Information extraction from large-scale WSNs: approaches and research issues: approaches and research issues: part II: query-based and macroprogramming approaches

Daniel, T. and Gaura, E.

Sensors & Transducers Journal, Vol. 94, Issue 7, July 2008, pp. 34-56

www.sensorsportal.com

Sensors & Transducers

ISSN 1726-5479

http://www.sensorsportal.com

Information Extraction from Large-scale WSNs: Approaches and Research Issues
Part II: Query-based and Macroprogramming Approaches

Tessa DANIEL, Elena GAURA
Cogent Computing Applied Research Centre, Coventry University, UK CV1 5FB
E-mail: e.gaura@coventry.ac.uk, www.cogentcomputing.org

Received: 29 June 2008 /Accepted: 21 July 2008 /Published: 31 July 2008

Abstract: Regardless of the application domain and deployment scope, the ability to retrieve information is critical to the successful functioning of any wireless sensor network (WSN) system. In general, information extraction procedures can be categorized into three main approaches: agent-based, query-based and macroprogramming led. Whilst query-based systems are the most popular, macroprogramming techniques provide a more general-purpose approach to distributed computation. Finally, the agent-based approaches tailor the information extraction mechanism to the type of information needed and the configuration of the network it needs to be extracted from. This suite of three papers (Part I-III) offers an extensive survey of the literature in the area of WSN information extraction, covering in Part I and Part II the three main approaches above. Part III highlights the open research questions and issues faced by deployable WSN system designers and discusses the potential benefits of both in-network processing and complex querying for large scale wireless informational systems. *Copyright © 2008 IFSA.*

Keywords: Information extraction, Query based systems, Macroprogramming

1. Introduction. Overview of Querying and Macroprogramming Approaches to Information Extraction from WSNs

This article is the second part in a three part series surveying information extraction approaches to large scale WSNs. For the reader's benefit, the introductory section from Part I is repeated here.

Regardless of specific application requirements, acquiring data and generating information is at the core of any WSN deployment. A wealth of literature exists, proposing numerous approaches to extracting information from WSN systems. Whilst some such approaches are suitable for small scale networks (up to 20-30 nodes) and have indeed been validated through implementation and deployment in real-life scenarios, many other proposals are at theoretical level, have been evaluated through simulation only, and are yet to be further studied to reach deployment stage. The latter studies are particularly concerned with methods and tools for extracting information either from very large scale networks or have a genericity constraint at the core of the approach.

Starting from a view that most deployed or theoretical information extraction approaches fall into one of three categories (agent-based, query based or macroprogramming based approaches), this paper surveys both query based and macroprogramming based systems, with a view to firstly identify which of the methods proposed in the literature have been evaluated at implementation level. (This analysis is needed as a large proportion of the research effort is at theoretical level and not readily suitable for practical deployments.) Secondly, looking out to common constraints shared by WSN systems (such as limited energy and communication resources for example), it is aimed here to identify which of the methods proposed in the literature make use of or support in-network information extraction. (This is important as approaches that utilize in-network processing like aggregation and filtering have been shown to be more energy efficient and therefore more desirable.) (Note that agent based systems have been extensively surveyed in Part I of this suite of papers).

Whilst agent based techniques potentially offer a range of advantages for large scale WSN systems (pending further development beyond the present state-of-the-art), most of the WSN information extraction approaches currently in operation are based on applicative query mechanisms. In this approach, requests are initiated via queries written in an appropriate declarative language, posed to the network, and data or information generated as a result. Data processing in such applications usually follows one of two approaches: centralized or distributed. For some systems, resources are not as severely constrained, hence, minimizing energy usage is not a major concern. In such unconstrained systems, a centralized approach to processing is often used where sensed values are pushed to a power-rich location for processing, which may involve cleaning and querying the data as part of more in-depth analysis.

In a large number of systems, however, constraints like computing and battery power dictate that applications be developed with energy-efficiency in mind. For many of these applications human intervention can be both time consuming and expensive, for example, in terms of changing batteries or adding new nodes. Therefore, a key design objective is to extend the lifetime of the network as much as possible. It is well-understood that power usage costs in WSNs are dominated by communication as opposed to computation [1, 2, 3]. Therefore, techniques which promote a decrease in communications while using in-network data reduction have been identified as key to creating more energy-efficient WSN applications. Distributed processing is proposed as a method that promotes processing of data on nodes in the network [2, 4]. This in-network processing takes advantage of nodes' processing power and may include techniques like data aggregation, fusion or elimination of redundant values through filtering [5, 6]. The net result is more energy-efficient applications since data transmission is reduced in exchange for in-network computation. Whilst this is apparently of obvious benefit towards long -life systems design, in-network computation is still in its infancy.

Query-based systems are, undoubtedly most popular mainly because they provide a usable, high level interface to the sensor network while abstracting away some of the low level details like the network topology and radio communication. They are very useful and provide a solution in cases where data needs to be retrieved from the entire network. With this ease of use, however, come a number of limitations.

The first limitation is in terms of what queries can be posed to the network. In general, the query-based applicative systems in use allow the issuing of restricted queries, ranging from those targeting raw sensor readings on nodes in the network to those requiring the computation of simple aggregates like average, maximum, and minimum over some attribute of interest, for the entire network. Second, the query languages used cannot easily express spatio-temporal characteristics which are an important aspect of the data generated in WSNs. Third, it is quite difficult if not impossible to construct information requests that represent higher level behavior, or involve just a 'subset' of the network (whether physical or logical) or require more complex in-network interactions between 'subsets' in order to generate information. Furthermore, as distributed computation is not the main focus of SQL-based query languages (which support most WSN query-based approaches) implementing arbitrary aggregation, for example, is quite difficult [7].

In contrast, *macroprogramming* has been proposed as an approach to information extraction that provides a more general-purpose approach to distributed computation compared to traditional query-based approaches. As applied to WSNs, macroprogramming approaches focus on programming the network as a whole rather than programming the individual devices that form the network. Many macroprogramming systems provide the ability to create programs that represent higher level behavior, a level of abstraction beyond that of the more popular query-based approaches. Global behavior can be specified, programmed and then translated to node level code. Ideally, the programmer is not concerned with low level details like network topology, radio communication or energy capacity.

Of interest with some macroprogramming systems are the application-defined, in-network abstractions (some based on local node interactions) that are used in data processing. One example is the Regiment system [8] in which a programming abstraction called a region is used. A region is described as a collection of spatially distributed signals with an example being the set of sensor readings from nodes in a geographic area. Regions as opposed to individual nodes are programmed (for example, an rfold operator is used to aggregate the values in a region into a single signal which can then be communicated to the user or used in further computation).

Macroprogramming, however, still presents a number of challenges. First, creating a powerful macroprogram requires a learning curve for the programmer. Expressing high-level requirements are not necessarily as intuitive to the user as perhaps SQL-based approaches are. Second, although proposed as an approach that eliminates the need for node-level programming, many of the current macroprogramming systems provide node-specific abstractions, undermining the rapid development and productivity advantages macroprogramming is meant to provide. Finally, because of the wide semantic gap that exists between the high level program and the node level code, compiler construction is quite challenging. The code generated as a result of compilation has to cater for not just computation but node-level communication as well.

This paper is structured as follows: Section 2 surveys the area of query based information extraction providing example systems and weighing their benefits; similarly, Section 3 looks at macroprogramming approaches and sample developments or successful deployments; Section 4 concludes the paper drawing comparisons between the three main approaches to extracting information and opens the stage for the last paper in the suite, Part III. Part III looks at the open research issues the community faces with respect to information extraction, raises research questions relating to the usefulness of in-network processing from an informational viewpoint and concludes the survey.

2. Query Based Information Extraction

Many researchers have taken the view that the sensor network can be considered a database from which information has to be requested and retrieved. The reasons for this are as follows: first, the

network is a collection of data albeit that data is dispersed across multiple nodes. Second, data needs to be retrieved from the network, and like a conventional database one needs to be able to query the network for what is desired and get a response. Third, the nodes in the network are of interest primarily in as far as the data they generate, this can also be termed a *data-centric* view of the WSN.

Many of the *sensor network databases* use a query language similar to the Structured Query Language (SQL) for constructing queries.

SQL is the most popular language used for creating, modifying and retrieving information from relational databases. It consists of a number of clauses including `SELECT - FROM - WHERE` and `GROUPBY` clauses which allow selection, join, projection and aggregation respectively. SQL-type queries constructed for WSNs are quite similar to those of traditional SQL both syntactically and semantically. The FROM clause, for example, can be used to either specify sensors or data stored in tables. The `GROUPBY` clause allows a `SELECT` statement to collect data across multiple records and group the results by one or more attribute values. In the WSN querying context, for example, records can be grouped by the node's id or by locations, etc. if these are attributes retrieved in the query.

In traditional database systems, data is accessed via an application or directly via a front end. Such applications insulate the user from the inner workings of the database system while allowing easy and meaningful queries to be applied to it in retrieving the necessary information. In a similar way, the sensor network can also be interfaced using a suitable application. The application would act as a query processor-type interface to the network taking into account the power and computation resource limitations on the devices in the network. According to [9] the challenge in query processing is not in processing data as quickly as possible but rather in figuring out a way to effectively respond to queries while transmitting as little data as possible. Several researchers have noted the benefits of this query-processor interface approach.

[10] describe a view of the network as a database with two methods for processing queries issued to the network: warehousing and distributed approaches. With the warehousing approach the processing of a query is separated from the interaction with the network itself. Essentially, the data is extracted and stored at a centralized location for processing. Queries in this case are pre-defined and usually ask for aggregate historical data, for example, '' for each rainfall sensor, display the average level of rainfall for 1999'' [4]. This model really mirrors a client/server approach to data collection as is found in a traditional distributed network [11].

There are two main disadvantages to this approach, however. First, it is not well suited to requests for continuous data, either because it is not possible to retrieve the required data from the node or because data is not retrieved frequently enough within the network to be able to answer a query [4]. Second is the high energy consumption that occurs when transmitting large quantities of data from the node to the centralized database and even more so in the case of a continuous stream of data, making the process very energy-inefficient [4, 10].

Alternatively, [4, 12, 10] propose a view of the network as a set of distributed databases where the composition of the query in effect determines what data is retrieved. Such queries have the advantages of efficiency because only what is required is actually retrieved, as well as flexibility since various queries can be formulated depending on what information is needed. This approach also takes advantage of the processing power on the node itself and where possible processes queries or parts of queries on them.

[13] note that data generation and routing in the sensor network can be seen as similar to the concepts of data storage and query processing in traditional databases and so also view the sensor network as a

database. They examine the challenges in implementing the network as such and identify the need for robustness of data access given the possibility of node failure and noise that can affect readings. The idea is that by creating a standard querying interface that allows less restrictive query semantics, approximate results can be obtained which can help in producing an energy-efficient implementation of the 'sensornet database application'. [13] are also of the view that query optimization is closely linked to routing and they propose an adaptive approach that takes into account the volatile nature of data and communication in the network.

Several major query based systems are described in detail below.

2.1. COUGAR

The COUGAR approach has been proposed in [4, 12, 10, 5] and more fully described in [14]. It builds on the Cornell PREDATOR object-relational database system [15] and models each sensor in the network as an abstract data type while signal processing functions are modeled as abstract functions returning sensor data. The network is viewed as a system where each node is a 'mini database' holding part of the data contained by the whole. COUGAR allows the issuing of declarative queries for information requests and uses a query optimizer to plan an in-network processing strategy. Queries in Cougar are SQL-based and are posed to a virtual table called sensors with one row per node per instant in time and one column for every attribute. (The same model is used in TinyDB and is described in Section 2.5).

A query proxy layer is placed on each node and interacts with both the routing and application layers in the system. A query optimizer is placed on a gateway node and distributes the query processing plans after it receives the queries from the user. COUGAR uses catalogue information and query specification to create the query plan which describes not only the flow of data between sensors but how information is to be computed on individual sensors. Once the plan is complete it is sent out to all relevant nodes. During execution, data is returned to the gateway node. In addition, a query proxy can inject queries into the network from arbitrary or specified nodes as it performs high-level services. Fig. 1 shows how the system executes a simple aggregate query. COUGAR is especially useful for executing continuous queries.

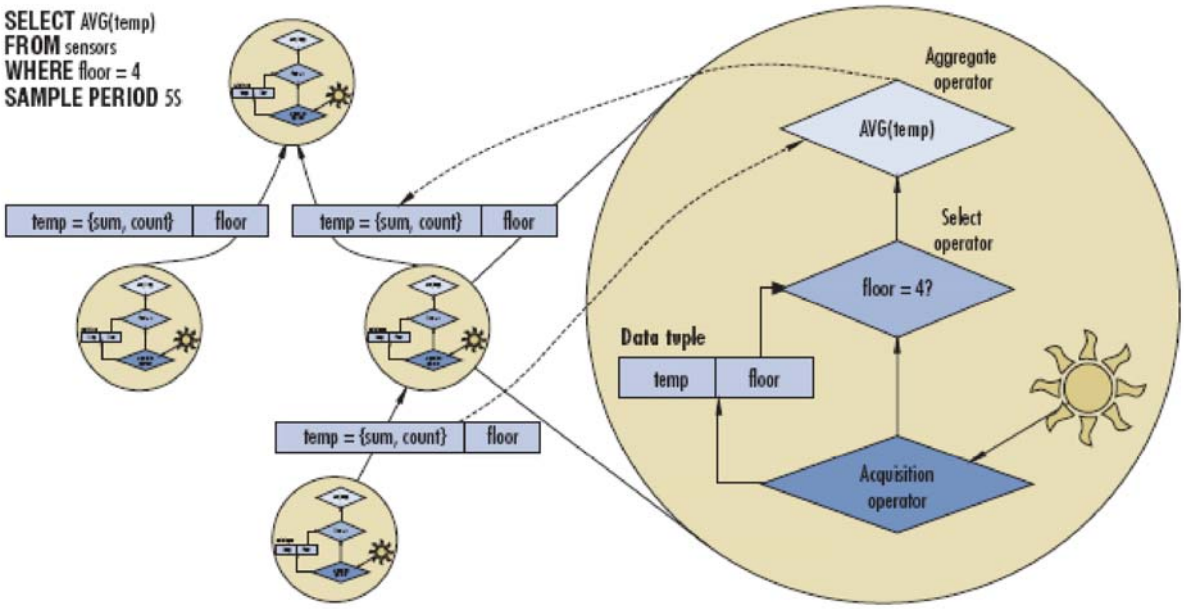

Fig. 1. A sensor network running the Cougar system executes a simple aggregate query taken from [16].

A big advantage of COUGAR is that it shields the user from needing to have any knowledge of the underlying network, or how data is generated and processed. A second advantage is that it increases efficiency in terms of energy consumption since in-network processing is allowed thereby reducing the amount of data that has to be sent back to the gateway node [17]. There are other architectures which promote this distributed approach to query processing, for example, IrisNet, described in [18], SINA (Section 2.2) and TinyDB (Section 2.5). Both TinyDB and Cougar follow a very similar query processing model and address the resolution of both simple and aggregate queries. Neither, however, facilitates the processing of other types of complex queries such as spatio-temporal or queries which target only a subset of the network.

2.2. Sensor Information Networking Architecture (SINA)

Like COUGAR, the SINA architecture described in [19] promotes a distributed database query interface that emphasizes in-network processing and reduction in power consumption in the network. It views the network as a collection of datasheets (a spreadsheet) with each datasheet containing the attributes relevant to the node it describes. A cell in this scheme represents an attribute and the collection of datasheets form what is called an associative spreadsheet or spreadsheet database representing the network. Each cell is therefore referred to using an attribute-based naming method.

SINA also makes use of clustering of low-level information in the network to increase efficiency. Clusters are formed from aggregations of sensors based on their power levels and proximity. Recursive aggregation can also be used to produce a 'hierarchy of clusters'. A cluster head within a cluster can then be elected to perform key functions like information filtering, fusion as well as aggregation [19]. Fig. 2 shows a model of a network with the SINA middleware.

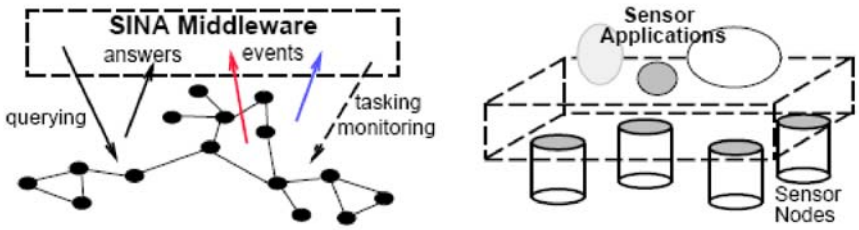

Fig. 2. Model of a sensor network and SINA middleware taken from [19].

SINA uses the sensor programming language Sensor Query and Tasking Language (SQTL) which serves as the programming interface between applications and the middleware. SQTL is a procedural scripting language built from Structured Query Language (SQL) and messages written in the language can be interpreted and executed by any node in the network, although messages can target specific nodes if required. SQTL also supports the generation of information as a result of the occurrence of, or change in, some phenomena (events). Three types of events are supported by SQTL. The first is an event generated when a message is received by a node, the second is an event triggered periodically by a timer, and the third is an event triggered by the expiration of a timer.

SINA modules run on each node in the network and allow querying as well as monitoring of events. Nodes are aggregated to form clusters and elected cluster heads perform filtering, fusion and aggregation tasks. The user issues a query and the SINA architecture selects the most suitable methods for information gathering and distribution based on the type of query issued as well as the current

status of the network. A node receiving the query will interpret it and request information from neighbouring nodes in evaluating it.

A number of information gathering mechanisms are employed to help reduce resource consumption and increase response quality, as follows:

1. **Sampling Operation** - for some applications a query may need to target the entire network in eliciting information of interest. Responses from all nodes in a network may result in response implosion [20], however, the effect can be reduced if nodes are able to make decisions on whether they should respond or not. SINA implements this 'decision-making' capability by assigning each node a *response probability* which determines whether they respond or not.
2. **Self-Orchestrated Operation** - in networks with a small number of nodes it is sometimes necessary that all nodes respond in order to improve the accuracy of a result. SINA uses what is called self-orchestration in an effort to reduce the problem of response implosion mentioned before. Here each node suspends its response transmission for a particular period of time to help reduce the likelihood of collisions occurring as could happen if nodes transmitted simultaneously.
3. **Diffused Computation Operation** - this method is used to implement aggregation functionality in the network. Each node is first assumed to have knowledge of its immediate neighbours and aggregation logic already programmed into the SQTL scripts is used to perform aggregation before routing the result to a designated node.

SINA is therefore particularly suited for aggregate and queries for replicated data in a cluster-based network configuration but does not address other complex queries like spatial or temporal queries which may also be suited for processing in a cluster-based configuration.

2.3. Data Service Middleware (DSWare)

Like SINA and Cougar, DSWare [21] is a data-centric middleware approach to providing information retrieval services within a WSN. DSWare, however, is aimed at handling real-time events and integrates a number of real-time data services while providing a database-like abstraction. In this way it can be considered another database-like approach adapted to sensor networks. It provides services for reliable data-centric storage and implementing alternative methods for improving real-time execution performance, as well as reliable data aggregating and decreased communication which are aimed at conserving the limited resources available within the network. Its architecture separates the routing layer from the DSWare and network layers as DSWare provides components for improving power-awareness and real-time-awareness of routing protocols (Fig. 3).

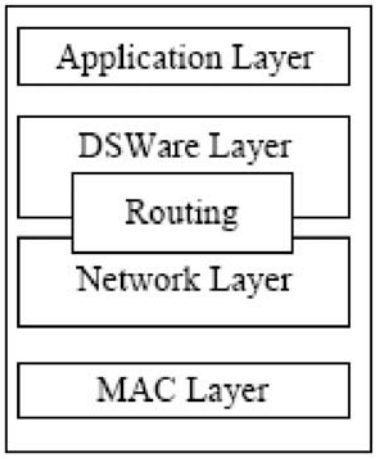

Fig. 3. DSWare architecture, taken from [21].

There are six services/components available in this middleware approach (Fig. 4):

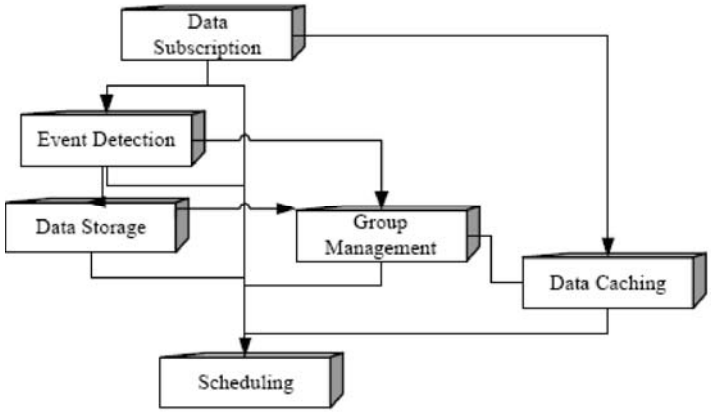

Fig. 4. DSWare Framework taken from [21].

1. **Data Storage** - this component allows storage of location-linked data so that future queries for similar data would not mean having the query re-issued or flooded through the entire network. In DSWare data lookup is implemented using a hashing function that maps data to physical storage nodes via a unique identifier while robustness is facilitated by using replicating needed data in several physical nodes which can then be mapped to a single logical node. Queries targeting that data can be directed to any one of these nodes in an effort to avoid collision and sustained overload on a single node.
2. **Data Caching** - this service provides copies of data that is requested often and spreads it through the network using the routing path. This serves to accelerate execution of the query while reducing communication aimed at accessing data for query processing.
3. **Group Management** - this component facilitates cooperation among nodes to accomplish more complex tasks. In some cases, for example, requested information requires multiple sensors combining their values to calculate a result. The group facility therefore can create a group when a query is issued. Nodes receiving some information on this group can decide whether they match the particular criteria described for that group. The group then manages the values of relevant sensors in accomplishing the given task after which the group is dissolved. In some cases, the query itself may expire before the group can be dissolved.
4. **Event Detection** - DSWare is based on event detection, an event being an activity that can be monitored or detected in the environment and is of interest to the application. An event is pre-registered depending on the application and can be classified as either atomic events (events that can be determined based on the observation made by sensor) or compound events (events than cannot be determined directly but rather need to be inferred from other atomic or 'subevents'). Like COUGAR, DSWare uses statements in a SQL-similar language for registering and cancelling events. After parsing the statement defining the event, DSWare generates the query execution plan and calls on the methods required to register, execute and cancel the event.
5. **Data Subscription** - this service is used to aid in more efficient data dissemination. If multiple nodes request the same data the subscription service puts copies at intermediate nodes which can then be accessed by the requestor nodes. This helps in reducing communication and helps conserve resources within the network.
6. **Scheduling** - this component provides a real-time scheduling mechanism as the default method with the option of adding on an energy-aware mechanism if the first has already been successfully implemented. All components of DSWare are scheduled using this component.

DSWare, because of its emphasis on providing event detection services and the mechanisms it provides for data caching, group management and data subscription, is particularly suited to aggregate queries as well as queries for replicated data. These aggregate queries, however, are simple aggregates over an attribute and do not include more complex aggregate queries where data from multiple clusters, for example, could be combined to resolve such queries.

2.4. Framework in Java for Operators on Remote Data Streams (Fjords)

The Fjords architecture presented in [22] shows how processing multiple queries can be managed over multiple sensors while allowing more efficient resource usage and keeping query throughput high. The focus in the research is on creating the underlying architecture that will support the processing of multiple queries over sensor data streams.

The architecture comprises operators configured for streaming data and Fjord operators called sensor-proxies whose function is to serve as a mediator between the query processing plan and sensors. Data is allowed to flow into the Fjord from the sensor and then pushed into the query operator. Query operators are not active pulling in data but rather wait till data is sent to them from the sensors. The fjord, in addition to combining streaming and fixed data also processes multiple queries and combines them into a single plan. Fig. 5 shows the query environment.

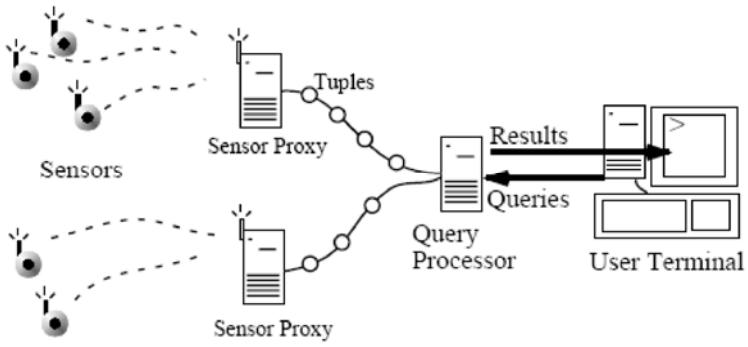

Fig. 5. Fjords query environment taken from [22].

The Fjords architecture is therefore suited to continuous queries and addresses a number of issues important to query processing. It does this in two ways. First, by using proxies, non-blocking operators and query plans which allow streaming data to be pushed through operators that pull from data sources, it allows merging of the both stream data and local data. Second, by letting proxies serve as mediators between query plans and sensors it allows query processing while at the same time taking into account power, communication and processor restrictions in the network.

The work here is similar to the COUGAR project in that they both focus on processing of streams of sensor data, however, COUGAR although it does incorporate in-network processing in the form of data aggregation, it does not focus specifically on energy efficiency and the resource constraints on sensor devices but rather on modelling the streams of data using abstract types. Fjords, however, look more closely into improving efficiency for processing of data streams. Although a viable approach for continuous queries it does not address complex queries or even aggregate type queries to any great extent and is more concerned with managing simple query processing over multiple sensor data streams.

2.5. TinyDB

All the approaches to query processing described above have one thing in common: they view query processing in the network as a modified version of that in traditional databases. In essence, each node in the network is seen as a generator of named data against which queries can be issued. [23] describe a distributed database approach that attempts to improve energy consumption by applying varying techniques for aggregation and optimization within the network. This aggregation service is called Tiny Aggregation (TAG) and is designed specifically for TinyOS [24] motes. TinyDB has been one of the more widely used and popular query processing systems and considerable work has gone into extending its capabilities both in terms of the types of queries that can be issued as well as improving its operation by integrating different routing protocols. TinyDiffusion [25] is one such example. It has been widely used in a number of real-life deployments as well as inspired the research into other query processing systems and hence will be discussed in full in this section.

TinyDB uses acquisitional query processing (ACQP) as described in [26] for in-network query processing. In terms of practical applications ACQP focuses on the location and cost of accessing data as a way of reducing power consumption. There is no assumption made that data exists at a particular location rather queries are only issued where it is known that data exists.

TinyDB can therefore be considered an ACQP engine, a distributed query processor that runs on all nodes in the network. TinyDB was designed to be deployed on the Berkeley Mica mote platform which runs the TinyOS operating system (Mica motes are arguably one of the most popular of WSN hardware platforms.)

Query Syntax: Queries in TinyDB are quite similar to SQL with the FROM clause being used to either specify sensors or data stored in tables (called materialization points).

Sensor tuples are stored in a table (*sensors*) with one row per node per instant in time and one column representing each attribute. An attribute could be light, temperature, sound, etc. Records in this table are stored for a short period usually and only ‘acquired’ when needed to resolve a query. An example of a query could be:

```
SELECT nodeid , light
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

This query states that each node should return its own id and light reading from the *sensors* table once per second for ten seconds. The results are either returned to the user or logged. The sample period is used to indicate when data collection should begin. The sensors table is an unbounded, continuous data stream of values and can only be sorted or used in joins via a specified *window*.

Time Synchronization: TinyDB adopts the Timing-Sync protocol presented by [27] which attempts to provide network-wide time synchronization in the sensor network. The algorithm proceeds in two stages: the **Level Discovery Phase** establishes a hierarchical topology for the network and occurs at the point of network deployment. In the second phase, the **Synchronization Phase**, a two way message is used to synchronize two nodes. The time synchronization process begins with the root node first sending a time-sync packet after which receiving nodes initiate the message exchange with the root node as described above in the synchronization phase. Once the nodes receive acknowledgment from the root node they adjust their clocks to that of the root node. This process is carried out until all nodes are synchronized with the root node.

Aggregation Queries: TinyDB supports aggregation of data which in turn allows reduction of data prior to transmission. The main difference between such queries in TinyDB and SQL is that the output to a query is a stream of values for the former while it is an aggregate value for TinyDB. The aggregate operators allowed are the same as that allowed in a normal relational database system, for example AVG, MAX, MIN which compute the average, maximum value and minimum values respectively for the attribute they are applied over.

Temporal Aggregates: The TinyDB system acknowledges that aggregates over values within a common sample interval are not the only type of aggregate values that a user may want. In some cases a user will need to perform some type of temporal calculation. For example, users of a habitat monitoring system may want to monitor the temperature as a frost moves through a geographical area. This can be done by measuring the maximum temperature over a period of time and reporting that temperature at set intervals. In TinyDB this can be implemented as a *sliding window* query, for example:

```
SELECT WINAVG(TEMP, 60s, 5s)
FROM SENSORS
SAMPLE PERIOD 1s
```

The above query would report the average temperature over the last 60 seconds every 5 seconds with a sampling rate of once per second.

Event-Based Queries: TinyDB also supports event-based queries which are generated either by another query or by some part of the operating system, the language provides an EVENT clause for that purpose and the code that generates the event is compiled into the node [28].

Continuous Queries: Continuous or lifetime-based queries are also supported using the LIFETIME clause as well as nested queries and offline delivery queries which allow data to be logged for non-real time delivery. Materialization points are used to implement offline logging.

Query Optimization: Queries in TinyDB are first parsed at the base station and then optimized to select the ordering of joins, selections and sampling. The optimizer determines the lowest power consumption which takes into account not only the cost of data acquisition but also processing and radio communication. Data acquisition, however, is the main source as it includes actions like sampling of sensors and transmission of query results. Query optimization, therefore, is focused on reducing the number and cost of data acquisition. Once a query has been optimized it is issued into the network in a binary format, where it is instantiated and executed.

Query Dissemination and Routing: TinyDB uses semantic routing trees (SRT) to help determine whether a query is forwarded or not.

Query Processing: After a query has been optimized and disseminated it is executed by the query processor. Execution follows a simple sequence of sleep, sampling, processing and delivery. Nodes sleep for as much time in an epoch as possible to conserve power and wake up to sample sensors and deliver results. The nodes are synchronized so parent nodes can ensure that child nodes are awake to process messages. Results are sampled and filtered based on the plan provided by the optimizer and then routed to the aggregation and join operators further up the query plan. Fig. 6 shows the TinyDB architecture.

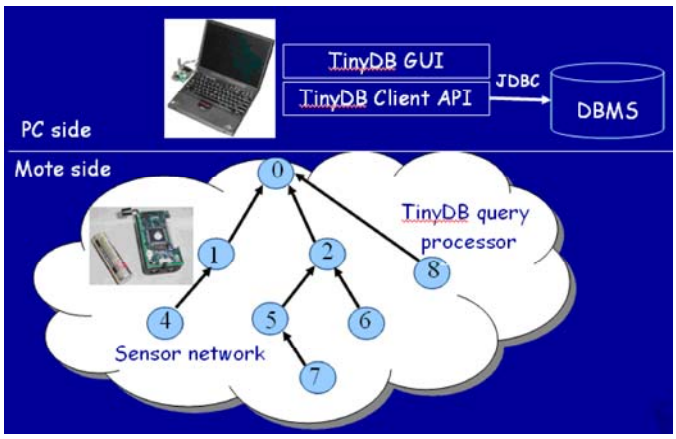

Fig. 6. TinyDB architecture taken from [29].

In summary, the TinyDB approach does allow quite a good deal of flexibility in the types of queries that can be processed and shows increased efficiency in terms of power consumption using a number of query optimization techniques. Some further work needs to be done on optimization of multiple queries as these are not addressed as well as implementing more sophisticated data delivery prioritization techniques. The authors stress, however, that with TinyDB using the ACQP method it is essential that data delivery, query language and optimization techniques take into account the underlying hardware capability and semantics in order to increase the likelihood of successful deployments.

Add-ons aimed at simplifying the deployment and development of applications for wireless sensor networks, are also being developed. The Tiny Application Sensor Kit (TASK) is built on top of TinyDB for that purpose [30]. This kit contains a relational database used for storage of sensor readings, a server to act as a proxy for the network on the Internet as well as a front-end that facilitates data selection and recording. Other architectures with a similar purpose also exist, for example jWebDust described in [31].

Although useful, like the other query-based approaches in use TinyDB has its drawbacks. The queries that can be issued are limited to those that target ‘low level’ sensed values on nodes in the network or those requiring the computation of simple aggregates like average, maximum, and minimum over some attribute. It does not allow the creation of nested queries or even nested, aggregate queries. In addition, the query language used cannot easily express spatio-temporal characteristics which are an important aspect of the data generated in WSNs. Therefore, there is still room for improvement and expansion in a query processing system that follows the same model but able to issue and process more powerful queries.

2.6. Active Query Forwarding (ACQUIRE)

Common to all of the query-based approaches described above is the clear distinction between the dissemination phase when the query is sent out and the response stage when results are sent back to the querying node. ACQUIRE described in [32], does not distinguish between these two stages but rather issues what is called an *active query*.

An active query, in addition to the simple queries for an attribute or attributes, also includes nested queries. Each subquery can be a query of interest in a different variable or value. The active query is propagated through the network node to node. At any point in time, an active node (the node carrying the active query) uses data from a set of neighbouring nodes within a look-ahead of *x* number of hops

and tries to resolve the query or part of the query if it can. The number of hops used can vary, but has been analyzed experimentally to determine what the most effective value is. The experiments in effect measured the average number of nodes from which new information is obtained during query forwarding based on different values for *x* and set that as the *effective look-ahead*. Fig. 7 gives an illustration of the ACQUIRE mechanism at work.

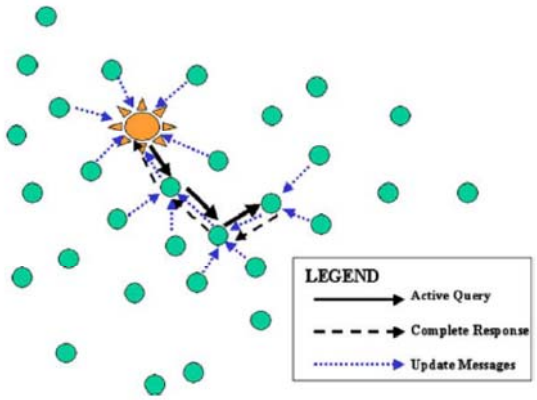

Fig. 7. Illustration of the ACQUIRE mechanism given a one hop lookahead. At each stage of the active query propagation the node carrying the query uses information gained to partially resolve the query, taken from [32].

Analysis of ACQUIRE has been restricted to networks exhibiting a regular grid topology. Given these restrictions, however, ACQUIRE worked better than flooding-based querying (FBQ) mechanisms like Directed Diffusion as well as had a 60-75% savings in energy consumption when compared to Expanding Ring Search (ERS) [32]. ACQUIRE is most suited to complex one-shot queries for replicated data and needs to be analyzed on more realistic networks where the topology is irregular or dynamic. The reported evaluation of ACQUIRE is further limited as it uses mathematical modelling to analyze the performance of the mechanism in terms of energy costs. It does not address implementation. Further, the complex queries although including nested queries do not include queries where spatial or temporal characteristics may be of interest.

3. Macroprogramming Approaches

In the applicative query approaches described above, the user must have some knowledge of what type of queries or operations are to be issued as well as what raw data needs to be targeted in order to retrieve the information. Although information is retrieved, in many cases, the user still has to make sense of what it means and whether it is relevant or not. For example, a user could issue a query in TinyDB requesting the average temperature over the network for a particular period of time. Once the results are received, however, the user would then have to look at this ‘information’ in deciding whether it indicates that a fire has started. If it does, the user then has to decide what to do next, for instance, issue additional queries. The ability to create a ‘program’ where events are anticipated and response (further queries) issued is not allowed in the system.

Macroprogramming approaches to information extraction aim to treat the wireless sensor network as a whole by directly specifying global behavior instead of programming individual nodes. It attempts to address the issues raised above in the form of a global program that is defined in a high level language that can capture operations going on in the network at a global level. A number of systems have been developed which incorporate global abstractions and in some cases couple them with node-level

abstractions in creating useful macroprograms. In this Section some of the more popular macroprogramming systems will be examined.

3.1. Node-level Abstractions

The underlying principle of macroprogramming is the creation of a global program that can capture network level operations. These macroprograms ultimately have to target node level data in order to function globally. A number of node-level abstractions have been proposed in the literature for modelling node level data and are used by macroprograms to target the data within the network. The data is not accessed on a low level, node by node basis but through these abstractions that provide access via logical 'collections' of nodes. The logical node-level abstractions proposed are usually based on one or more attributes within the WSN. This can be a dynamic attribute like a sensed value (temperature, for example) or a static attribute like geographic location (in the case of stationary nodes). Following is a description of some of the more popular node-level abstractions put forward in the literature.

Hood: Whitehouse et al. [33] describe a neighbourhood abstraction called a *hood* (which is defined by a set of criteria for selecting neighbours) and a set of variables to be shared within the hood. Hoods are mainly defined geographically with nodes being included in hoods based on the number of hops from a node. For example, a hood could be a one-hop or two-hop neighbourhood, over which temperature readings are shared or a one-hop neighbourhood over light and temperature readings. A node can therefore define multiple neighbourhoods over different variables.

Whitehouse et al. use a broadcast/filter mechanism to allow data sharing and neighbourhood discovery. Shared attributes are broadcast and receiving nodes cache any attributes of interest from valuable neighbours. A neighbour's value is based on how the neighbourhood has been defined. For example, a node may define a routing neighbourhood that caches location information of valuable routing nodes [33]. Once a neighbourhood has been defined and attributes broadcast, interested observers add the broadcasting node to its neighbour list and cache the attribute or attributes of interest. Fig. 8 shows a model for an application with two neighbourhoods and two shared attributes.

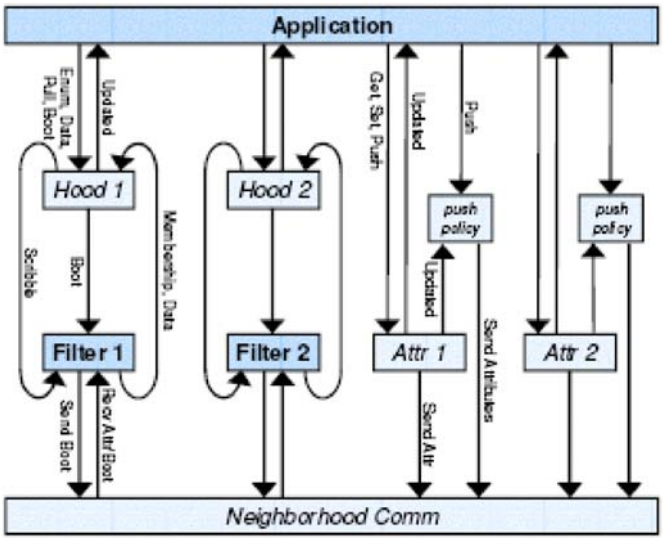

Fig. 8. Component model for an application with two neighbourhoods and two shared attributes taken from [33].

Abstract Regions and Region Streams: Welsh and Mainland [34] propose *abstract regions*, spatial operators used to hold local communications in regions which can be defined in terms of radio connectivity or geographic location. A region in that sense would serve as an identifier for neighbouring nodes who can then share data, identify each other, and reduce data among them. Nodes could then be manipulated via their common interface. An abstract region could therefore be defined as a relationship between a node and its neighbours and the main aim of this work is to move away from the need to construct low-level mechanisms for routing and data collection and instead focus on a higher level interface that is flexible enough to allow a user to implement queries. Such a system would not, for instance, have to consider routing, data dissemination or state management in order to function effectively, however it would still allow a programmer to create applications that can adjust to changing network conditions as well as adjust energy consumption to improve accuracy and latency.

3.2. Semantic Streams

Semantic streams, a system proposed by [35] allows a user to issue queries over *semantic* values **directly** without identifying what data should be targeted or what operations should be used on that data. It allows the user to interact with the sensor network using declarative statements, for example, "I want the ratio of cars to trucks in the parking garage". There is no need to construct code to actually check the data for the existence of cars or trucks rather components referred to as inference units are used to facilitate interpretation of sensor data. This is in direct contrast to other approaches that issue queries over raw sensor data like TinyDB and Cougar, and is of interest as it is similar in principle to the work proposed on complex queries in this research.

The Programming Model: The semantic streams framework uses the semantic services programming model which contains two main elements, inference units and event streams. An event stream is simply a flow of asynchronous events, each of which represents some real-world entity (for example, in the car park query given earlier, a car detection has properties such as the location at which it was detected, its speed and direction). An inference unit is a *process* that operates on an event stream. It infers semantic information about its environment and either adds it as a property to an existing event stream or generates a new one. Fig. 9 shows the semantic streams model within its service-based architecture.

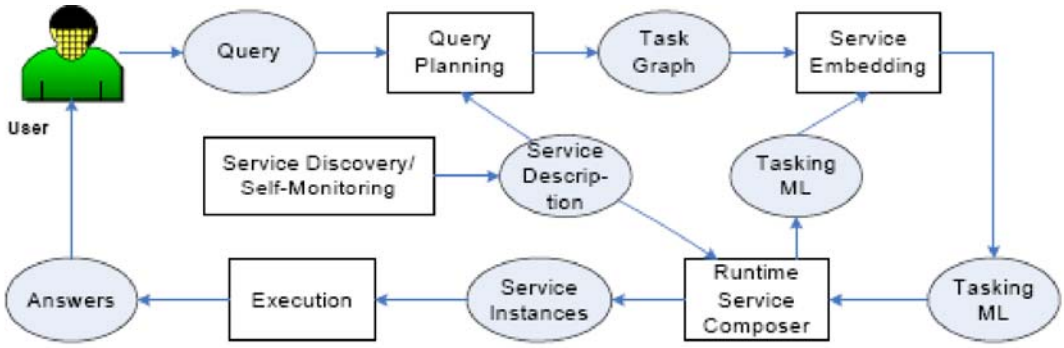

Fig. 9. The semantic streams query processing model taken from [35].

As a stream moves from a sensor through inference units its events acquire new semantic properties. A more detailed example, a vehicle detector service, follows.

The Query Language: Each inference unit is specified using a first-order logic description of the semantic information it needs in its input stream and that it adds to its output streams as well as any relationships between the two. A markup language (the Semantic Streams markup and query language) is used to construct a logical description of that semantic information. A number of predicates are used to describe sensors and services: the sensor predicate defines the type and location of each sensor; the service, needs and creates predicates describe a service, the semantic information that it needs and creates. In query processing these are treated as rules and their pre- and post-conditions; the stream, isa and property predicates describe an event stream and the type and property of its events. An example of a sensor declaration is as follows:

`sensor(` *magnetometer*, [[60,0,0], [70,10,10]])

This defines a magnetometer that covers a three-dimensional cube defined by the pair of 3-D coordinates.

An example of a service would be:

`service(` *magVehicleDetectionService*,
`needs(` `sensor(`*magnetometer*, *R*)),
`creates(` stream(*X*), isa(*X*, *vehicle*),
`property(`*X*, *T*, *time*), property(*X*, *R*, *region*)))

This vehicle detector service uses a magnetometer to detect vehicles and generates an event stream with the time and location in which the vehicles were detected.

Once a set of sensors and services have been declared the user can begin to issue queries. An example of a simple query would be:

`stream`(X), `isa`(X, vehicle)

This query would result in true if a set of services could be composed to generate events X that are known to be vehicles. All known possible service compositions would be generated. To constrain the result more predicates could be added. For instance,

`stream`(*X*), `isa`(*X*, *car*)
`property`(*X*, [[10,0,0], [30,20,20]], *region*)

This would restrict the result to cars found in the region described by the pair of 3-D coordinates given.

Query Processing: An inference engine is used to determine which sensors and services will provide the required semantic information. The inference engine employs the backward-chaining algorithm [36] using the pre-conditions and post-conditions of the services. An attempt is made to match each element of the query to the post-condition of the service; if this is successful the pre-conditions are added to the query. Once all pre-conditions have been matched to actual sensor declarations (without pre-conditions) the process terminates. In other words, the process terminates once all inference units' requirements have been met by physical sensors. A description of a real-world application of the semantic streams framework is described fully in [35].

In the semantic model described, all services and their descriptions are maintained in a central repository or query server. Resource usage is optimized because the inference engine reuses services when possible, reuses resources and operations and the query language used allows great flexibility in

how the query processor executes queries. The language also allows the user to specify constraints. When a user issues a query as an event stream the query planning engine generates a task graph which is then assigned to a set of nodes in the network. The service runtime on each node then accepts the graphs and instantiates services as required, resolves any conflicts between tasks and available resources and then executes the query.

One major limitation of the system is the semantic markup language used. Although suitable for simple examples it cannot capture more complex applications. Another drawback of the system is that the query processor cannot reason at runtime and so two applications which may possibly have to access the same device, for instance, would not be able to run simultaneously. It does not address sensor network issues that are not semantic transformations. For example, routing data between nodes in a sensor network would not change the semantics of the data being routed. Semantic Streams can not differentiate between different routing algorithms and so cannot take advantage of their features if needed.

3.3. The Regiment Macroprogramming System

Regiment, proposed by [37] is a functional programming language for sensor networks. Its data model is based on the concept of region streams, where sensor nodes are represented as streams of data that can be grouped into regions for the purpose of in-network aggregation or event detection. Collections of nodes, therefore, can be represented both spatially and temporally. A region stream is used to describe a collection of nodes with a logical, topological or geographical relationship. For example, a programmer may be interested in nodes which fall within certain 3D coordinates. The corresponding region stream would represent all sensor values for those nodes.

Operations allowed on region streams include *fold* which aggregates values across nodes in the region to an anchor node and *map* which applies a function over all values in a region stream. A *fold* requires inter-node communication while a map does not. Regiment, as is the case with other functional programming languages, allows functions to take functions as arguments.

The Regiment compiler converts the high level program into node-level code that targets an abstract machine model called the token machine. This is an intermediate language that links token handlers to a token name which represents a task to be executed by a sensor node once it receives a token of that name. A token may be generated locally or through a radio message. The token machine language described by [38] elaborates on these concepts and defines the Token Machine Language which is based on an abstract machine model called Distributed Token Machines (DTMs). Typed messages with a small payload (a fixed size buffer) are referred to as token messages and used by DTMs for communication. Each token has an associated token handler which is executed when a token message is received by a node. DTMs, in addition, allow for concurrency, state management and communication. The aims of TML are to provide abstractions for key requirements in the sensor network including data dissemination and communication and provide a framework within which complex algorithms, such as those that allow in-network aggregation, can be implemented.

[8] describe the Regiment Macroprogramming System which extends the work done on the Regiment language and token machine interface. It uses ‘signals’ which represent attribute values for nodes, for example, the temperature read by a sensor, and groups signals into regions which are then used as a programming abstraction upon which different operations can be performed (aggregation for instance). Region membership is not static and may vary over time, due to changes in sensed values, node or communication failure or new nodes being added to the network. Key here is that the system abstracts away details of storage, communication and data acquisition from the user and allows the compiler to map global operations on regions and signals to local network structures [8].

The Regiment system has been evaluated in a number of simulated chemical plume detection macroprograms with a network size of 250 nodes over an area of 5000 x 5000m. The results demonstrated the flexibility of the system in maintaining good communication performance [8].

3.4. Kairos

In contrast to some programming approaches that focus on providing high level abstraction for local node behaviour in a distributed computation, the Kairos macroprogramming system [39] focuses on abstractions for specifying global behaviour. This distributed computation encompasses the entire network but uses a centralized approach. The abstraction considers the sensor network as a collection of nodes that can be tasked at the same time from one program. The programming model is based on shared-memory-based parallel programming models over message passing architectures and three programming abstractions are made available in the system. Fig. 10 shows the architecture of the Kairos macroprogramming system.

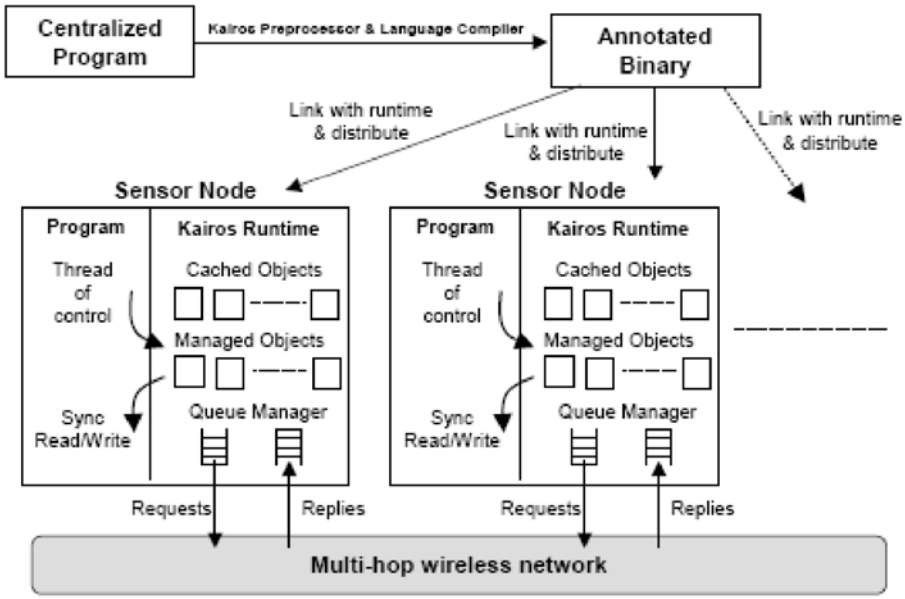

Fig. 10. The Kairos programming architecture taken from [39].

Programming Abstractions: Kairos provides three abstractions for manipulation by the programmer. The first is the *node* abstraction where programmers are allowed to manipulate individual nodes or lists of nodes. Nodes have integer-based identifiers and are represented by a node datatype which makes available operators like equality, ordering and type testing. There are also functions made available for manipulating lists of nodes.

A second abstraction is the set or list of all one-hop neighbours of a specified node which is made available using a get_neighbours() function. This abstraction is quite similar to the region and hood abstractions and it reflects the natural construct of radio neighbourhood of the nodes in the network.

The third abstraction is remote data access which allows a programmer to read from variables at a named node using a variable@node construct. In Kairos only a node may write to its variable although multiple nodes may have read access.

Programming Mechanism: A distinguishing characteristic of the Kairos system is that a programmer writes a single centralized program representing the distributed computation. First, the program is preprocessed to produce annotated source code which is then compiled into a binary. During this stage, the Kairos pre-processor identifies and translates remote data references to calls to the Kairos runtime. The binary can then be distributed to all nodes in the network using a code distribution facility. Although the initial program is a global level one, after compilation a node-level version is created that details what an individual node's functions are, when and what data remote and local it manipulates.

When a copy of the node-level program is instantiated and executed on a node, the runtime exports and manages variables that are owned by a particular node but are referenced by remote nodes.

Kairos has been used to implement programs for a variety of problems from routing tree construction to vehicle tracking [39].

3.5. Knowledge-Representation for Sentient Computing

An interesting approach is proposed by [40] called a scalable knowledge representation and abstract reasoning system for Sentient Computing. Sentient Computing promotes the view that an application can be made more aware by perceiving the environment and reacting to changes in it. A gap exists, however, between the level of abstraction in the knowledge of the sentient world such a system requires in order to function and the low-level data produced by sensors in the environment. [40] propose the creation of a deductive component that would interact with the low level data, perform some type of reasoning function in order to deduce a higher-level abstract knowledge. This knowledge can then be used by the application layer. The deductive knowledge base layer therefore is really a domain specific language component positioned as a type of middleware layer between the application and hardware layers. A diagram of the system architecture is displayed in Fig. 11.

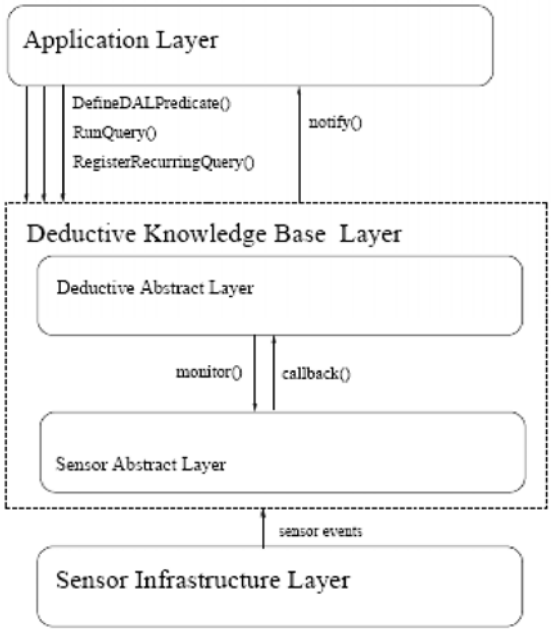

Fig. 11. System architecture, taken from [40].

Knowledge Representation: An *event* in this context is defined as an occurrence of interest taking place instantaneously at a specific time. The *Sentient* environment refers to the physical environment.

The *current logical state* of the Sentient environment includes all known facts about the sentient environment from an *initial event* to a *terminal event*. These events can include any events of interest to the *Sentient Application layer*. The *Sensor Abstract Layer* (SAL) keeps a low-level view on the current logical state as sensors in the environment update it through sensed events. The *Deductive Abstract Layer* (DAL) maintains a high-level or abstract knowledge of the current logical state through its interaction with the SAL.

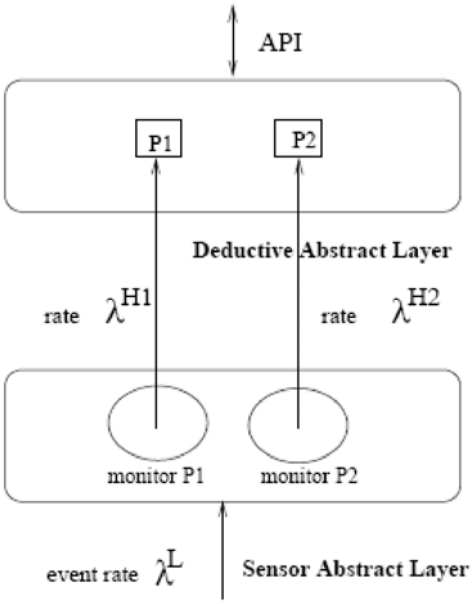

Fig. 12. Interaction between DAL and SAL, taken from [40].

The two layers use a *monitor-callback* communication scheme to interact. The application layer issues a monitor call which causes the SAL to filter through to the DAL those low-level changes that affect the abstract knowledge in the DAL. The DAL, therefore, does not have to monitor all the data and is updated at a much lower rate than the SAL. This is depicted in Fig. 12.

Both DAL and SAL were described using first order logic and queries which are quite similar to SQL SELECT statements are used by the application layer to retrieve the stored knowledge about the *Sentient* environment. Experiments conducted with a prototype implementation showed that the two-layered architecture where the low-level and high-level interactions are separated (SAL and DAL) was more efficient than a single-layered one for a number of reasons [40]. First, queries executed in the DAL were of a simpler form with fewer conditions so used less computational resources. Second, the knowledge update rate triggered by *assert* or *retract* commands in the DAL is lower than that of the SAL making DAL more computationally efficient. Experiments with more complex queries have not been carried out.

4. Concluding Remarks

The three approaches to information extraction surveyed over parts I and II show systems and techniques with varying degrees of ease of use, flexibility, informational complexity and expressiveness. While the querying approaches are easier for a user to manipulate they are restricted by the lack of expressiveness of the query languages they provide for requesting information. This in turn restricts the type and level of information that can be retrieved. Although it is clear that simple queries for attributes and simple aggregates are possible and extensively demonstrated in the

deployments to date, informational query-based systems that go beyond that (for example spatially and temporally aware queries) are practically non-existent. Query-based systems are also limited as they do not allow users the flexibility to target smaller areas of interest within the network but in effect force the user to task the entire network in order to get a response, an issue in terms of energy efficiency. An advantage however is that these systems have been extensively used, the languages used familiar to users of sensor network systems and are relatively easier to design and implement when compared to other information extraction mechanisms.

In contrast, the agent-based approaches are more expressive and flexible in terms of what functions can be performed and the ability they incorporate to make decisions while in the network. Agents can act autonomously, multiple agents can run on a node at the same time, and multiple applications can co-exist in the network (this can be a problem with many query-based and macroprogramming systems). Mobile agents can move, clone themselves and can act to deal with unexpected changes in the environment. Agent-based systems, however, are hindered by the difficulty they present for non-expert users to design and program. It is also apparent that despite the many agent-based techniques and models proposed in the literature, actual deployments have been minimal and restricted to very small networks (less than 10 nodes) with the agents being, at most, just pieces of mobile code with minimal autonomy or decision-making capability.

The Macroprogramming approach appears to be one solution to the problem of the limitations of query-based systems while at the same time promoting the expressiveness inherent in agent-based systems. Some researchers consider macroprogramming an extension to query-based applicative approaches and attempt to adhere to the ease of usability mantra of query-based systems, while at the same time providing users the ability to access higher level information. The reality, however, is that the systems are not nearly as intuitive as many of the SQL-based querying systems and require a learning curve for the programmer. In addition, the approach seeks to eliminate the need for low-level programming on the node but in reality has to provide node-specific abstractions which undermine rapid program development. Finally, the high abstraction level although advantageous on the one hand in terms of shielding the user from the underlying workings of the network (radio communication and network topology, for example) presents a challenge when constructing compilers that need to cater not just for computation but node level communication as well. Macroprogramming therefore presents similar challenges to the agent-based approaches.

Overall, the approaches to information extraction examined highlight that trade-offs must be made between ease of use and expressiveness coupled with flexibility (for instance in terms of decision-making capabilities within the network) in selecting a suitable information extraction system. Clearly, the query-based and macroprogramming approaches are useful in a number of applications but raise a number of challenges as outlined above. There is a definite gap between the ease of use provided by the more popular query-based mechanisms and the expressiveness of the global and node level abstractions provided by macroprograms as well as the flexibility of agent-based systems. For purposes of any applied research in WSNs, the idea of usability and utility are paramount. Future research directions and open issues are presented in Part III of this suite of papers. To that end, the authors here put the development of a declarative query-based system forward as being a viable option. Given the limitations already described above, a number of features have to be incorporated into such a query system. In order to make the querying approach more attractive, the development of a query language that allows a user to more richly express high level information requests which can target the full breadth of collectable information presented by sensors in the network is promoted. Complex queries could be suitable constructs that can allow the expression of spatio-temporal characteristics and requests requiring more involved in-network interactions to generate information. Incorporating within this query processing system useful elements that will aid the processing of these complex queries, such as abstractions already employed in some macroprogramming approaches to facilitate resolution of this higher level information requests would be necessary. Such a *hybrid* approach, it is

anticipated, will retain the simplicity and ease of use of the traditional query-based approaches while simultaneously allowing the inclusion of useful logical abstractions provided by macroprogramming approaches to facilitate dissemination, processing and resolution of more powerful queries than those available in query processing systems today.

References

[1]. W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, Energy-efficient communication protocol for wireless microsensor networks, In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, Maui, HI, USA, Vol. 2, 2000, pp. 10-16.

[2]. S. Madden, R. Szewczyk, M. J. Franklin, D. Culler, Supporting aggregate queries over ad-hoc wireless sensor networks, In *Proceedings of 4th IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, USA, 2002, pp. 49–58.

[3]. G. J. Pottie, W. J. Kaiser, Wireless integrated network sensors, *Communications of the ACM*, 43, 5, 2000, pp. 51–8.

[4]. P. Bonnet, J. Gehrke, P. Seshadri, Querying the physical world, *IEEE Personal Communications*, 7, 5, 2000, pp. 10–15.

[5]. Y. Yao, J. Gehrke, Query processing for sensor networks, In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, 2003.

[6]. J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, D. Ganesan, Building efficient wireless sensor networks with low-level naming, *Operating Systems Review (ACM)*, Banff, Alta, Canada, 35, 2002, pp. 146–159.

[7]. Ryan Newton, Compiling Functional Reactive Macroprograms for Sensor Networks, *Masters Thesis*, Massachusetts Institute of Technology, 2005.

[8]. R. Newton, G. Morrisett, M. Welsh, The regiment macroprogramming system, In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, 2007, Cambridge, Massachusetts, USA, pp. 489-498.

[9]. A. Silberstein, Push and pull in sensor network query processing, In *Proceedings of Southeast Workshop on Data and Information Management (SWDIM '06)*, Raleigh, North Carolina, 2006, http://www.cs.duke.edu/~adam/

[10]. P. Bonnet, J. Gehrke, and P. Seshadri, Towards sensor database systems, Mobile Data Management. Second International Conference, MDM 2001, Proceedings (Lecture Notes in Computer Science Vol., 1987), Hong Kong, China, *Springer-Verlag*, 2001, pp. 3–14.

[11]. R. Schollmeier, A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, In *Proceedings of the 1st International Conference on Peer-to-Peer Computing*, Linkoping, Sweden, 2002, pp. 101–102.

[12]. P. Bonnet, P. Seshadri, Device database systems, In *Proceedings of the International Conference on Data Engineering*, San Diego, California, 2000, p. 194.

[13]. R. Govindan, J. M. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker, The sensor network as a database, Technical Report 0-771, Computer Science Dept., University of Southern California., 2000.

[14]. Y. Yao, J. Gehrke, The cougar approach to in-network query processing in sensor networks, *SIGMOD Rec.*, 31, 3, 2002, pp. 9–18.

[15]. P. Seshadri, Predator: A resource for database research, *SIGMOD Rec*, 27, 1, 1998, pp. 16–20.

[16]. J. Gehrke and S. Madden, Query processing in sensor networks, *IEEE Pervasive Computing*, 3, 1, January - March 2004, pp. 46–55.

[17]. A. Demers, J. Gehrke, R. Rajmohan, N. Trigoni, Y. Yong, The cougar project: a work-in-progress report, *SIGMOD Record*, 32, 4, 2003, pp. 53–59.

[18]. S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan, Irisnet: An architecture for enabling sensor-enriched internet service, *Technical IRP-TR-03-04*, Intel Research, June 2003.

[19]. Shen Chien-Chung, C. Srisathapornphat, and C. Jaikaeo, Sensor information networking architecture and applications, *IEEE Personal Communications*, 8, 4, 2001, pp. 52–59.

[20]. L. B. Ruiz, J. M. Nogueira, and A. A. F. Loureiro, Manna: a management architecture for wireless sensor networks, *IEEE Communications Magazine*, 41, 2, 2003, pp. 116–125.

[21]. L. Shuoqi, S. H. Son, J. A. Stankovic, Event detection services using data service middleware in distributed sensor networks, In *Proceedings of Information Processing in Sensor Networks. 2nd International*