in segments with a density of at least 5 vehicles. 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 6 shows a rendering of the traffic simulation.

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, 500 nodes are used. In both cases, the mobile infrastructure comprises 500 mobile nodes.

Two queries were tested for each node density, Q1 and Q2, covering respectively 6.25% and 25% of the overall simulation area (cf. Definition 3). Both were placed on a high mobile node density area. The set of metrics captured was averaged over 10 runs, corresponding to different fixed node placements.

1) Workload Distribution: One of the purposes of the experimental evaluation was to determine how the effort required to evaluate a query is spread among the fixed nodes. For

that, workload is measured as the number of data acquisition and aggregation events occurring and processed at each fixed node. Specifically, the former pertains to the number of GPS sensor readings received and processed by the acquiring node, while the latter refers to the number of inputs handled by the global aggregation stage and corresponds to the updates received for each segment aggregated by that node. To derive a total workload at each node, the two are added with equal weights.

The experimental results obtained for Q1 and Q2, in both low and high density node scenarios, are presented in Figures 7 to 12.

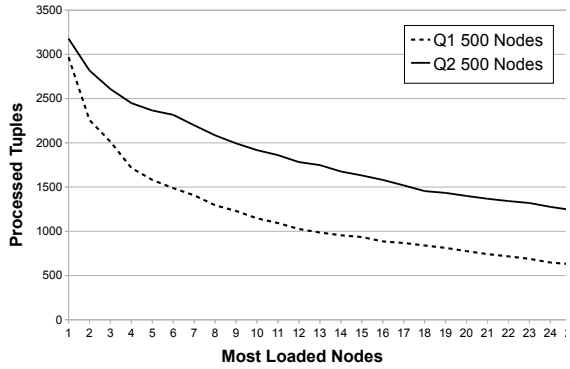

Figure 7. Total workload for queries Q1 and Q2, in high density scenario, covering 6.25% and 25% of the target area, respectively

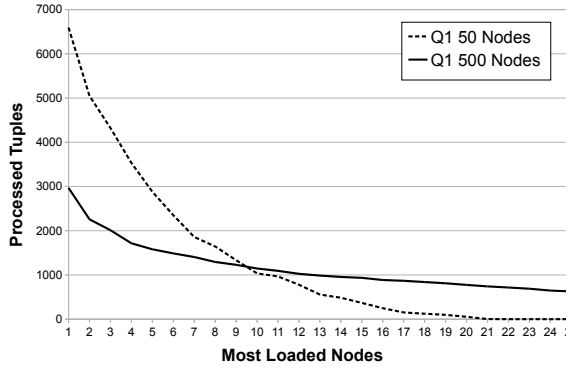

Figure 8. Total workload for query Q1 in low and high density scenarios

QTree balances load more effectively in the high node density scenario, where the most loaded node handles 7.4% and 3.1% of the total work, respectively for Q1 and Q2. In this setting, the additional work introduced by Q2 is distributed among participating nodes and does not affect significantly the maximum workload. Lower node density affects QTree negatively - in this case, the most loaded node handles 19.2% (Q1) and 12.5% (Q2) of total work, and Q2 results in a significant increase of the maximum workload (57.7%), relative to Q1.

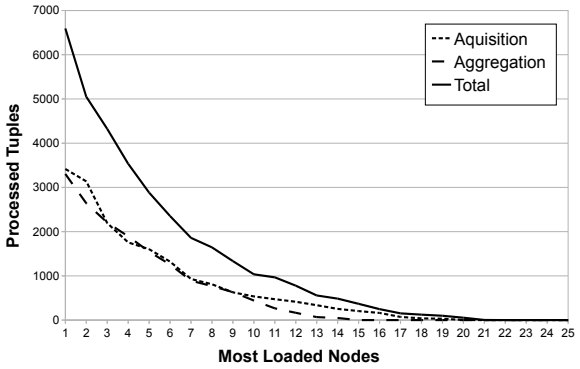

Figure 9. Workload decomposition for Q1 in low density scenario

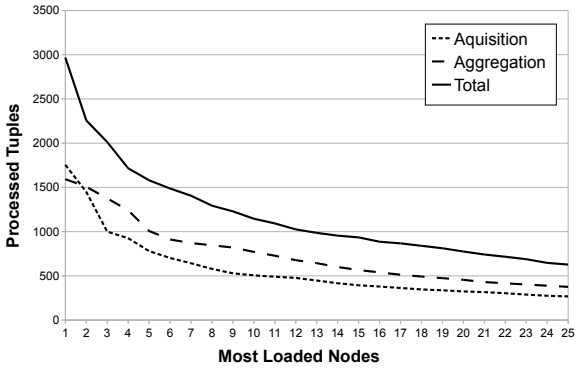

Figure 10. Workload decomposition for Q1 in high density scenario

2) Query Success and Latency: Query success is given by the percentage of accurate detections, including false negatives, 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) were not considered for the evaluation. The results are only indicative, as the traffic patterns produced by the traffic model are highly dynamic compared to real world conditions, with several short lived congestions occurring during query evaluation. Figure 11, which plots query success versus detection latency for both queries, shows a success rate in excess of 90% within the 120 second allowed window. Moreover, it shows that detections take longer on average in the high node density scenario, which can be explained by the difference in the aggregation tree depth that is needed in that case.

3) Communication Load: Cost measurements were also made regarding the number of messages exchanged during the execution of a query, providing an indication of the expected performance in terms of network usage. This communication load includes data messages and binding events. The former accounts for tuples exchanged between peers, relative to

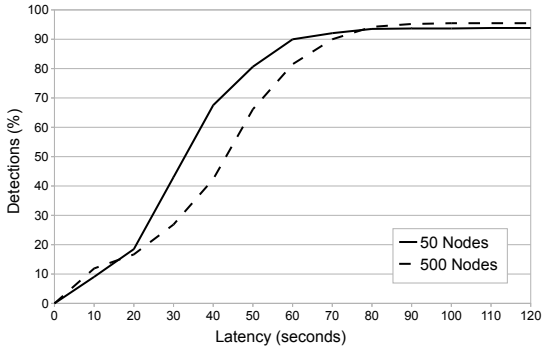

Figure 11. Query success and latency for low and high density scenarios

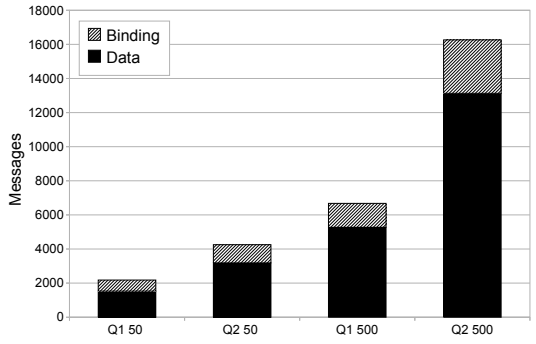

Figure 12. Messages exchanged for Q1 and Q2, in low and high density scenarios

query data (incomplete aggregations that are forwarded up the aggregation tree) and query results (complete aggregations that are forwarded to the query root. While the latter capture the additional overhead associated with uploading the data from mobiles to a fixed node.

In all cases, the number of messages sent by individual nodes is limited at 1 message per pipeline stage every 10 seconds by the use of the *timeWindow* operator, thus the exchanged messages reflect the number of nodes involved in query processing. It was observed, as shown in Figure 12, that executing the same query in different node density settings has a large impact on message traffic. Going from the low to the high setting results in an increase in the number of messages of about 200% for Q1 and 280% for Q2. The impact of the query area on message traffic is also high, in both low and high density settings - resulting in an increase of 95% for low density and around 140% for high density - for a 300% increase of the query area.

V. Final remarks

4Sensing abstracts application developers from the complexity inherent to a distributed infrastructure, such as the actual location of relevant data and the balancing of processing work, and supports common processing and aggregation tasks though a library of out-of-the-box components. Applications can share data through *virtual tables*, thus promoting the development of application mashups, while keeping control over the granularity of published data. Although the QTree distribution strategy is affected by the unbalanced distribution of sensed data, specially for lower node densities, the strength of the strategy resides in the ability to limit the propagation of partial state by leveraging

Directions for future research include the exploration of efficient delivery of query results to the mobile node base, and the optimized processing of multiple overlapping queries; finally we would like to investigate the aspects related to data persistence and historical queries.

References

[1] W. Bloomin. http://whatsbloomin.com, June 2010.

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

[3] D. Cuff, M. Hansen, and J. Kang. Urban sensing: out of the woods. *Commun. ACM*, 51(3):24–33, 2008.

[4] 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.

[5] J. Eriksson, L. Girod, B. Hull, R. Newton, S. Madden, and H. Balakrishnan. The Pothole Patrol: Using a Mobile Sensor Network for Road Surface Monitoring. In *The Sixth Annual International conference on Mobile Systems, Applications and Services (MobiSys 2008)*, Breckenridge, U.S.A., June 2008.

[6] A. P. 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.

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

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

[9] W. Grosky, A. Kansal, S. Nath, J. Liu, and F. Zhao. Senseweb: An infrastructure for shared sensing. *Multimedia, IEEE*, 14(4):8–13, Oct.-Dec. 2007.

[10] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. K. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, Boulder, CO, November 2006.

[11] N. Kotilainen, M. Weber, M. Vapa, and J. Vuori. Mobile chedar " a peer-to-peer middleware for mobile devices. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 86–90, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Y. J. L. Marie Kim, Jun Wook Lee and J.-C. Ryou. Cosmos: A middleware for integrated data processing over heterogeneous sensor networks. *ETRI Journal*, 30(5), October 2008.

[13] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 337–350, New York, NY, USA, 2008. ACM.

[14] Mohan, Prashanth and Padmanabhan, Venkata and Ramjee, Ramachandran . Nericell: Rich Monitoring of Road and Traffic Conditions using Mobile Smartphones. In *Proceedings of ACM SenSys 2008*, November 2008.

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

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