The Stream System: a Data Collection and Communication Abstraction for Sensor Networks

Giuseppe Amato*, Paolo Baronti*, Stefano Chessa*† and Valentina Masi*
*Istituto di Scienza e Tecnologie dell'Informazione
Area della Ricerca, CNR di Pisa-S.Cataldo - 56124 Pisa, Italy
†Computer Science Department, University of Pisa
Largo B. Pontecorvo 3 - 56127 Pisa, Italy

***Abstract*— Sensor network software is still in its youth. Due to sensor hardware limitations and the highly specific nature of application domains, existing software is generally poorly structured. It mixes data collection activities with data management, data storage/retrieval and intra/inter-sensor data communication with the actual data processing required by the application.**

We identify data collection, intra-sensor and inter-sensor communication as recurring activities in sensor network applications and propose a software module that abstracts these activities: the Stream System.

Applications developed on top of if can be organized as a set of computational entities that are provided with a uniform view of such activities through the concept of streams. Streams represent a flow of data records that either (i) originate from a local transducer and can be read by a local entity, (ii) originate from a local entity and can be read by another local entity or (iii) originate from a local entity and can be read by a remote entity i.e., one running on a different sensor.

Applications disregard the actual implementation details of collecting transducer readings and passing such data to other local or remote computational entities and rely on the Stream System to provide a record oriented data service in this respect.

I. Introduction

Sensor Networks are a special kind of wireless ad hoc network where nodes (sensors) are small devices with extremely constrained resources. They are equipped with a small amount of memory, a set of transducers, a low range, low bandwidth radio transceiver and are powered by on board batteries. Table I presents hardware characteristics for the Crossbow mica2 mote: a low end sensor widely used in research fields [1].

Microcontroller	ATMel ATmega128L 7.37 MHz
Instruction memory	128 KB
Data memory	4 KB
Secondary storage	512 KB Flash memory
Radio	Variable range 315/433/915 MHz
Bandwidth	50 Kbs gross data rate
Power	2 AA batteries
Typical transducers	Light, temperature, magnetism, audio, acceleration

TABLE I
CROSSBOW MICA2 HARDWARE CHARACTERISTICS.

The distinguishing characteristic of sensor hardware is the possibility of sampling light, temperature, magnetism, humidity, acceleration, etc. from the surrounding environment and sending this data to neighbors.

Trivial data gathering applications request sensors to periodically sample a certain phenomena and forward these readings to a special *sink* node that is connected to a PC by special hardware (e.g., a wire, a high power radio link, a satellite link). The actual data processing happens on the PC. A problem with this approach is that it requires a high level of message generation/relay in the network. Since battery power is a scarce resource and the radio interface is the most power-hungry device on the sensor, it is extremely important to limit message exchange as much as possible.

A way to do this is to move (at least a part of) the computation into the sensor network. This solution saves energy because (i) it reduces the number of messages sent given that many of them can be replaced by one which carries some computed value (e.g., an aggregate) and (ii) executing instructions (i.e., carrying out computations) is less expensive than sending bits over the radio. This approach, adopted in systems like TinyDB [2], Cougar [3], directed diffusion [4], requires that the PC operator inject complex tasks (known as queries in database oriented applications) into the network, assigning data collection, computation and communication activities to the sensors.

To this aim, a careful design of software modules, to run on sensor nodes, is really important to guarantee sensor network reconfigurability and flexibility in developing and deploying new applications.

However, as discussed in next section, existing applications have often a monolithic architecture, which make it difficult to adapt them to new environments, situations, and applicative requirements, and to test new solutions for the various services required for their functionalities.

In this paper we identify the basic activities needed for data acquisition, manipulation, and delivery. This activities are centralized in a single module, the *Stream System*, which can be thought as the equivalent of the file system in traditional computer systems. The Stream System functionalities reflect the high dynamic variability of data and distributed

Work funded in part by the European Commission in the framework of the "SatNEx" NoE project (contract N. 507052), and by MIUR in the framework of the Italian project "IS-MANET".

access/processing observed in wireless sensor networks, and it can be used as a building block for higher level applications, such as sensed data management softwares, sensor databases, etc.

The rest of the paper is organized as follows: Section II gives motivations for the Stream System development and use. Section III introduces the Stream System and describes the interface it offers to an application module. We give implementation guidelines in Section IV and present some details for a real implementation on sensor hardware in Section V. We finally draw the conclusions in Section VI.

II. Motivation

Existing sensor network applications are rather monolithic in the sense that they do not structure and do not logically divide data acquisition, data management, data processing and data communication tasks. Due to resource (memory) limitations and highly application specific uses of sensor networks, existing code tends to freely mix the above tasks.

We believe that identifying common activities and defining lightweight modules that carry them out in a general, application independent, way is a step toward easier sensor software development and reuse. We actually implemented this module on the TinyOs/nesC platform as described in Section V.

The Stream System offers an unidirectional data collection and data communication abstraction to higher layers. The basic concept is that of a *stream*. It represents a generic unidirectional data channel that is able to carry data. The Stream System provides functionalities for creating and destroying streams and for writing and retrieving records to/from existing streams. Streams are also used to access data acquired by transducers.

The context where we envision the use of streams is that of *operator-driven* computations (or *dataflow-like* computations). Operators can be thought of as independent agents that have one or more inputs, perform some operations on those inputs and possibly produce an output. In our model, operator inputs are data read from streams, and the operator outputs are data they write to a stream. By defining different operator types and interconnecting them with streams, we can construct complex computation/data processing tasks. Figure 1 illustrates this concept with a simple graphical representation of a computation based on streams and operators. Circles represent operators and lines connecting them represent streams. The computation diagram is organized as a tree with leaf nodes producing data records, interior nodes being full fledged operators with inputs and output and the root corresponding to the result of the computation.

One possible application based on operators and the use of the Stream System is a distributed query processor for sensor networks based on the relational model [5]. In [6] we describe an early implementation of such a system where relational operators like selection (σ), projection (π), join ($\bowtie$) and spatial and temporal averages, were implemented relying on the Stream System that we present here. Streams, in this system, were used to implement dynamically varying versions

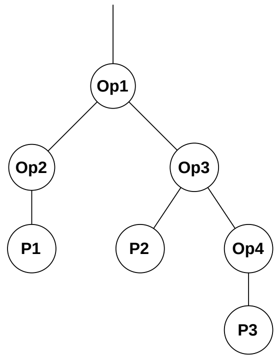

Fig. 1. Computation model with streams and operators. P nodes produce data records and Op nodes are operators which take input records and produce an output record.

of *tables* in the relational model. Operators act on stream records as (similarly to) the corresponding relational algebra operators act on table records. Combining these operators with streams we have been able to process relational algebra queries in a sensor network.

Figure 2 illustrates an example query, expressed in the relational algebra, involving two sensors and employing streams and operators. On sensor S1 two sensor streams(S) (see

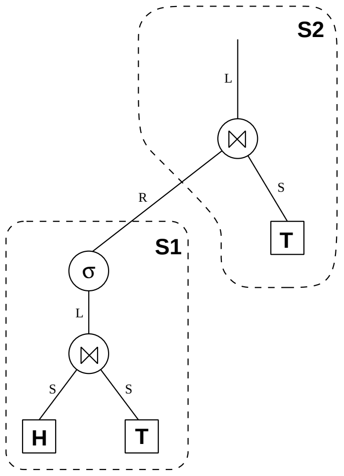

Fig. 2. Example relational algebra query with operators and streams. S = sensor stream, L = local stream, R = remote stream, H = humidity transducer, T = temperature transducer.

Section III for a description of the various stream types) are used to pass humidity (H) and temperature (T) readings to a join operator ($\bowtie$). A local stream (L) carries the output of the latter operator to a selection operator (σ), to filter the records on the basis of some predicate. The output crosses the network on a remote stream (R) to sensor S2 where a join operator combines such records with the result of local sampling of temperature through a sensor stream.

III. The Stream System

The Stream System offers three types of streams: *sensor* streams, *local* streams and *remote* streams. *Sensor* streams are the basic abstraction for collecting readings from transducers. Sensor streams can only be read from operators since the writing is carried out by the associated transducers (these can be thought of as virtual operators writing to sensor streams). Local streams represent a local data channel. Operations to write records to and read records from the stream must occur on the same sensor where the stream was created. In terms of the operator concept from Section II, both the writing and reading operators must reside on the local node. *Remote* streams require cooperation between two nodes since they intend to provide a data channel between two different nodes. Write operations can be carried out on one of them (the stream write-end) and read operations can take place on the other (the read-end). Figure 3 illustrates these concepts.

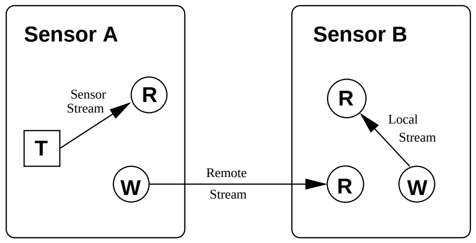

Fig. 3. Stream types. T = transducer, W = writing operator, R = reading operator.

If we make a comparison to traditional computer systems, sensor streams play the role of files, which applications read data from. Local streams play the role of pipes, for communication between local tasks. Remote streams play the role of sockets, for communication between tasks executed on different nodes.

The three stream types have peculiar characteristics which relate to their different semantic roles. The Stream System offers a set of homogeneous functionalities defined on all three stream types, thus providing a common abstraction.

In the following we use a command/event-based notation to define streams and their operation. A key concept in command/event-based systems, that we use in our specification, is that of *split-phase* operations. A split-phase operation is executed in two phases. First, a request to start the operation is made: the requests return immediately, without waiting for the operation to be completed. Second, when the requested operation is completed an event is fired to notify it. Split-phase operations are typically used when operations execution involves interacting with hardware devices or in case of long running operations, to allow an asynchronous interaction. Operations involving simple data structure manipulation can immediately complete without requiring a later event notification (i.e., they are not split-phase).

An application identifies a stream through its stream descriptor, which is returned by a stream constructor invocation. Stream constructors are specific to the various types of streams and will be discussed later on. Various operations can be applied to a stream by means of its descriptor. With command

```
command write(
     stream_desc,
     buffer,
     length
);
```

the user writes a record to a stream. A stream maintains data in a finite size queue. The write operation appends a value to the end of the queue. Whenever a write operation finds the queue full, it simply discards the first queue element before appending its own (which actually corresponds to evicting the oldest data).

The stream descriptor is supplied through argument `stream_desc` while arguments `buffer` and `length` give the starting address of the buffer containing the data and its length in bytes, respectively. Data stored in a stream can be accessed by means of the read operation

```
command read(
     stream_desc
);
```

where argument `stream_desc` identifies the stream, as usual. Read operation is split-phase: read request immediately returns; if data are available in the stream queue, the Stream System returns the oldest one to the user by signaling event `readDone()`, elsewhere the event will be fired as soon as new data are available. When the event is signaled, the read element is removed from the queue. Event `readDone()` is defined as

```
event readDone(
     stream_desc,
     buffer,
     length
);
```

Argument `stream_desc` identifies the stream while `buffer` and `length` pass the actual data pointer and its byte length.

Finally, when a stream is no longer needed, it must be destroyed with command

```
command close(
     stream_desc
);
```

The following Subsections give additional information on the operations permitted and defines the three types of streams: sensor streams, local streams, and remote streams.

A. Sensor Streams

Sensor streams provide an abstraction for collecting readings from local transducers. It is not possible for an application to write to a sensor stream since its write-end is automatically associated with a transducer. It is the Stream System that writes data to the stream on the basis of readings collected from the transducer. Sensor streams come in two flavors: *periodic* and *on-demand*.

Periodic sensor streams unconditionally collect readings from the associated transducer with a fixed rate. The sampling activity is driven by a timer that the Stream System automatically restarts when it fires. Upon timer expiration, the Stream System commands a reading from the transducer and writes a record of 3 elements to the stream: (*nid*, *ts*, *val*). *nid* is the (network unique) node id, *ts* is a timestamp of the reading and *val* is the actual reading. The user can later retrieve these records using command `read()` and receiving event `readDone()`.

On-demand sensor streams only perform a reading from the transducer when explicitly requested to do so i.e., when the user requests to read a record from the stream. When the transducer supplies its reading, the Stream System writes a record (the content is the same as for periodic sensor streams) to the stream and immediately returns it to the user signaling event `readDone()`.

The sensor stream constructor is

```
command stream_desc openS(
      transducer,
      queue_size,
      sampling_type,
      sampling_rate
);
```

Argument `transducer` encodes one of the transducers available on the node (e.g., light, temperature, humidity, acceleration, magnetism, etc.). `queue_size` gives the size of the queue data structure used to store data records. The stream type (i.e., either periodic or on demand) is specified through argument `sampling_type`. If the user requests a periodic sensor stream, it must also supply the sampling rate in `sampling_rate`. The call immediately returns the assigned stream descriptor.

B. Local Streams

The read-end and write-end of a local streams are on the same node. An application can both write to and read from a local stream. Of course, different tasks of the applications are expected to distinctively read and write records to such streams.

Creation of a local stream is achieved through command

```
command stream_desc openL(
      queue_size
);
```

that immediately returns the assigned stream descriptor. As for sensor streams, argument `queue_size` gives the size of the queue.

C. Remote Streams

Remote streams are by far more complicated. They interconnect two distinct sensors so writing can only happen on one of them (the write-end) and reading can only happen on the other (the read-end). The complications in the interface commands/events used for opening a remote stream stem from the fact that both endpoint sensors must be aware of it.

To request the opening of a remote stream from sensor A to sensor B, the application modules on both sensors must create their own stream end with command

```
command openR(
      destination,
      queue_size,
      symbolic_id
);
```

The Stream System takes care of interacting with a lower level Network module and setting up the remote stream. Section IV gives more details on what happens undercover.

In the call happening on sensor A, argument `destination` gives the identity of remote sensor B. Various approaches could be used to identify nodes. Among them, node ids, coordinates (either physical or virtual), geographic areas, roles.

On sensor B, `destination` simply identifies the local node. From an implementation point of view, the Stream System module on sensor A (the stream write-end) turns to the Network module asking it to set up a communication channel to sensor B. On the other hand the Stream System module on sensor B (the read-end) plays a passive role (see Section IV-B for more details) and just keeps track of the open request received. Argument `destination` allows the Stream System module to decide whether it is the write-end or the read-end.

Argument `symbolic_id` is the same on both stream endpoints and is used to identify the same stream in the two nodes. Finally, argument `queue_size` gives the size of the queue as explained for sensor and local streams above.

Constructor for remote streams, differently than constructors for local and sensor streams, is split-phase to cope with possible network latencies. In fact, in case of a remote stream, the open requests immediately returns and, when the stream has been set up, the Stream System modules on the two ends independently notify this by signaling event

```
event openRDone(
      stream_desc,
      symbolic_id
);
```

Argument `stream_desc` contains the assigned stream descriptor and `symbolic_id` is needed by the application module to associate the event with the previous `openR()` command.

IV. Implementation Guidelines

This Section discusses some issues related to the implementation of streams and the procedure for setting up and operating a remote stream.

A. The Network Module

In order to implement remote streams, the Stream System module needs assistance from a Network module offering a connection oriented service. Here we present a hypothetical interface for this module. When opening a remote stream, the Stream System asks the Network module to establish a unidirectional communication channel (a connection) to a remote sensor with the following command

```
command channel_id connect(
      destination
);
```

As discussed in Section III-C, there are several options to identify sensors. Argument `destination` serves to identify the remote end. The command returns a locally allocated, network unique, channel id (`channel_id`).

Since the operation outcome depends on the availability of the necessary resources on each node of the path to the destination, it cannot be determined at the time the command call returns but will be signaled later with an event.

The connection establishment procedure can be implemented with a *connect* message going through the network toward the destination and reserving the necessary resources along the way. A *connect-ack* message could then travels back to the originating sensor confirming that the connection has been accepted by the destination Network module. When such a message arrives at the source Network module, the connection establishment procedure terminates, and the Network module signals event

```
event connectDone(
     channel_id
);
```

to informs the user. Argument `channel_id` is used by the Stream System to retrieve data associated with the pending connection request.

The user can now send packets on the established connection invoking command

```
command send(
     channel_id,
     buffer,
     length
);
```

It passes the id of the channel to identify the connection (`channel_id`) as well as a pointer to the data buffer and the corresponding length. The Network module packs this byte array into the MAC message payload (after the network header) and sends the message.

The service offered by the Network module is not necessarily reliable: reliability could be provided by MAC level or application level acknowledgments or be completely missing. Also, sending a packet is not contemplated as a split-phase operation in the network interface. In other words, a successful return from `send()` reports that the Network module has filled one of its message buffers and that it will attempt to send the message on the requested connection. No event is signaled to the user when the message is actually sent.

Upon receiving a message from a remote sensor over an established connection, the Network module must pass the payload up. It achieves this by signaling event

```
event receive(
     channel_id,
     buffer,
     length
);
```

where `buffer` is a pointer to the message payload and `length` is the payload size, in bytes. Argument `channel_id` identifies the channel for the received message.

Finally command

```
command disconnect(
     channel_id
);
```

can be used on the source sensor (write-end) to ask the Network module to shut down (i.e., terminate) an existing connection. Servicing this request implies deallocating channel resources on all nodes of the data path and may be performed by sending a special *disconnect* message along the path or by simply letting a connection activity timer expire on all such sensors. There is no event signaling upon operation completion.

B. Streams

Internally, a stream is implemented as a finite size queue. The number of elements in the queue is specified when the stream is opened, by means of argument `queue_size` to commands `openS()`, `openL()` and `openR()`. In case of local streams and sensor streams the queue data structure is clearly allocated on the same node that put values in it (the `write()` operation for local streams, the transducers for sensor streams).

For remote streams the queue data structure resides on the read-end Stream System module. Thus, writing a record to a remote stream means passing it to the Network module that will transport it to the destination node, which will finally store it in the queue.

In case that an acknowledgement mechanisms is required to advise the writer that data arrived to destination, this should be obtained with an acknowledge event fired by the Stream System module at the destination node and received by the application running in the source node. For brevity we did not model this event in this paper.

The procedure for setting up a remote stream is rather complex since it involves several interactions between the Stream System module and the underlying Network module. Figure 4 depicts the temporal sequence of commands (full arrows) and events (dashed arrows) when a remote stream must be opened from sensor A to sensor B. To avoid excessive cluttering only significant arguments to command/event calls are indicated. Dotted lines indicate message generation in the MAC modules, the actual sending, propagation and network traversal, reception in the destination MAC module and passing up to the Network module. Rectangles report on local data structures as they appear after the immediately preceding command or event completes. In response to an `openR()` command, A's Stream System module invokes command `connect()`, asking the Network module to establish a data channel to B's Network module. `connect()` returns a channel id (c_id in the Figure) and instructs the Network module to send a *connect* message. At this point `openR()` stores the channel id together with the stream symbolic id (s_id) and returns control to the application module.

Upon receiving the *connect* message B's Network module replies with a *connect-ack* message (see Section IV-A) which fires the `connectDone()` events on A. In turn, the

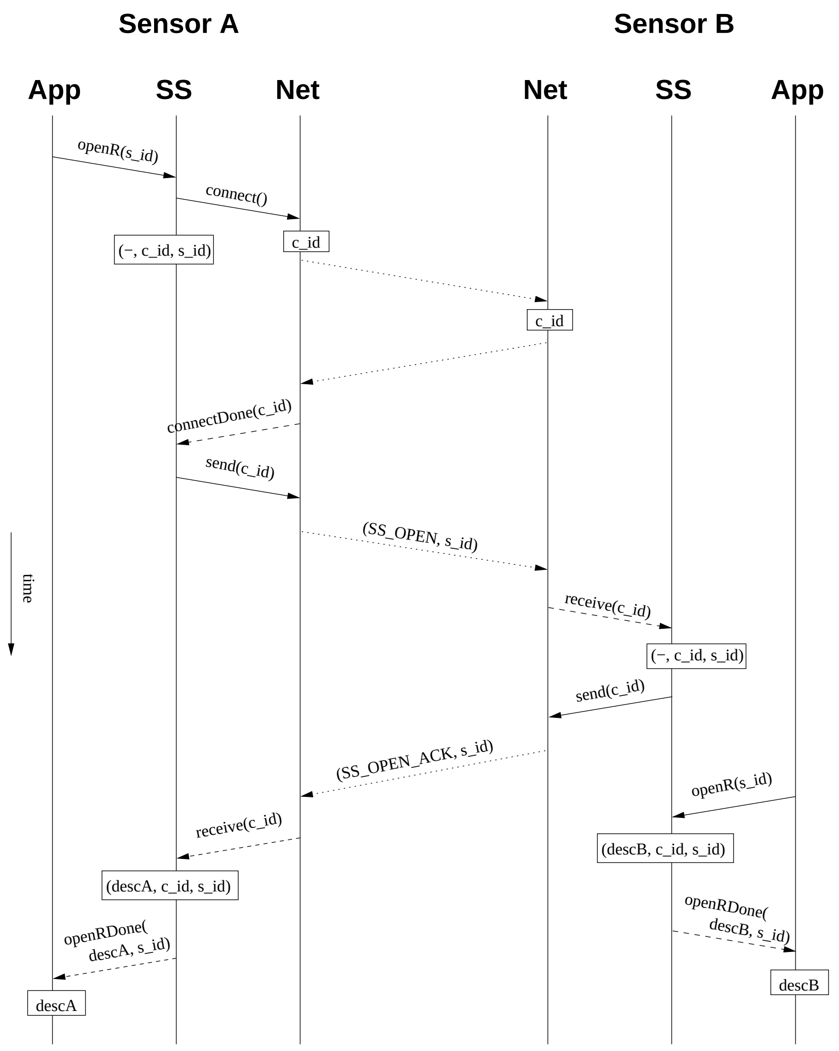

Fig. 4. Command and event sequence when opening a remote stream from sensor A to sensor B (App = Application, SS = Stream System, Net = Network).

`connectDone()` retrieves the data structures for the remote stream by means of the channel id, prepares an `SS_OPEN` message that includes the stream symbolic id and hands it to the Network module for sending on the established channel.

Upon receiving the message, B's Network module signals event `receive()` to B's Stream System module. Assuming B's application module did not request to open the stream yet, the Stream System module records the arrival of the message associating the stream symbolic id with the channel id. It then replies with a `SS_OPEN_ACK` message.

Finally, when A's Network module receives such message it passes it up to the Stream System module which retrieves the stream data structures and signals event `openRDone()` to the application, passing the stream symbolic id and a newly allocated stream descriptor.

When B's application module invokes command `openR()`, the Stream System module discovers that it already received the `SS_OPEN` message from the other side and signals event `openRDone()` with the stream symbolic id and a locally allocated stream descriptor as arguments.

Note that when B's application module invokes `openR()` before the Stream System module has received message `SS_OPEN` from its peer in A, it cannot associate any channel id with it, yet. For this reason it keeps track of the symbolic id (as specified in the `openR()` locally invoked) and, upon arrival of the `SS_OPEN` message from A's Stream System module, B's Stream System signals event `openRDone()` and fills its data structures with the channel id.

Writing a record to a remote stream means sending the record over the network to a remote sensor. The Stream System modules on both stream endpoints interact with the local Network modules in order to implement this operation. Figure 5 illustrates the sequence of calls and events that take place for a remote stream connecting sensor A to sensor B. A's application module writes a record to the remote stream invoking command `write()` and passing the stream descriptor as well as the actual data record. The Stream System looks up its data structures and retrieves the channel id associated with the user supplied stream descriptor. It prepends a header (`SS_WRITE`) indicating the message type and asks

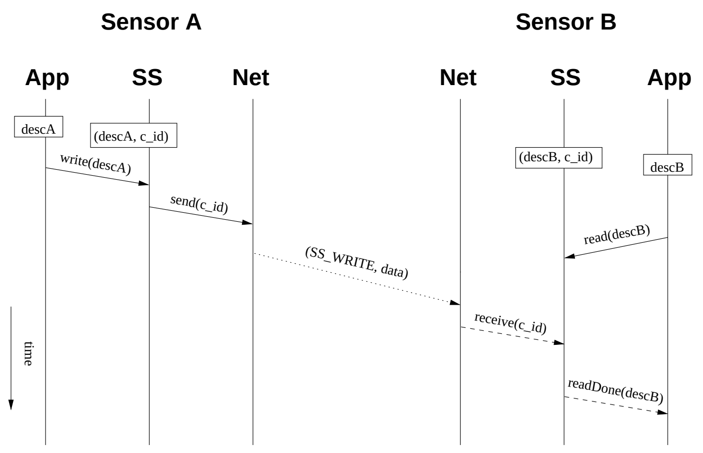

Fig. 5. Command/event sequence when data is written to a remote stream (App = Application, SS = Stream System, Net = Network).

the Network module to send the message over the channel (`send()`).

Upon receiving this message, B's Network module signals event `receive()` passing the channel id as an argument. B's Stream System module retrieves the stream descriptor that is associated to the channel id and writes the record into the stream data structures. If B's application module previously commanded a read from the stream, the Stream System module can now signal event `readDone()`, passing the stream descriptor and the data record as arguments (Figure 5 illustrates this case). Otherwise, the Stream System module simply stores the record in the stream data structure and passes it to the application module when requested with a later `read()` command.

V. The Stream System Implementation

A. The Platform

We implemented the Stream System module and the underlying Network module on the TinyOs/nesC platform (the implementation is available in [7]). nesC [8] is a language that shares its basic constructs with C but defines an event driven programming model. A program is composed of modules that interact with each other through *interfaces*. Modules consist of data structure and function definitions. An interface is a means to characterize the possible interactions between modules. Its listing specifies *commands* and *events* as function prototypes.

A module may *provide* or *use* a certain interface. In the first case the module definition must include the implementation of (provide code for) all the commands defined in the interface. Within the functions it defines, it may *signal* (e.g., call) events from the interface. On the other hand, a module that uses an interface is free to call, within its functions, any of the commands defined in the interface and must implement all the events defined in the interface.

TinyOs [9], [10] is a simple operating system developed at UC Berkeley and written in nesC. Originally conceived for the mica sensor hardware, it now works on several sensor architectures including the Crossbow mica2, mica2dot and micaz series [1] as well as the Moteiv telos and tmote sky series [11]. It consists of a set of nesC modules that provide basic functionalities abstracting the underlying hardware.

A TinyOs/nesC application builds up from a set of modules, some from TinyOs and some from the application programmer, that fit together interacting according to the associated interfaces.

TinyOs does not support true multiprogramming. It only allows a single application but permits several user tasks to be scheduled for execution. Tasks cannot preempt each other: the currently running task can schedule another task for execution (a FIFO discipline is used) but scheduled tasks cannot run until the current task terminates. Hardware interrupts can preempt a running task. Anything beyond basic interrupt processing is done by posting user tasks.

The Stream System is nothing but a nesC module that offers (implements) some functionalities according to a well defined interface and signals events to notify of conditions. It relies on the existence of a Network module implementing a network (lower level) interface and providing interconnections between any two sensors in a multihop network. It also interacts with a Transducer Abstraction Module that abstracts the transducer devices on the sensor, providing commands to read any specific transducer and signaling events when readings are available. Figure 6 illustrates how the Stream System module fits into the system and how it interacts with the other modules.

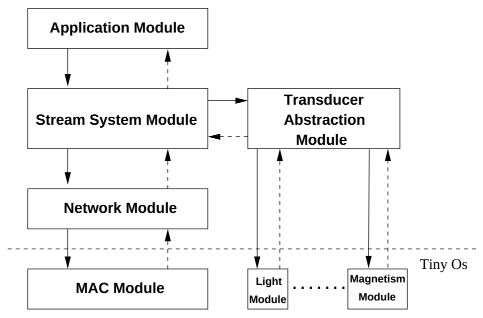

Fig. 6. Stream System Module interaction diagram. Full arrow lines indicate commands while dashed arrow lines indicate events.

B. Network Module Implementation

This Section gives details concerning our implementation of the Network module. Among the possible routing strategies available for wireless ad hoc networks and sensor networks in particular, geographic (greedy) routing [12], [13] is probably the best candidate for sensor networks since it is simple, easy to implement and behaves reasonably well in relatively dense networks. The implementation of our Network module relies on greedy routing for the connection establishment procedure and uses the algorithm described in [14] to assign three-dimensional virtual coordinates to the sensors.

Energy resources are a limiting factor in sensor networks so it is essential to reduce radio operation intervals as much as possible. Several approaches [15], [16] exploit MAC and/or network layer information to reduce energy consumption by scheduling radio activities. We have also designed [17] a novel algorithm that attempts to turn off the radios on the basis of application-provided information. When an application needs to periodically send fixed size/rate packets to another node trough a given path (multi-hop communication), the algorithm exploits communication timing information, transmission times, and average medium access delays, to optimally schedule radio activities in the path.

In order to implement greedy routing we let the application module supply the virtual coordinates of the destination node as argument `destination` to command `openR()`. The Stream System passes it unchanged to the Network module in command `connect()`. To achieve energy efficient communication the application module provides additional arguments to command `open()`, indicating when it will start sending packets on the remote stream, the interval between consecutive packets and the size of each packet. Again, the Stream System simply passes these values to the Network module as additional arguments to `connect()`.

The Network module uses greedy routing to send the *connect* message to the destination when it needs to establish a new connection. All sensors along the path allocate an entry in their connection forwarding table to associate the channel id with a neighbor (the one they forward the message to). The sensors also configure their radio activity intervals as required by the energy saving algorithm.

When the destination sensor receives the *connect* message, it also replies with a *connect-ack* message that returns (along the reverse path) to the originator of the *connect* message and confirms connection establishment.

After a connection has been established the sensors in the path only turn on their radios when the next message is expected and for up to some maximum amount of time. Any message they receive contains the channel id (the coordinates of the destination are no longer needed) which they use to look up their connection forwarding table and correctly forward the message to the next hop in the path.

VI. Conclusions

The Stream System is a data collection and data communication abstraction model suitable for sensor networks. An application built on top of a real implementation can be structured as a set of operators distributed among the nodes of the sensor network. Operators read inputs, do some processing and produce some output. The output of an operator becomes input of another by means of a stream interconnection. A special type of stream serves as a transducer data source. The application totally disregards issues concerning transducer operation as well as moving data from one node to another. We also introduced a real implementation of our model on the TinyOs/nesC platform.

We have implemented a wireless sensor network database relying on this Stream System: operators of the query algebra take as inputs and returns as outputs values respectively from and to streams.

References

[1] Crossbow Technology Inc., http://www.xbow.com.

[2] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The design of an acquisitional query processor for sensor networks." in *SIGMOD Conference*, 2003, pp. 491–502.

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

[4] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks." in *MOBICOM*, 2000, pp. 56–67.

[5] E. Codd, "A relational model for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[6] G. Amato, P. Baronti, and S. Chessa, "MaD-WiSe: Programming and Accessing Data in a Wireless Sensor Network," in *Proceedings of the International Conference on "Computer as a tool" EUROCON 2005*, Belgrade, Serbia & Montenegro, November 2005.

[7] MaD-WiSe, http://mad-wise.isti.cnr.it.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," in *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, San Diego, CA, USA, June 2003, pp. 1–11.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System Architecture Directions for Networked Sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, USA, November 2000, pp. 93–104.

[10] TinyOs Community Forum, http://www.tinyos.net.

[11] Moteiv Corporation, http://www.moteiv.com.

[12] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia, "Routing with Guaranteed Delivery in Ad Hoc Wireless Networks," *ACM Wireless Networks Journal*, vol. 7, no. 6, pp. 609–616, November 2001.

[13] B. Karp and H. T. Kung, "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks," in *Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiCom 2000)*, Boston, MA, USA, August 2000, pp. 243–254.

[14] A. Caruso, S. Chessa, S. De, and A. Urpi, "GPS Free Coordinate Assignment and Routing in Wireless Sensor Networks," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2005)*, Miami, FL, USA, March 2005, pp. 150–160.

[15] W. Ye, J. Heidemann, and D. Estrin, "Medium Access Control With Coordinated Adaptive Sleeping for Wireless Sensor Networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 3, pp. 493–506, June 2004.

[16] G. Lu, N. Sadagopan, B. Krishnamachari, and A. Goel, "Delay Efficient Sleep Scheduling in Wireless Sensor Networks," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2005)*, Miami, FL, USA, March 2005, pp. 2470–2481.

[17] G. Amato, P. Baronti, and S. Chessa, "Connection-Oriented Communication Protocol in Wireless Sensor Networks," Istituto di Scienza e Tecnologie dell'Informazione - CNR, Tech. Rep. 2005-TR-10, 2005.