Technical report LSIR-REPORT-2006-006

The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks[⋆]

Karl Aberer, Manfred Hauswirth, Ali Salehi

School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
`firstname.lastname@epfl.ch`

Abstract. The lack of standardization and the continuous inflow of novel sensor network technologies have made their deployment the main factor of manpower consumption, considerably complicate the interconnection of heterogeneous sensor networks, and make portable application development a challenging and time-consuming task. To address these problems we propose our Global Sensor Networks middleware which supports the rapid and simple deployment of a wide range of sensor network technologies, facilitates the flexible integration and discovery of sensor networks and sensor data, enables fast deployment and addition of new platforms, provides distributed querying, filtering, and combination of sensor data, and supports the dynamic adaption of the system configuration during operation. GSN offers virtual sensors as a simple and powerful abstraction which enables the user to declaratively specify XML-based deployment descriptors in combination with the possibility to integrate sensor network data through plain SQL queries over local and remote sensor data sources. The paper describes GSN's conceptual model and system architecture, and demonstrates the efficiency of the implementation through experiments with typical high-load application profiles. The GSN implementation is available from http://globalsn.sourceforge.net/.

Keywords: Sensor networks, sensor middleware, sensor internetworking

1 Introduction

Until now, research in the sensor network domain has mainly focused on routing, data aggregation, and energy conservation inside a single sensor network. The deployment, application development, and standardization aspects have only been addressed to a limited extent so far. However, as the price of wireless sensors diminishes rapidly we can expect to see large numbers of autonomous sensor networks. Major challenges in such a "Sensor Internet" environment are the sharing and integration of data among heterogeneous sensor networks and minimizing of deployment efforts which are a key

[⋆] The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322 and was (partly) carried out in the framework of the EPFL Center for Global Computing.

cost factor in any system. As successfully demonstrated in other domains, a standard strategy to address these problems is to devise a middleware which provides powerful abstractions, codifying the essential requirements and concepts of a domain and offering flexible means for integrating the concrete physical platforms. This speeds up deployment and additionally pushes forward standardized APIs which simplifies application development and enables application portability across all systems supported by the middleware.

Our Global Sensor Network (GSN) middleware follows this rationale and provides a uniform platform for fast and flexible deployment and integration of heterogeneous sensor networks. The design of GSN follows four main design goals:

Simplicity. GSN is based on a minimal set of powerful abstractions which can be configured and adopted easily to the user's needs. Sensor networks and data streams can be specified in a declarative way using XML as the syntactic framework and SQL as the data manipulation language.

Adaptivity. GSN allows the user to add new types of sensor networks and facilitates dynamic (re-) configuration of the system during run-time without having to interrupt ongoing system operation through a container-based implementation.

Scalability. To support very large numbers of data producers and consumers with a variety of application requirements, GSN considers scalability issues specifically for distributed query processing and distributed discovery of sensor networks. To meet this requirement, the design of GSN is based on a peer-to-peer architecture.

Light-weight implementation. GSN is easily deployable in standard computing environments (no excessive hardware requirements, standard network connectivity, etc.), portable (Java-based implementation), requires minimal initial configuration, and provides easy-to-use, web-based management tools.

In the following sections of the paper we describe GSN's key abstractions and design decisions, its architecture and implementation, and evaluate the system using real-world scenarios to demonstrate its efficiency and applicability.

2 Virtual sensors

The key abstraction in GSN is the *virtual sensor*. Virtual sensors abstract from implementation details of access to sensor data and correspond either to a data stream received directly from sensors or to a data stream derived from other virtual sensors. A virtual sensor can be any kind of data producer, for example, a real sensor, a wireless camera, a desktop computer, a cell phone, or any combination of virtual sensors. A virtual sensor may have any number of input data streams and produces exactly one output data stream based on the input data streams and arbitrary local processing. The specification of a virtual sensor provides all necessary information required for deploying and using it, including:

- metadata used for identification and discovery;
- the structure and properties of the data streams which the virtual sensor consumes and produces;
- a declarative SQL-based specification of the data stream processing performed in the virtual sensor;
- functional properties related to stream quality management, persistency, error handling, life-cycle management, and physical deployment.

To support rapid deployment, these properties of virtual sensors are provided in a declarative deployment descriptor. Figure 1 shows an example which defines a virtual sensor that reads two temperature sensors and in case both of them have the same reading above a certain threshold in the last minute, the virtual sensor returns the latest picture from the webcam in the same room together with the measured temperature.

```
 1   <virtual-sensor name="room-monitor" priority="11">
 2      <addressing>
 3         <predicate key="geographical">BC143</predicate>
 4         <predicate key="usage">room monitoring</predicate>
 5      </addressing>
 6      <life-cycle pool-size="10" />
 7      <output-structure>
 8         <field name="image" type="binary:jpeg" />
 9         <field name="temp"  type="int" />
10      </output-structure>
11      <storage permanent="true" history-size="10h" />
12      <input-streams>
13         <input-stream name="cam">
14            <stream-source alias="cam"  storage-size="1"
15                           disconnect-buffer-size="10">
16               <address wrapper="remote">
17                  <predicate key="geographical">BC143</predicate>
18                  <predicate key="type">Camera</predicate>
19               </address>
20               <query>select * from WRAPPER</query>
21            </stream-source>
22            <stream-source alias="temperature1" storage-size="1m"
23                           disconnect-buffer-size="10">
24               <address wrapper="remote">
25                  <predicate key="type">temperature</predicate>
26                  <predicate key="geographical">BC143-N</predicate>
27               </address>
28               <query>select AVG(temp1) as T1 from WRAPPER</query>
29            </stream-source>
30            <stream-source alias="temperature2"  storage-size="1m"
31                           disconnect-buffer-size="10">
32               <address wrapper="remote">
33                  <predicate key="type">temperature</predicate>
34                  <predicate key="geographical">BC143-S</predicate>
35               </address>
36               <query>select AVG(temp2) as T2 from WRAPPER</query>
37            </stream-source>
38            <query>
39               select cam.picture as image, temperature.T1 as temp
40               from   cam, temperature1
41               where  temperature1.T1 > 30 AND
42                      temperature1.T1 = temperature2.T2
43            </query>
44         </input-stream>
45      </input-streams>
46   </virtual-sensor>
```

Fig. 1. A complex virtual sensor definition using other virtual sensors as input

A virtual sensor has a unique name (the `name` attribute in line 1) and can be equipped with a set of key-value pairs (lines 2–5), i.e., associated with metadata. Both types of addressing information can be registered and discovered in GSN and other virtual sensors can use either the unique name or logical addressing based on the metadata to refer to a virtual sensor. The example specification above defines a virtual sensor with three input streams which are identified by their metadata (lines 17–18, 25–26, and 33–34), i.e., by logical addressing. For example, the first temperature sensor is addressed by specifying two requirements on its metadata (lines 25–26), namely that it is of type temperature sensor and at a certain physical certain location. By using multiple input streams Figure 1 also demonstrates GSN's ability to access multiple stream producers simultaneously. For the moment, we assume that the input streams (two temperature sensors and a webcam) have already been defined in other virtual sensor definitions (how this is done, will be described below).

In GSN data streams are temporal sequences of timestamped tuples. This is in line with the model used in most stream processing systems. The structure of the data stream a virtual sensor produces is encoded in XML as shown in lines 7–10. The structure of the input streams is learned from the respective specifications of their virtual sensor definitions. Data stream processing is separated into two stages: (1) processing applied to the input streams (lines 20, 28, and 36) and (2) processing for combining data from the different input streams and producing the output stream (lines 38–43). To specify the processing of the input streams we use SQL queries which refer to the input streams by the reserved keyword `WRAPPER`. The attribute `wrapper="remote"` indicates that the data stream is obtained through the network from another GSN instance.

In the given example the output stream joins the data received from two temperature sensors and returns a camera image if certain conditions on the temperature are satisfied (lines 38–43). To enable the SQL statement in lines 39–42 to produce the output stream, it needs to be able to reference the required input data streams which is accomplished by the `alias` attribute (lines 14, 22, and 30) that defines a symbolic name for each input stream. The definition of the structure of the output stream directly relates to the data stream processing that is performed by the virtual sensor and needs to be consistent with it, i.e., the data fields in the `select` clause (line 40) must match the definition of the output stream in lines 7–10.

In the design of GSN specifications we decided to separate the temporal aspects from the relational data processing using SQL. The temporal processing is controlled by various attributes provided in the input and output stream specifications, e.g., the attribute `storage-size` (lines 14, 22, and 30) defines the time window used for producing the input stream's data elements. Due to its specific importance the temporal processing will be discussed in detail in Section 3.

In addition to the specification of the data-related properties a virtual sensor also includes high-level specifications of functional properties: The `priority` attribute (line 1) controls the processing priority of a virtual sensor, the `<life-cycle>` element (line 6) enables the control and management of resources provided to a virtual sensor such as the maximum number of threads/queues available for processing, the `<storage>` element (line 11) allows the user to control how output stream data is persistently stored, and the `disconnect-buffer-size` attribute (lines 15, 23, 31) specifies the amount of storage provided to deal with temporary disconnections.

For example, in Figure 1 the `priority` attribute in line 1 assigns a priority of 11 to this virtual sensor (10 is the lowest priority and 20 the highest, default is 10), the `<life-cycle>` element in line 6 specifies a maximum number of 10 threads, which means that if the pool size is reached, data will be dropped (if no pool size is specified, it will be controlled by GSN depending on the current load), the `<storage>` element in line 11 defines that the output stream's data elements of the last 10 hours (`history-size` attribute) are stored permanently to enable off-line processing, the `storage-size` attribute in line 14 defines that the last image taken by the webcam will be returned irrespective of the time it was taken, whereas the `storage-size` attributes in lines 22 and 30 define a time window of one minute for the amount of sensor readings subsequent queries will be run on, i.e., the `AVG` operations in lines 28 and 36 are executed on the sensor readings received in the last minute which of course depends on the rate at which the underlying temperature virtual sensor produces its readings, and finally, the `disconnect-buffer-size` attributes in lines 15, 23, and 31 specify up to 10 missed sensor readings to be read after a disconnection from the associated stream source.

The query producing the output stream (lines 39–42) also demonstrates another interesting capability of GSN as it also mediates among three different flavors of queries: The virtual sensor itself uses continuous queries on the temperature data, a "normal" database query on the camera data and produces a result only if certain conditions are satisfied, i.e., a notification analogous to pub/sub or active rules.

In contrast to Figure 1, which shows the specification of a virtual sensor for processing data streams received from other virtual sensors, Figure 2 shows a virtual sensor producing a data stream generated directly by a sensor, in this case a TinyOS-based wireless sensor.

```
1   <virtual-sensor name="Light-sensor1" priority="11">
2      <class>gsn.vsensor.BridgeVirtualSensor</class>
3      <author>Ali Salehi</author>
4      <email>ali.salehi@epfl.ch</email>
5      <description>A TinyOS temperature vsensor</description>
6      <life-cycle pool-size="10" />
7      <addressing>
8         <predicate key="geographical">BC143-N</predicate>
9         <predicate key="type">temperature</predicate>
10     </addressing>
11     <output-structure>
12        <field name="temperature" type="int" />
13     </output-structure>
14     <storage permanent="true" history-size="10s" />
15     <input-streams>
16        <input-stream name="temperature" >
17           <stream-source alias="tsensor" storage-size="1">
18              <address wrapper="tinyos">
19                 <predicate key="host">lsirpc24.epfl.ch</predicate>
20                 <predicate key="port">9001</predicate>
21              </address>
22              <query>
23                 select WRAPPER.TEMPERATURE as temperature,
24                        WRAPPER.TIMED as timestamp from WRAPPER
25              </query>
26           </stream-source>
27           <query>
28              select temperature from tsensor
29           </query>
30        </input-stream>
31     </input-streams>
32  </virtual-sensor>
```

Fig. 2. Virtual sensor definition for a temperature sensor using a TinyOS mote

This virtual sensor obtains its data stream from a specific sensor network which is directly attached to the computer hosting the GSN instance, rather than through a remote GSN instance. Thus the sensor is addressed physically (lines 18–21) rather than logically. Despite this necessary difference in addressing, locally and remotely produced data streams are treated logically the same way and the hosting GSN instance does not know whether the data is coming from a local or a remote sensor. This specification also relies on the availability of a wrapper implementation that connects the system to the specific type of sensor or sensor network. The implementation of the wrapper is referenced explicitly in line 18 by the attribute `wrapper="tinyos"`. GSN already includes wrappers for many platforms. The effort to implement new wrappers is quite low and we will discuss this in detail in Section 5.1.

The virtual sensor mediates between GSN and the physical sensor. The interface and access to the sensor is provided by the wrapper as described above. Additionally, GSN needs to be able to interact with the virtual sensor and this is supported by another API whose implementation is referenced in line 2. `BridgeVirtualSensor` is such an implementation provided by GSN by default which just takes the results of the input streams (lines 16–32) and provides them in the format specified in lines 11–13. This is just the simplest case as in fact, this API in meant to support arbitrary application-specific processing, if required. For example, special detection or image processing algorithms for image data coming from a camera may be provided here by providing a customized implementation of the `AbstractVirtualSensor` class defining the API (`BridgeVirtualSensor` also implements `AbstractVirtualSensor`).

Due to space limitations we cannot describe all possible configuration options, for example, the various notification possibilities, such as email or SMS. A complete list along with descriptions and examples is available from the GSN website at http://globalsn.sourceforge.net/.

Virtual sensors are a powerful abstraction mechanism which enables the user to declaratively specify sensors and combinations of arbitrary complexity. It hides the physical details of the actual sensing devices as much as possible, but still facilitates the explicit control of any processing aspect, if required. Virtual sensors can be defined and deployed to a running GSN instance at any time without having to stop the system. Also dynamic unloading is supported but should be used carefully as unloading a virtual sensor may have undesired (cascading) effects.

3 Data stream processing and time model

Data stream processing has received substantial attention in the recent years in other application domains, such as network monitoring or telecommunications. As a result, a rich set of query languages and query processing approaches for data streams exist on which we can build. A central building block in data stream processing is the time model as it defines the temporal semantics of data and thus determines the design and implementation of a system. Currently, most stream processing systems use a global reference time as the basis for their temporal semantics because they were designed for centralized architectures in the first place. As GSN is targeted at enabling a distributed "Sensor Internet," imposing a specific temporal semantics seems inadequate and maintaining it might come at unacceptable cost. GSN provides the essential building blocks for dealing with time, but leaves temporal semantics largely to applications allowing them to express and satisfy their specific, largely varying requirements. In our opinion, this pragmatic approach is viable as it reflects the requirements and capabilities of sensor network processing.

In GSN a data stream is a set of timestamped tuples. The order of the data stream is derived from the ordering of the timestamps and GSN provides basic support for managing and manipulating the timestamps. The following essential services are provided:

1. a local clock at each GSN container;

2. implicit management of a timestamp attribute (TIMEID);
3. implicit timestamping of tuples upon arrival at the GSN container (reception time);
4. a windowing mechanism which allows the user to define count- or time-based windows on data streams.

In this way it is always possible to trace the temporal history of data stream elements throughout the processing history. Multiple time attributes can be associated with data streams and can be manipulated through SQL queries. Thus sensor networks can be used as observation tools for the physical world, in which network and processing delays are inherent properties of the observation process which cannot be made transparent by abstraction. Let us illustrate this by a simple example: Assume a bank is being robbed and images of the crime scene taken by the security cameras are transmitted to the police. For the insurance company the time at which the images are taken in the bank will be relevant when processing a claim, whereas for the police report the time the images arrived at the police station will be relevant to justify the time of intervention. Depending on the context the robbery is thus taking place at different times.

The temporal processing in GSN is defined as follows: The production of a new output stream element of a virtual sensor is always triggered by the arrival of a data stream element from one of its input streams. Thus processing is event-driven and the following processing steps are performed:

1. By default the new data stream element is timestamped using the local clock of the virtual sensor provided that the stream element had no timestamp.
2. Based on the timestamps for each input stream the stream elements are selected according to the definition of the time window and the resulting sets of relations are unnested into flat relations.
3. The input stream queries are evaluated and stored into temporary relations.
4. The output query for producing the output stream element is executed based on the temporary relations.
5. The result is permanently stored if required (possibly after some processing) and all consumers of the virtual sensor are notified of the new stream element.

This logical data flow inside a GSN node is shown in Figure 3.

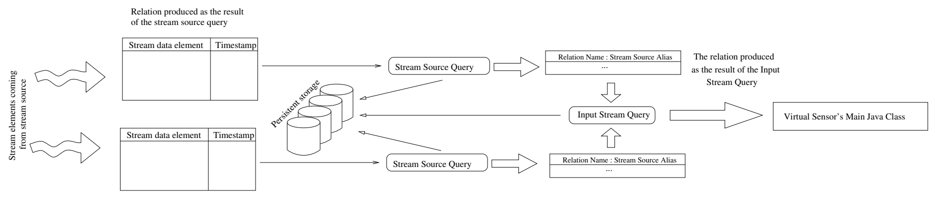

Fig. 3. Conceptual data flow in a GSN node

Additionally, GSN provides a number of possibilities to control the temporal processing of data streams, for example:

– The rate of a data stream can be bounded in order to avoid overloading the system which might cause undesirable delays.

- Data streams can be sampled to reduce the data rate.
- A windowing mechanism can be used to limit the amount of data that needs to be stored for query processing. Windows can be defined using absolute, landmark, or sliding intervals.
- The lifetime of data streams and queries can be bounded such that they only consume resources when actually active. Lifetimes can be specified in terms of explicit start and end times, start time and duration, or number of tuples.

As tuples (sensor readings) are timestamped, queries can also deal explicitly with time. For example, the query in lines 39–42 of Figure 1 could be extended such that it explicitly specifies the maximum time interval between the readings of the two temperatures and the maximum age of the readings. This would additionally require changes in the input stream definitions as the input streams then must provide this information, and also the averaging of the temperature readings (lines 28 and 36) would have to be changed to be explicit in respect to the time dimension. GSN supports all this but due to space limitations we cannot go into further detail (see the user and developer guides at http://globalsn.sourceforge.net/ for a full specification).

Additionally, GSN can easily support the integration of continuous and historical data. For example, if the user wants to be notified when the temperature is 10 degrees above the average temperature in the last 24 hours, he/she can simply define two stream sources, getting data from the same wrapper but with different window sizes, i.e., 1 (count) and 24h (time), and then simply write a query specifying the original condition with these input streams.

Although GSN's time model is simple it is very flexible and supports a wide range of application scenarios, for example:

- In a fire alarm scenario when a smoke sensor fires, the user may want to be notified every 5 minutes and not every 10ms when the sensor does the observation. This can be accomplished by bounding the rate.
- In an e-home scenario, the user may want to be notified once when the sound sensor reacts to a crying baby. This can accomplished simply by limiting the number of readings to 1.
- In a monitoring scenario, a security guard may only want to be notified of alarms when he is on duty which can be specified via the query's lifetime.
- The user may be interested in the average of a randomly chosen subset of sensor readings, which could be specified via a sampling rate on the raw sensor readings.

If the user does not explicitly specify the temporal processing model, GSN uses default policies which try to best use the available system resources while trying to achieve load balancing among the sensors and the associated data processing for globally optimal performance.

To specify the data stream processing a suitable language is needed. A number of proposals exist already, so we compare the language approach of GSN to the major proposals from the literature. In the Aurora project [1] (http://www.cs.brown.edu/research/aurora/) users can compose stream relationships and construct queries in a graphical representation which is then used as input for the query planner. The Continuous Query Language (CQL) suggested by the STREAM project [2] (http://www-db.stanford.edu/stream/) extends standard SQL syntax with new constructs for temporal semantics and defines a mapping between streams and relations. Similarly, in Cougar [3] (http://www.cs.cornell.edu/database/cougar/) an extended version of SQL is used, modeling temporal characteristics in the language itself. The StreaQuel language suggested by the TelegraphCQ project [4] (http://telegraph.cs.berkeley.edu/) follows a different path and tries to isolate temporal semantics from the query language through external definitions in a C-like syntax. For example, for specifying a sliding window for a query a *for*-loop is used. The actual query is then formulated in an SQL-like syntax.

GSN's approach is related to TelegraphCQ's as it separates the time-related constructs from the actual query. Temporal specifications, e.g., the window size and rates, are specified in XML in the virtual sensor specification, while data processing is specified in SQL. At the moment GSN supports SQL queries with the full range of operations allowed by the standard SQL syntax, i.e., joins, subqueries, ordering, grouping, unions, intersections, etc. The advantage of using SQL is that it is well-known and SQL query optimization and planning techniques can be directly applied.

4 System architecture

GSN uses a container-based architecture for hosting virtual sensors. Similar to application servers, GSN provides an environment in which sensor networks can easily and flexibly be specified and deployed by hiding most of the system complexity in the GSN container. Using the declarative specifications, virtual sensors can be deployed and reconfigured in GSN containers at runtime. Communication and processing among different GSN containers is performed in a peer-to-peer style through standard Internet and Web protocols. By viewing GSN containers as cooperating peers in a decentralized system, we tried avoid some of the intrinsic scalability problems of many other systems which rely on a centralized or hierarchical architecture. Targeting a "Sensor Internet" as the long-term goal we also need to take into account that such a system will consist of "Autonomous Sensor Systems" with a large degree of freedom and only limited possibilities of control, similarly as in the Internet.

Figure 4 shows the layered architecture of a single GSN container.

Each GSN container hosts a number of virtual sensors it is responsible for. The virtual sensor manager (VSM) is responsible for providing access to the virtual sensors, managing the delivery of sensor data, and providing the necessary administrative infrastructure. The VSM has two subcomponents: The life-cycle manager (LCM) provides and manages the resources provided to a virtual sensor and manages the interactions with a virtual sensor (sensor readings, etc.). The input stream manager (ISM) is responsible for managing the streams, allocating resources to them, and enabling resource sharing among them while its stream quality manager subcomponent (SQM) handles sensor disconnections, missing values, unexpected delays, etc., thus ensuring the QoS of streams. All data from/to the VSM passes through the storage layer which is in charge of providing and managing persistent storage for data streams. Query processing in turn relies on all of the above layers and is done by the query manager (QM) which includes the query processor being in charge of SQL parsing, query planning, and execution of queries (using an adaptive query execution plan). The query repository manages all registered queries (subscriptions) and defines and maintains the set of currently active queries for the query processor. The notification manager deals with the delivery of events and query results to registered, local or remote consumers. The notification manager has an extensible architecture which allows the user to largely customize its functionality, for example, having results mailed or being notified via SMS.

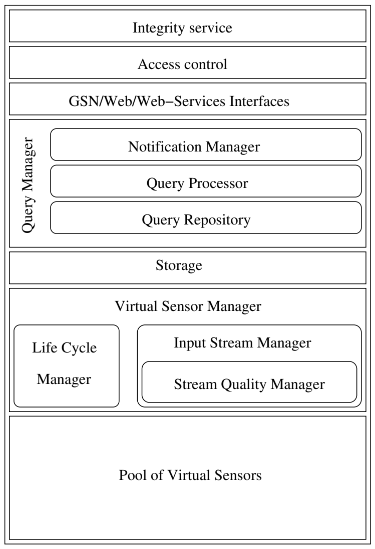

Fig. 4. GSN container architecture

The top three layers of the architecture deal with access to the GSN container. The interface layer provides access functions for other GSN containers and via the Web (through a browser or via web services). These functionalities are protected and shielded by the access control layer providing access only to entitled parties and the data integrity layer which provides data integrity and confidentiality through electronic signatures and encryption. Data access and data integrity can be defined at different levels, for example, for the whole GSN container or at a virtual sensor level.

To demonstrate the logical steps in data stream processing inside this architecture, we include UML sequence diagrams for two major control flows: Figure 5 shows how data produced by a sensor is processed inside a GSN node and Figure 6 shows the control flow if one GSN node uses data coming over the network from another GSN node's virtual sensor.

In Figure 5 the wrapper receives a data item from the producer (physical sensor or another local/remote virtual sensor) and notifies the wrapper repository about the newly received data. The wrapper repository checks for the stream sources registered to receive data from this wrapper and notifies them (a wrapper can be used by one or more stream sources). The stream source manager evaluates the combined query encompassing all the stream source queries which use the wrapper and provides the input stream manager with a list of the input streams which might be affected by the newly arrived data. The input stream manager then evaluates the combined query encompassing all the input stream sources using the affected stream source and in case one or more of the input streams produce a new results, GSN will notify the virtual sensors using them.

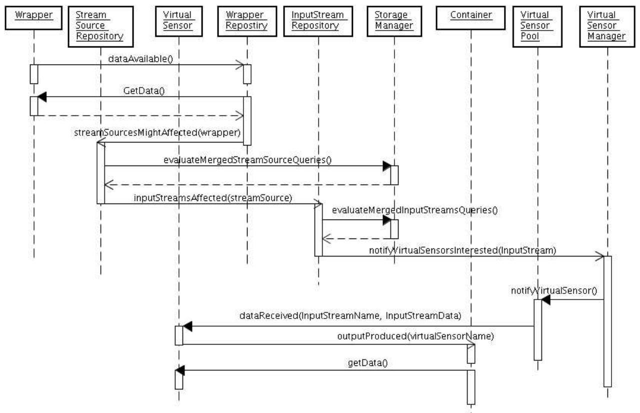
Wrapper
Stream Source Repository
Virtual Sensor
Wrapper Repostiry
InputStream Repository
Storage Manager
Container
Virtual Sensor Pool
Virtual Sensor Manager
dataAvailable()
GetData()
streamSourcesMightAffected(wrapper)
evaluateMergedStreamSourceQueries()
inputStreamsAffected(streamSource)
evaluateMergedInputStreamsQueries()
notifyVirtualSensorsInterested(InputStream)
notifyVirtualSensor()
dataReceived(InputStreamName, InputStreamData)
outputProduced(virtualSensorName)
getData()

Fig. 5. Control flow for a wrapper producing a data stream

In turn, if a virtual sensor produces a new output, the GSN container retrieves it from the virtual sensor, retrieves the registered clients of the virtual sensor, and notifies them if necessary via the notification manager (this is not shown in Figure 5).

In Figure 6 new sensor data from a sensor becomes available at GSN node *A*.

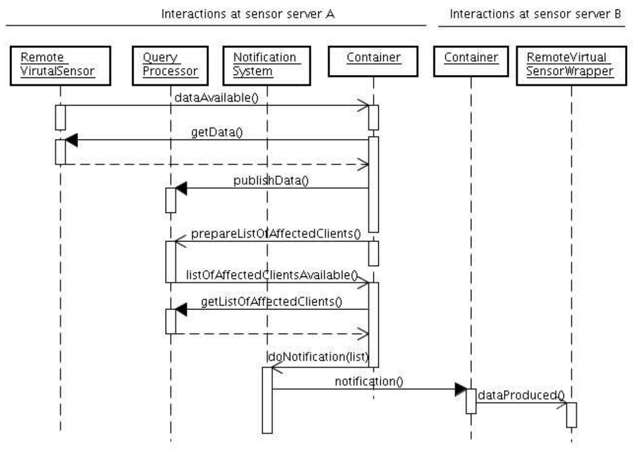
Interactions at sensor server A
Interactions at sensor server B
Remote VirutalSensor
Query Processor
Notification System
Container
Container
RemoteVirtual SensorWrapper
dataAvailable()
getData()
publishData()
prepareListOfAffectedClients()
listOfAffectedClientsAvailable()
getListOfAffectedClients()
doNotification(list)
notification()
dataProduced()

Fig. 6. Using a remote virtual sensor

The GSN container retrieves the new data and hands it to the local query processor who runs all queries affected by the new data. As a result of this processing some of the registered clients should receive new data. Thus the GSN container requests a list of clients for which new data has become available and instructs the notification manager to do the actual notification. In the figure we assume that new data for a query registered from node *B* has become available and thus node *B*'s virtual sensor is notified which acts as a virtual data source and in turn notifies the specific wrapper being responsible for the remote virtual sensor. Then basically the same interactions as already described for Figure 5 are triggered at node *B*.

5 Implementation

The GSN implementation consists of the GSN-CORE, implemented in Java, and the platform-specific GSN-WRAPPERS, implemented in Java, C, and C++, depending on the available toolkits for accessing specific types of sensors or sensor networks. The implementation currently has approximately 20,000 lines of code and is available from SourceForge (http://globalsn.sourceforge.net/). GSN is implemented to be highly modular in order to be deployable on various hardware platforms from workstations to small programmable PDAs, i.e., depending on the specific platforms only a subset of modules may be used. GSN also includes visualization systems for plotting data and visualizing the network structure. In the following sections we are going to discuss some of the key aspects of the GSN implementation

5.1 Adding new sensor platforms

For deploying a virtual sensor the user only has to specify an XML deployment descriptor as described in Section 2, if GSN already includes software support for the concerned hardware/software. Adding a new type of sensor or sensor network can be done by supplying a Java wrapper conforming to the GSN API. At the moment GSN provides the following wrappers:

HTTP generic wrapper is used to receive data from devices via HTTP GET or POST requests, for example, the AXIS206W wireless camera.

Serial forwarder wrapper enables interaction with TinyOS compatible motes. The serial forwarder is the standard access tool for TinyOS provided in the TinyOS package.

USB camera wrapper is used for dealing with cameras connected via USB to the local machine. As USB cameras are very cheap, they are quite popular as sensing devices. The wrapper supports cameras with OV518 and OV511 chips (see http://alpha.dyndns.org/ov511/).

TI-RFID wrapper enables access to Texas Instruments Series 6000 S6700 multi-protocol RFID readers.

WiseNode wrapper supports access to WiseNode sensors (developed by CSEM, Switzerland, http://www.csem.ch/).

Generic UDP wrapper can be used for any device using the UDP protocol to send data.

Generic serial wrapper supports sensing devices which send data through the serial port.

Additionally, we provide template implementations for standard cases and frequently used platforms. If wrapper implementations are shared publicly this also facilitates building a reusable code base for virtually any sensor platform. The effort to implement wrappers is quite low. Table 1 shows the sizes of the wrappers currently shipped with GSN in terms of lines of source code (mainly in Java).

Wrapper type	Lines of code
TinyOS	120
WiseNode	75
Generic UDP	45
Generic serial	180
Wired camera	300
Wireless camera (HTTP)	60
RFID reader (TI)	50

Table 1. Code sizes of wrappers

New wrappers can be added to GSN without having to rebuild or modify the GSN container (plug-and-play). Upon startup GSN locates the wrapper classes and loads them into the system. After that the wrappers for which virtual sensors have been defined locally are initialized while unused wrappers do not consume resources. Wrappers can also be parameterized, so that a virtual sensor can provide initialization parameters to the wrapper, e.g., the acceptable packet format for TinyOS.

5.2 Dynamic resource management

The highly dynamic processing environment we target with GSN requires adaptive dynamic resource management to allow the system to quickly react to changing processing needs and environmental conditions. Dynamic resource management accomplishes three main tasks:

Resource sharing: As the user can modify/remove/add virtual sensors on-the-fly during runtime, the system needs to keep track of all resources used by the individual virtual sensors and enforce resource sharing among sensors (wrappers) where possible.

Failure management: If GSN detects a faulty virtual sensor or wrapper, e.g., by runtime exceptions, GSN undeploys it and releases the associated resources.

Explicit resource control: The user can specify explicit memory and processing requirements and restrictions. While restrictions are always enforced, requirements are handled depending of the globally available resources of the GSN instance. GSN tries to share the available resources in a fair way taking into account the explicitly specified resource requirements, if provided.

Dynamic resource management is performed at several levels in GSN as shown in Figure 7. Separating the resource sharing into several layers logically decouples the requirements and allows us to achieve a higher level of reuse of resources. In the following we will discuss the different levels.

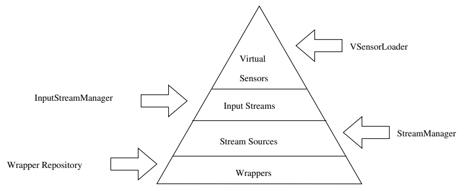

Fig. 7. Hierarchical resource sharing in GSN

Wrapper sharing. Wrappers communicate directly with the sensors which involves expensive I/O operations via a serial connection or wireless/wired network communication. To minimize the costs incurred by these operations GSN shares wrappers among virtual sensors accessing the same physical/virtual sensors. To do so each GSN node maintains a repository of active wrappers. If a new virtual sensor is deployed, the node first checks with the wrapper repository whether an identical wrapper already exists, i.e., wrapper name and initialization parameters of the `<wrapper>` element in the virtual sensor definitions are identical. If a match is found, the new virtual sensor is registered to the existing wrapper as a consumer. If not, a new wrapper instance is created and registered with the wrapper repository. In the case of remote sensor accesses this strategy is applied at both the sending and receiving sides to maximize the sharing, i.e., multiple virtual sensors on one GSN node share a wrapper for the same remote sensor and on the node hosting the sensor the wrapper is shared among all nodes accessing it.

Data sharing. The raw input data produced by the wrappers is processed and filtered by the stream sources to generate the actual input data for the input streams of a virtual sensor. For this purpose a stream source defines what part of the raw input data is used by the associated stream source query to produce the stream source's output data, i.e., by defining the available storage, sampling rates, and window sizes a view on the raw data is defined on which the stream source query is executed. In terms of the implementation each wrapper is assigned a storage holding the raw data and stream source queries are then defined as SQL views on this data store.

This has a number of advantages: (1) It minimizes the storage consumption as raw data is only stored once. Especially if the sensor data is large, e.g., image data, this is relevant. (2) If the sensor data comes from a power-constrained or slow device, power is conserved and processing is sped up. (3) Different processing strategies can be applied to the same data without having to replicate it, for example, image enhancement algorithms and object detection can use the same raw image data.

In the same way as a wrapper can be shared by multiple stream sources, a stream source can also be shared among multiple input streams at a higher level, and input streams in turn are shared by multiple virtual sensors. In essence each of the layers in Figure 7 can be viewed as a resource pool where each of the individual resources in the pool can be shared among multiple resources at the next higher level. Conversely, each higher level resource can also use any number of lower level resources.

5.3 Query planning and execution

In GSN each virtual sensor corresponds to a database table and each sensor reading corresponds to a new tuple in the related table. As we use a standard SQL database as our low-level query processing engine, the question is how to represent the streaming logic in a form understandable for a standard database engine (as already described, GSN separates the stream processing directives from the query). We address this problem by using a query translator which gets an SQL query and the stream processing directives as provided in the virtual sensor definition as inputs and translates this into a query executable in a standard database. The query translator relies on special support functions which emulate stream-oriented constructs in a database. These support functions are dependent on the database used and are provided by GSN (currently we support HSQLDB and MySQL). Translated queries are cached for subsequent use.

Upon deployment of a virtual sensor VS, all queries Q_i contained in its specification are extracted. Each query $Q_i(VS_1, \dots, VS_n)$ accesses one or more relations $VS_1, \dots, VS_n$ which correspond to virtual sensors. Then the query translator translates each $Q_i(VS_1, \dots, VS_n)$ into an executable query $Q_i^t(VS_1, \dots, VS_n)$ as described above and each $Q_i^t(VS_1, \dots, VS_n)$ is declared as a view in the database with a unique identifier Id_i. This means whenever a new tuple, i.e., sensor reading, is added to the database, the concerned views will automatically be updated by the database. Additionally, a tuple (VS_j, Id_i, VS) for each $VS_j \in VS_1, \dots, VS_n$ is added to a special view registration table. This procedure is done once when a virtual sensor is deployed.

With this setup it is now simple to execute queries over the data streams produced by virtual sensors: As soon a new sensor reading for a virtual sensor VS_d becomes available, it is entered into the according database relation. Then the database server queries the registration table using VS_d as the key and gets all identifiers Id_r registered for new data of VS_d. Then simply all views V_r affected by the new data item can be retrieved using the Id_r and all V_r can be queried using a `SELECT * FROM V_r` statement and the resulting data can be returned to the virtual sensor containing V_r (third column in the registration table). Since views are automatically updated by the database querying them is efficient. However, with many registered views (thousands or more) scalability may suffer. Thus GSN does not produce an individual query for each view but merges all queries into a large select statement, and the result will then be joined with the view registration table on the view identifier. Thus the result will hold tuples that identify the virtual sensor to notify of the new data. The reasons for applying this strategy are that (1) database connections are expensive, (2) with increasing number of clients and virtual sensor definitions, the probability of overlaps in the result sets increases which automatically will be exploited by the database's query processor, and (3) query execution in the database is expensive, so one large query is much less costly than many (possibly thousands) small ones.

Immediate notification of new sensor data is currently implemented in GSN and is an eager strategy. As an alternative also a lazy strategy could be used where the query execution would only take place when the GSN instance requests it from the database, for example, periodically at regular intervals. In practice the former can be implemented using views or triggers and the latter can be implemented using inner selects or stored procedures.

6 Evaluation

GSN aims at providing a zero-programming and efficient infrastructure for large-scale interconnected sensor networks. To justify this claim we experimentally evaluate the throughput of the local sensor data processing and the performance and scalability of query processing as the key influencing factors. As virtual sensors are addressed explicitly and GSN nodes communicate directly in a point-to-point (peer-to-peer) style, we can reasonably extrapolate the experimental results presented in this section to larger network sizes. For our experiments, we used the setup shown in Figure 8.

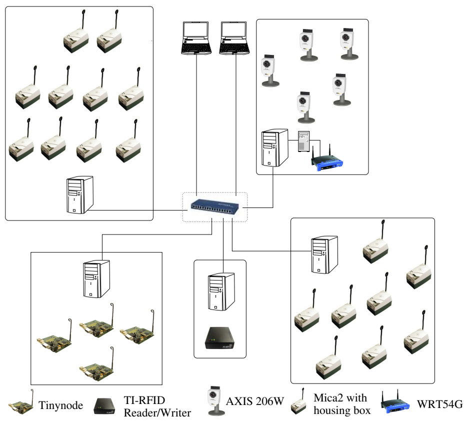

Fig. 8. Experimental setup

The GSN network consisted of 5 standard Dell desktop PCs with Pentium 4, 3.2GHz Intel processors with 1MB cache, 1GB memory, 100Mbit Ethernet, running Debian 3.1 Linux with an unmodified kernel 2.4.27. For the storage layer use standard MySQL 5.18. The PCs were attached to the following sensor networks as shown in Figure 8.

– A sensor network consisting of 10 Mica2 motes, each mote being equipped with light and temperature sensors. The packet size was configured to 15 Bytes (data portion excluding the headers).
– A sensor network consisting of 8 Mica2 motes, each equipped with light, temperature, acceleration, and sound sensors. The packet size was configured to 100 Bytes (data portion excluding the headers). The maximum possible packet size for TinyOS 1.x packets of the current TinyOS implementation is 128 bytes (including headers).
– A sensor network consisting of 4 Tiny-Nodes (TinyOS compatible motes produced by Shockfish, http://www.shockfish.com/), each equipped with a light and two temperature sensors with TinyOS standard packet size of 29 Bytes.
– 15 Wireless network cameras (AXIS 206W) which can capture 640x480 JPEG pictures with a rate of 30 frames per second. 5 cameras use the highest available compression (16kB average image size), 5 use medium compression (32kB average image size), and 5 use no compression (75kB average image size). The cameras are connected to a Linksys WRT54G wireless access point via 802.11b and the access point is connected via 100Mbit Ethernet to a GSN node.
– A Texas Instruments Series 6000 S6700 multi-protocol RFID reader with three different kind of tags, which can keep up to 8KB of data. 128 Bytes capacity.

The motes in each sensor network form a sensor network and routing among the motes is done with the surge multi-hop ad-hoc routing algorithm provided by TinyOS.

6.1 Internal processing time

In the first experiment we wanted to determine the internal processing time a GSN node requires for processing sensor readings, i.e., the time interval when the wrapper gets the sensor data until the data can be provided to clients by the associated virtual sensor. This delay depends on the size of the sensor data and the rate at which the data is produced, but is independent of the number of clients wanting to receive the sensor data. Thus it is a lower bound and characterizes the efficiency of the implementation.

We configured the 22 motes and 15 cameras to produce data every 10, 25, 50, 100, 250, 500, and 1000 milliseconds. As the cameras have a maximum rate of 30 frames/second, i.e., a frame every 33 milliseconds, we added a proxy between the GSN node and the WRT54G access point which repeated the last available frame in order to reach a frame interval of 10 milliseconds. All GSN instances used the Sun Java Virtual Machine (1.5.0 update 6) with memory restricted to 64MB.

The experiment was conducted as follows: All motes and cameras were set to the same rate and produced data for 8 hours and we measured the processing delay. This was repeated 3 times for each rate and the measurements were averaged. Figure 9 shows the results of the experiment for the different data sizes produced by the motes and the cameras.

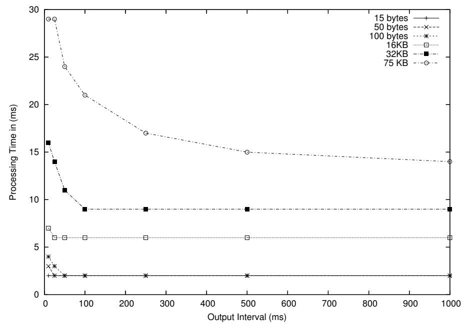

Fig. 9. GSN node under time-triggered load

High data rates put some stress on the system but the absolute delays are still quite tolerable. The delays drop sharply if the interval is increased and then converge to a nearly constant time at a rate of approximately 4 readings/second or less. This result shows that GSN can tolerate high rates and incurs low overhead for realistic rates as in practical sensor deployments lower rates are more probable due to energy constraints of the sensor devices while still being able to deal also with high rates.

6.2 Scalability in the number of queries and clients

In this experiment the goal was to measure GSN's scalability in the number of clients and queries. To do so, we used two 1.8 GHz Centrino laptops with 1GB memory as shown in Figure 8 which each ran 250 lightweight GSN instances. The lightweight GSN instance only included those components that we needed for the experiment. Each GSN-light instance used a random query generator to generate queries with varying table names, varying filtering condition complexity, and varying configuration parameters such as history size, sampling rate, etc. For the experiments we configured the query generator to produce random queries with 3 filtering predicates in the `where` clause on average, using random history sizes from 1 second up to 30 minutes and uniformly distributed random sampling rates (seconds) in the interval $[0.01, 1]$.

Then we configured the motes such that they produce a measurement each second but would deliver it with a probability $P < 1$, i.e., a reading would be dropped with probability $1 - P > 0$. Additionally, each mote could produce a burst of R readings at the highest possible speed depending on the hardware with probability $B > 0$, where R is a uniformly random integer from the interval $[1, 100]$. I.e., a burst would occur with a probability of $P * B$ and would produce randomly 1 up to 100 data items. In the experiments we used $P = 0.85$ and $B = 0.3$. On the desktops we used MySQL as the database with the recommended configuration for large memory systems. Figure 10 shows two results for stream element sizes (SES) of 30 Bytes and 32KB.

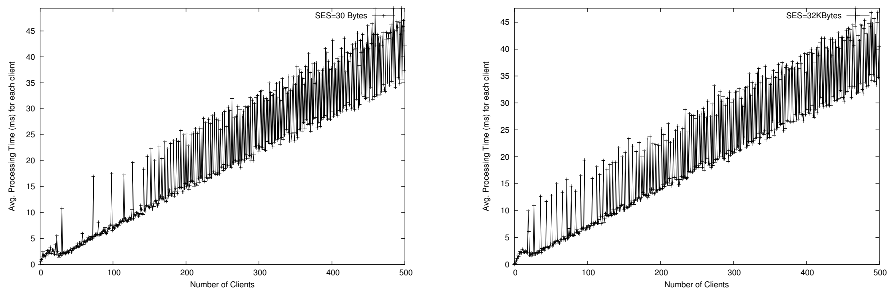

Fig. 10. Query processing latencies in a GSN node

The spikes in the graphs are bursts as described above. Basically this experiment measures the performance of the database server under various loads which heavily depends on the used database. As expected the database server's performance is directly related to the number of the clients as with the increasing number of clients more queries are sent to the database and also the cost of the query compiling increases. Nevertheless, the query processing time is reasonably low as the graphs show that the average time to process a query if 500 clients issue queries is less than 50 milliseconds. If required, a cluster could be used to the improve query processing times which is supported by most of the existing databases already.

In the next experiment shown in Figure 11 we look at the average processing time for a client excluding the query processing part. In this experiment we used $P = 0.85$, $B = 0.05$, and R is as above. We can make three interesting observations from Figure 11:

1. GSN only allocates resources for virtual sensors that are being used. The left side of the graph shows the situation when the first clients arrive and use virtual sensors. The system has to instantiate the virtual sensor and activates the necessary resources for query processing, notification, connection caching, etc. Thus for the first clients to arrive average processing times are a bit higher. CPU usage is around 34% in this interval. After a short time (around 30 clients) the initialization phase is over and the average processing time decreases as the newly arriving clients can already use the services in place. CPU usage then drops to around 12%.

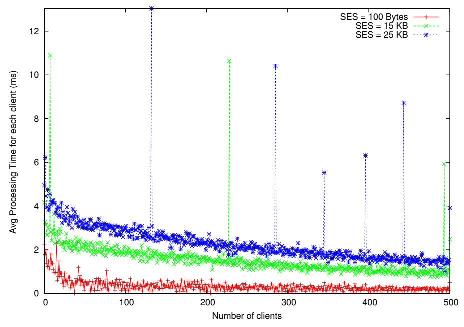

Fig. 11. Processing time per client

2. Again the spikes in the graph relate to bursts. Although the processing time increases considerably during the bursts, the system immediately restores its normal behavior with low processing times when the bursts are over, i.e., it is very responsive and quickly adopts to varying loads.
3. As the number of clients increases, the average processing time for each client decreases. This is due to GSN's data sharing functionalities described in Section 5.2. As the number of clients increases, also the probability of using common resources and data items grows.

7 Related work

So far only few architectures to support interconnected sensor networks exist. Sgroi et al. [5] suggest basic abstractions, a standard set of services, and an API to free application developers from the details of the underlying sensor networks. However, the focus is on systematic definition and classification of abstractions and services, while GSN takes a more general view and provides not only APIs but a complete query processing and management infrastructure with a declarative language interface.

Hourglass [6] provides an Internet-based infrastructure for connecting sensor networks to applications and offers topic-based discovery and data-processing services. Similar to GSN it tries to hide internals of sensors from the user but focuses on maintaining quality of service of data streams in the presence of disconnections while GSN is more targeted at flexible configurations, general abstractions, and distributed query support.

HiFi [7] provides efficient, hierarchical data stream query processing to acquire, filter, and aggregate data from multiple devices in a static environment while GSN takes a peer-to-peer perspective assuming a dynamic environment and allowing any node to be a data source, data sink, or data aggregator.

IrisNet [8] proposes a two-tier architecture consisting of sensing agents (SA) which collect and pre-process sensor data and organizing agents (OA) which store sensor data in a hierarchical, distributed XML database. This database is modeled after the design of the Internet DNS and supports XPath queries. In contrast to that, GSN follows a symmetric peer-to-peer approach as already mentioned and supports relational queries using SQL.

Rooney et al. [9] propose so-called EdgeServers to integrate sensor networks into enterprise networks. EdgeServers filter and aggregate raw sensor data (using application specific code) to reduce the amount of data forwarded to application servers. The system uses publish/subscribe style communication and also includes specialized protocols for the integration of sensor networks. While GSN provides a general-purpose infrastructure for sensor network deployment and distributed query processing, the EdgeServer system targets enterprise networks with application-based customization to reduce sensor data traffic in closed environments.

Besides these architectures, a large number of systems for query processing in sensor networks exist. Aurora [1] (Brandeis University, Braun University, MIT), STREAM [2] (Stanford), TelegraphCQ [4] (UC Berkeley), and Cougar [3] (Cornell) have already been discussed and related to GSN in Section 3.

In the Medusa distributed stream-processing system [10], Aurora is being used as the processing engine on each of the participating nodes. Medusa takes Aurora queries and distributes them across multiple nodes and particularly focuses on load management using economic principles and high availability issues. The Borealis stream processing engine [11] is based on the work in Medusa and Aurora and supports dynamic query modification, dynamic revision of query results, and flexible optimization. These systems focus on (distributed) query processing only, which is only one specific component of GSN, and focus on sensor heavy and server heavy application domains.

Additionally, several systems providing publish/subscribe-style query processing comparable to GSN exist, for example, [12].

8 Conclusions

The full potential of sensor networks will be unleashed as soon as they can be deployed and integrated easily. At the moment these are cumbersome tasks due to the many heterogeneous hardware and software platforms in the sensor network domain. To alleviate the problem and enable fast and flexible deployment and interconnection of sensor networks we have presented our Global Sensor Networks (GSN) middleware. GSN hides arbitrary stream data sources behind its virtual sensor abstraction and provides simple and uniform access to the host of heterogeneous technologies available, through powerful declarative specification and query tools which support zero-programming integration of sensor networks and on-the-fly configuration and adaptation of the running system. GSN's architecture is highly modular in order to be deployable on various hardware platforms. The experimental evaluations of GSN demonstrate that the implementation is highly efficient, offers good performance even under high loads and scales gracefully in the number of nodes, queries and query complexity. GSN is implemented in Java and C/C++ and is available from SourceForge at http://globalsn.sourcefourge.net/.

References

1. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.B.: Scalable Distributed Stream Processing. In: CIDR. (2003)
2. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom., J.: STREAM: The Stanford Data Stream Management System. In: Data-Stream Management: Processing High-Speed Data Streams. Springer (2006)
3. Yao, Y., Gehrke, J.: Query Processing in Sensor Networks. In: CIDR. (2003)

4. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: CIDR. (2003)
5. Sgroi, M., Wolisz, A., Sangiovanni-Vincentelli, A., Rabaey, J.M.: A service-based universal application interface for ad hoc wireless sensor and actuator networks. In: Ambient Intelligence. Springer Verlag (2005)
6. Shneidman, J., Pietzuch, P., Ledlie, J., Roussopoulos, M., Seltzer, M., Welsh, M.: Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard University, EECS (2004) http://www.eecs.harvard.edu/~syrah/hourglass/papers/tr2104.pdf.
7. Franklin, M., Jeffery, S., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., Hong, W.: Design Considerations for High Fan-in Systems: The HiFi Approach. In: CIDR. (2005)
8. Gibbons, P.B., Karp, B., Ke, Y., Nath, S., Seshan, S.: IrisNet: An Architecture for a World-Wide Sensor Web. IEEE Pervasive Computing **2**(4) (2003)
9. Rooney, S., Bauer, D., Scotton, P.: Techniques for Integrating Sensors into the Enterprise Network. IEEE eTransactions on Network and Service Management **2**(1) (2006)
10. Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa Projects. Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society (2003)
11. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The Design of the Borealis Stream Processing Engine. In: CIDR. (2005)
12. Gray, A.J.G., Nutt, W.: A Data Stream Publish/Subscribe Architecture with Self-adapting Queries. In: International Conference on Cooperative Information Systems (CoopIS). (2005)