TinyStream Sensors

Pedro Furtado

University of Coimbra,
Polo II, Coimbra

pnf@dei.uc.pt

Abstract. Stream processing engines have been proposed in the past for handling streaming data coming from data sources. But considering sensor networks, there is a need for an approach that allows stream models to reach also computation-capable constrained embedded devices and to implement storage, exchange and computation on those. We propose a stream model that implements sensor-device data handling. The stream processing abstraction and interface allows small motes to store and process the data locally and to route processed data to consumer streams on-demand. This eliminates the need to code motes operation in lower-level languages, allows easy configuration of operations of different types and saves communication energy. The approach is quite useful in diverse contexts, including wireless sensor networks. We describe the approach and show its advantages experimentally.

Keywords: distributed systems, sensor networks, stream processing.

1 Introduction

Researchers and companies have realized over the years that high-level programming approaches are necessary for increasing the adoption of technologies in practical settings. Stream and complex event processing engines realize that goal for handling streaming data from data sources. However, up to now those engines were not available within constrained embedded devices of sensor networks. Devices such as wireless sensors are small nodes with computation, storage and radio power, but they are programmed in low-level fashion, in spite of being deployed as part of a wider heterogeneous networked system for monitoring and actuating over the physical world. The use of microcontrollers with some processing and storage capabilities allows in-sensor processing and retention of data, and the use of flash memory offers gigabytes of space. Data can be managed inside the embedded node, we can even collect a whole day or month of data before communicating it to some collector node, saving a lot of battery.

What is needed is some high-level programming abstraction that models data storage and retrieval using typical operations, data processing, data routing between producers and consumers, time-ordered operation (e.g. keep a full day of data before computing some statistics and sending it to a consumer stream in some remote work-

station). Stream models provide all these features, fulfilling the data management needs in the heterogeneous sensor networks context. On the implementation perspective, small streams with few samples fit into a few bytes in the small memories of the embedded devices, while larger streams are flash-disk resident. All operations work over both memory and flash streams.

Our TinyStream proposal defines the language, processing engine and efficient implementation of operations in constrained devices. We show how it applies stream processing engine mechanisms to individual sensor nodes as well as the whole distributed system, without the need for application programmers to know low-level programming details of individual devices. This way they concentrate on the application objectives. Our experimental results show that the approach occupies only a conveniently small footprint in constrained embedded devices, which receive simple codified stream commands that are then parsed and executed in the node. Operations are implemented with small memory requirements, in order to run on the constrained devices.

Previous proposals related to this one include complex event processing engines, which are not designed for running in constrained devices and do not fit their limitations, and sensor network middleware such as TinyDB [4] or Cougar [1], which do not manage data inside individual nodes and provide only a system-wide database model, transforming database queries into data collection code forwarding data to the sink. TinyStreams adds the time-ordered stream model – e.g. one can create a stream to collect 10 hours of data, compute a summary and then send it to some remote Smartphone; autonomous node engines and querying – we can create, drop, delete or update streams in memory or flash in individual nodes; as well as easy and powerful stream-based data routing specification as parts of stream commands – we can specify that the stream data should be periodically routed into another remote stream.

The paper is organized as follows: section 2 discusses related work. Section 3 discusses the model and architecture of the approach, then section 4 presents experimental results and section 5 concludes the paper.

2 Related Work

Previous work related to this one includes stream processing engines and high-level programming approaches and middleware for sensor networks.

Stream processing or complex events processing engines have been proposed before in the context of processing high-rate data from data sources. Examples of readily deployable CEP engines include StreamBase and Esper [10,11]. Examples adapted to over the internet integration and processing of data from sensor sources include GSN [12] and Hourglass[13]. However, those engines are not deployable in individual constrained embedded devices (they merely see internet data sources), and there have been no previous works on developing such engines for embedded devices, even though stream models offer extremely useful primitives for time-ordered streams and data routing between streams. There has also been no effort in the past into creating

node-wise stream processing engines that allow direct operation in individual embedded device nodes.

Previous works concerning data processing in sensor networks closest to this one include database-like models TinyDB [4] and Cougar [1]. Both TinyDB and Cougar provide a database front-end to a sensor network by running a small database-like query engine at a sink node. They assume that data is forwarded into the sink for processing, without providing means for explicitly commanding how individual nodes should manage their data. They are therefore not node engines, nor do they apply stream processing to manage and route the data inside the sensor network. TinyStreams, on the contrary, creates streams in individual nodes, can place those in memory or flash, provides operations for individual nodes to manage their data and provides stream producer-consumer data routing primitives for intuitive formulation of stream routing between nodes.

While sensor network resident streams are a novelty, databases resident in embedded devices have been mentioned before for instance in [3] and in PicoDBMS [8] (a small database engine developed for smart cards).

From the perspective of sensor network usage in ubiquitous applications, wireless sensor networks are common nowadays and provide means to deploy easily and extensively sensors and actuators in very disparate application settings. Being able to declaratively configure what such a sensor network is to do and how it sends data to some remote logging station is a very important step forward in the wider adoption of the technology.

Many of the use cases involve collecting sensor data. In those applications sensors send data to workstations. Users must specify when to sample, how much time to keep the data, how to transform the data and when and how to transfer data remotely. With TinyStreams this is easily done by non-experts, users may even wish to keep hours or days of data in the sensor devices themselves and issue queries either ad-hoc or pre-planned to retrieve the data when required. In data logging applications the sensed data is stored and the logs are collected later on. The data can be logged in flash and retrieved later with a query. Keeping data in the sensors for longer also reduces energy consumption significantly. This is because data transmission consumes a lot more power than logging data to flash or aggregating it in the sensors and sending it much less often. This capability is crucial in many applications where a wireless sensor network is expected to work autonomously on batteries.

TinyStreams is also a language and operation configuration abstraction for sensor networks. In the rest of this section, we review a classification for sensor network programming abstractions and examples of such approaches.

According to [18], there are two main sub-classes concerning sensor network programming abstractions: one sub-class focuses on providing the programmer with abstractions that simplify the task of specifying the node local behavior of a distributed computation. Consequently, the overall system behavior must be described in terms of pair-wise interactions among nodes within radio range; differently, the second sub-class is characterized by higher-level abstractions used to program the system as a whole (macro-programming), regardless of the single devices. Likewise, in [17] low-level programming abstractions are programming languages that require the pro-

grammer to code individual nodes and to specify inter-node communications in detail, while high-level programming abstracts away those details and provides ways to specify the behavior globally. According to [17], some of the relevant characteristics concerning language are: communication and computation perspectives, programming idiom and distribution model.

The communication perspective distinguishes languages that directly offer constructs for physical neighborhoods (e.g. NesC [14] and ATaG [19]), those that allow targeting subsets of nodes depending on application-level information (e.g. Regiment [20] and Pieces [21]) and those offering a global view programming of the system (e.g. TinyDB [4]). The main advantage that some global view programming styles such as TinyDB offer is simplicity, while the drawbacks are related to the lack of flexibility and reach, since the user does not control details. TinyStreams allows users to work at any of those levels, since it is possible to manage and route between neighbors, over groups or over the whole system using individual commands.

The computation perspective distinguishes imperative approaches, which are programming solutions based on sequential or event-driven semantics (e.g. Abstract Regions [22] and Pleaides [23], or platform code such as NesC), and declarative solutions, which are usually very concise in describing the system behavior using, for instance, database-style or rule-oriented semantics (e.g. TinyDB [4] and Cougar [1]). Functional paradigms express application processing by applying one or more functions to data sensed in some part of the system (e.g. Regiment [20] and snBench [24]). Flask [16] is also a functional, domain specific language embedded in Haskell, offering high-level reusable abstractions to the sensor network, and FlaskDB is a macro-programming language over it. Flask uses an intermediate distributed dataflow graph model that is compiled into node-level binaries. If more than a single idiom is associated to address different aspects, those are hybrid approaches, such as the one presented in ATaG [19]. Declarative SQL-like approaches are very intuitive and easy to use and to learn, not so with rule-based systems or functional paradigms. The imperative approaches range from low-level platform languages, which require whole specifications, to more abstract solutions, which are not particularly intuitive or easy to use. TinyStreams is declarative and streamSQL-like, similar to the idioms of stream engines such as Streambase [12].

Distribution models can be classified as database-oriented, where SQL-like queries are used as in a relational database (e.g. TinyDB [4]), data sharing-oriented, where nodes can read or write data in the shared memory space (e.g. Kairos [18] and Abstract Regions [22]); or as message passing, based on exchanging messages between nodes (e.g. NesC [14], DSWare [25] or Contiki [15]). Message passing paradigms are typically much more flexible, since they allow the programmer to specify exactly what is exchanged and how. The advantage of other alternatives such as database-style or data-sharing is to allow the user to specify complex patterns of processing and sharing data with only a few, system-wide commands, hiding the precise details of communications into their code generation logic. Clearly, a good compromise solution would be one that would allow the specification of data exchanges at the level desired by the programmer, which varies with application context. Stream to stream

routing, where streams may reside in any node or group of nodes, makes TinyStreams such an approach.

3 TinyStreams

TinyStreams implements a stream model over embedded device nodes and over the heterogeneous system that includes those nodes and other computing devices. Concerning the individual embedded devices, TinyStreams implements a local stream management engine for querying, creating and dropping streams, inserting, deleting and updating tuples, both in memory and on flash. Concerning the distributed system, TinyStreams allows definition of references and stream routing primitives to route data from producers to consumers along the distributed system. **Fig. 1** shows the main TinyStreams modules on embedded device nodes. Embedded operating systems such as TinyOS [14] or Contiki [15] offer API primitives concerning memory, file system, radio, sense and actuation. TinyStreams queries are submitted in an sql-like syntax in a client console, parsed locally and forwarded to target nodes pre-parsed in a compact fixed-offsets bytearray. Embedded nodes have a TinyStreams engine on top of the operating system. The engine receives command byte arrays and stream data messages (through the communication module shown in **Fig. 1**), parses them and processes the commands and data using data access, sense and act primitives. The 'Stream and Query Processor' module implements tuple-at-a-time processing that requires very little memory and manages streams in memory or flash depending on their size. Memory operation is faster but only serves small streams, while file-based operation is able to handle larger streams. The query processor handles simple select expressions, conditions, alarms, sampling windows and other typical SQL-like primitives.

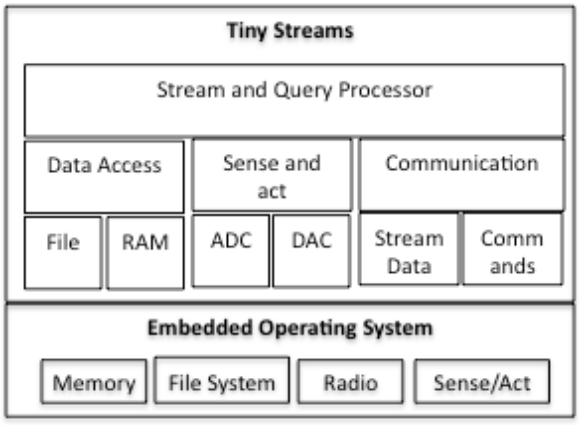

Fig. 1. Node Engine Modules

3.1 Model and Commands

Streams are time-ordered sets of tuples, and a tuple is a sequence of attribute values corresponding to attributes with an attribute data type and domain. Stream metadata is a structure defining the set of attributes and corresponding types and domains. A stream may have a window, which limits the number of tuples that can be in the stream at any time. While flash-resident streams can be created with no window (similar to database tables), memory-resident streams must fit into the constrained memory size, therefore they have window size limitations.

Fig. 2 shows a windowed memory stream and a file-based flash-resident stream.

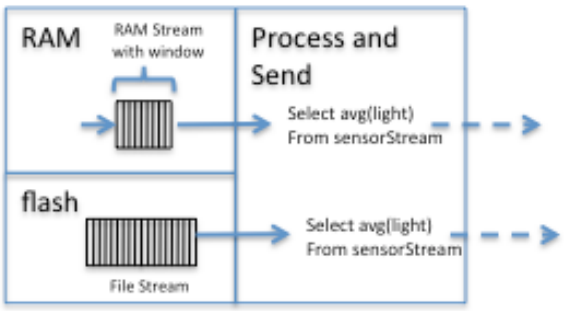

Fig. 2. Memory and Flash Streams

TinyStreams has data access primitives abstracting away from memory versus flash stream residency. Since embedded devices have sensors, there are also predefined special "sensor streams" that correspond to each sensor. If the sensor is named "sensorX", the corresponding stream will be called sensorX and will have a single attribute named "value" that corresponds to the sensor readings. A sampling clause allows users to command acquisition with a sampling rate. The syntax for a sensor stream collecting 24 hours' worth of per-minute sensor values is:

```
Create stream sensorXvalues
in DepositsSensorNodes
as select value
from sensorX
window 24 hours
sample every 1 minute;
```

Table 1 shows TinyStreams SQL-like constructs.

Stream data routing is based on consumer streams specifying that they consume data from some producer stream(s). For instance, the following consumer stream seating in a remote control station – SensorXData - gets its data from a producer stream SensorXvalues. The SensorXvalues producer stream can be a stream running in each of a set of sensor nodes.

Stream creation and dropping
 	
(streams with no window and tables are equivalent entities)
	`Create stream a`
	`(nodeID numeric,`
	`a numeric)`
	
	`Create table a`
	`(nodeID numeric,`
	`a numeric)`
	
	`Drop stream a;`
	`Drop table a;`

Stream creation from select, with window and sampling rate	`Create stream`
	`sensorXvalues`
	`in DepositsSensorNodes`
	`as select nodeID, value`
	`from sensorX`
	`window 24 hours`
	`sample every 1 minute;`

Select command	`Select nodeID avg(value)`
	`from sensorXvalues`
	`Group by nodeID`

Insert command	`Insert into a values(2);`
Delete command	`Delete from a where` `nodeID=1;`

Table 1. TinyStreams SQL Constructs

```
Create stream SensorXData
in controlstation as
Select *
From sensorXvalues;
```

If sensorXvalues was created with a 24 hour window, its values will be forwarded into SensorXData in the control station every 24 hours. If, instead of logging every value to the control station, one wants to get only a summary of the values, one way to do that would be to aggregate in the sensors and issue a query for the aggregated values (or registering a control station stream with periodic aggregation query).

Create stream sensorXvalues in DepositsSensorNodes as select nodeID, value from sensorX window 24 hours sample every 1 minute;	Create stream sensorXvaluesAgg in DepositsSensorNodes as select nodeID, avg(value) from sensorXvalues group by nodeID;	Create stream SensorXData in controlstation as Select * From sensorXvaluesAgg;

Fig. 4. Example Collecting Aggregated Stream

3.2 Stream Creation and Querying

Stream creation syntax allows a user to create a stream from a list of attributes or to create a stream as a select command with multiple optional clauses:

```
Create stream streamName
[in [nodeID| nodeSet]] as
Select [select expressions]
From [ sensorID | streamName ]
[Group by clause]
[sample clause]
[window clause]
[storage clause];
```

Commands are submitted through a console in a node with access to the distributed sensor network system. This console has an associated catalog that keeps node addresses information and node referencing identifiers, which are created to ease the task of specifying nodes and node groups in commands. The following example identifies node address suffixes and a set of two nodes as "DepositsSensorNodes".

```
SensorNode1 = "1333:8068";SensorNode2 = "137b:d539";
DepositsSensorNodes = {SensorNode1,SensorNode2};
```

Sensor node identifiers are specified using the "in" clause of stream creation commands. The "from" clause specifies input from streams. The stream with the name in the "from" clause (producer) will be sending its output into the stream that is being created (consumer). The producer and consumer streams may be in different nodes, commanding data forwarding from producer to consumer. Since the producer stream may be in more than one node, we can for instance command stream production in multiple nodes with a single command, by specifying a node set referencing in the "in" clause, and send all the data from those producers to a consumer node by specifying a stream that consumes from that multiple-node stream.

The "sample" clause is useful for sensor streams, indicating how often the sensors should be sampled.

The "window" clause indicates the size of the data that should be kept at any time, in either number of values or time period. This allows data holders to have a constant

size, since data enters and leaves in ordered fifo order, while maintaining up to window size of data. When a window fills-up, its data is sent to consumer streams and the window is emptied for another round. A time-based stream window is defined by specifying a time unit (e.g. "window 1 hour"), while a size-based window is specified with a number of tuples (e.g. "window 10 tuples").

A stream may be stored either in memory (if it is sufficiently small to fit there) or on flash disk. The storage clause allows users to specify where to store the data.

A metadata structure describes each stream. The structure contains the stream name, attribute names and domains (NUMERIC, LONG, STRING). The physical representation of tuples is through a compact byte-array record of the attribute values.

Creation of a consumer stream also creates a periodic query to fill the consumer from producer streams. Alternatively, a query can also be posed as a one-time query. Processing a query involves retrieving tuples one-by-one into memory, operating on the tuple, incrementally computing aggregations if specified, then either sending the result tuples through the communication interface to a consumer stream in the form of a stream data message, writing the result in stream storage, or printing the result in the console or in a serial port. The tuple-by-tuple processing saves a lot of memory.

Stream selection projects attributes and may aggregate values along tuples of the stream. The aggregation functions are COUNT, MAX, AVG, MIN, and SUM, each of which is updated for each processed tuple that satisfies the SELECT predicate. The result set of tuples will contain a single tuple for each group of the aggregated values. As an example, the following query retrieves the average and maximum temperature per month:

```
Create stream temperatureSummary
in BuildingNodes
as
SELECT AVG(value),
MAX(value),month(timestamp),year(timestamp)
FROM temperatureSensor
Group by month(timestamp),year(timestamp);
```

The query processor computes aggregations incrementally. For instance, a maximum is computed incrementally as the maximum between the current maximum and the value of the current tuple; likewise, a sum is the current sum plus the new value from the current tuple, and an average is the current sum divided by the current number of tuples. This is done for each aggregation group, groups being addressed as a hashmap with the key being the group attribute values.

Conditions are added through the where clause, selecting a subset of the tuples in a stream. While processing the current tuple, the query processor verifies whether the conditions evaluate to true and only considers the tuple for further processing if the condition is true.

The delete command removes all tuples matching the condition indicated in the command. The delete is implemented by scanning all tuples, selecting those that do not match the delete condition into a new stream that replaces the previous one.

Stream drop commands free the memory occupied by the data and the metadata structure.

Since embedded sensor devices frequently also actuate on some physical system through DAC interfaces, actuation conditions and syntax is added to the approach. **Fig. 5** shows an example closing shades if a temperature alarm goes on (temperature > 30) and opening them if it goes off (temperature<25). This is done in the sensor nodes themselves. The example also shows the use of variables and customized functionality (the closeOpenShades code, which is developed in the platform coding language).

```
shades=SensorNodes.Action(code="dev/closeOpenShades",
                     api={closeShades(),openShades()});
shades.openShades();
shadesOpen=true;

create Stream temperatureBasedShadesOpenClose
in SensorNodes as
  select NodeID, value
  From temperatureSensor
  Where temperature>30
  Action {

    If(shadesOpen==true)
      shades.closeShades();
      shadesOpen=false;
  }
  Where temperature<25
  Action {
    If(shadesOpen==false)
      shades.openShades();
      shadesOpen=true;
  }
```

Fig. 5. Specifying an Action

3.3 Query Processor

Fig. 6 shows the command processing path in the networked environment. Users submit a command through a console. A parser interprets the commands and accesses a catalog for addressing references and other details, producing a command bytecode. That bytecode is sent to the target nodes, which receive it through a communications interface, parse it and process against the local streams. The processing of a stream is done by a 'Process & Send' (P&S) functionality.

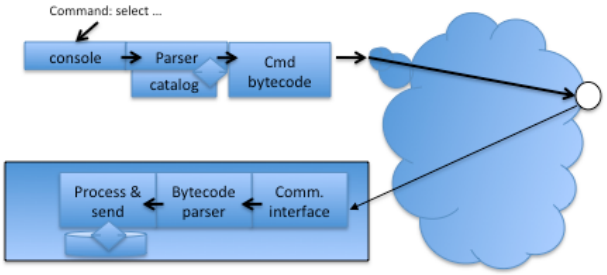

Fig. 6. Command Processing Path

All commands have a command type, and SQL-related bytecoded commands (create, drop, delete, update, insert, select) all have a stream identifier. The drop command only has those elements, but stream creation, selection, delete and update commands have a stream-like structure with fields for expressing query clauses.

The select clause fields may be attributes, constants or aggregation functions, with specific codes for identifying each individual alternative - operands, functions and operators.

The query processor in each node responds to the following timer and network events:

- If a stream creation command arrives through the communication interface, the stream metadata is created and the necessary memory space is created for the stream. If the stream is flash resident, a new file is created for the stream. If the stream has a timed window, a window-related timer is created and armed for periodically waking up and operating on the stream. If the stream is a sensor stream, with sampling rate, then a sensor acquisition-related timer is created and armed for periodically waking up and acquiring the signal through the sensor ADC;
- Stream window timer expiration –> the stream selection query stored in stream metadata is executed against the stored stream using the Process&Send functionality;
- One-time-query (Select command) arriving through the communication interface -> the stream selection query is executed against the stored stream data using the Process&Send functionality;
- Acquisition timer expiration -> the hardware ADC is sampled and a tuple is created with the corresponding values. The tuple is inserted into the stream window, either in memory or in file;
- Stream data arriving through the communication interface -> if stream data arrives in the communication interface, the consumer stream is identified in the message. The data is stored in the data area of that stream, either in memory or in flash.
- Window expiration -> if a stream has a window and a registered select query, and the window size is reached, then the select query stored in the stream metadata is retrieved and applied against the stored stream data using the Process&Send functionality;
- Other commands arriving through the communication interface -> the commands are executed.

Tuple-by-tuple operation is implemented in the "Process & Send" (P&S) functionality in the following manner: the stream data is scanned tuple-by-tuple (either from RAM or flash). For each tuple, the query processor first applies where clause conditions to determine if the tuple is to contribute to further computation and output. If the condition evaluates to true, then P&S looks at each select clause field and:

- If the field is a constant, it outputs the constant into a temporary tuple space;
- If the field is an attribute, it copies the attribute value of the current tuple into the temporary tuple space;
- After a tuple is processed, if the query is not specifying an aggregation, then the result tuple is returned immediately;
- If the field is an aggregation (e.g. sum, count, avg, max, min), the attribute values of the current tuple update a temporary aggregation computation structure for the select aggregation expressions. The aggregation computation structure maintains a set of additive aggregation computations for the select field expression. The additive aggregation computations are: count - n, linear sum -ls, square sum - ss, maximum - max and minimum - min. This structure allows immediate return of (sum, avg, max, min, count) expressions as soon as all the tuples have been processed. If there is a group by clause, then there is a hashmap with the group-by values as keys and an aggregation structure of the type described above for each hashmap entry. Each tuple now updates the aggregation computations for the corresponding aggregation structure.

If the query specifies an aggregation, after all the tuples have been processed, the query processor needs to take the aggregation structures and return the aggregation values that are needed by the query.

4 Evaluation

TinyStreams was implemented as an evolution of a system configuration interface developed for an industrial application in the context of European Project Ginseng - Performance Controlled Wireless Sensor Networks. It was implemented on top of the Contiki operating system using C programming language. It can be ported to other operating systems for resource-constrained devices, by adjusting the storage and communication layers to the new operating system API. The main operating system interface API primitives needed are send/receive and read/write. At the level of TinyStreams, the read and write primitives are abstracted, with implementations for both flash and RAM. Flash storage was implemented on the Coffee file system [9] in the prototype.

We measured the code size and query processing performance with simple queries over our experimental testbed.

The total code size of the TinyStreams node component, shown in **Fig. 7** (the TinyStreams code running on the embedded device) is less than 8KB, which fits nicely into the ROM available in most typical devices. We also compiled the base Contiki operating system code and then the same code with parts of the TinyStreams func-

tionality, in order to assess the code size of those parts (the difference between the total size and the size of the code with the operating system only). The results show that the simple tuple-by-tuple query processing algorithm with no complex optimizer code and with fixed in-memory stream structures results in a small code size. This is also shown as the number of lines of code.

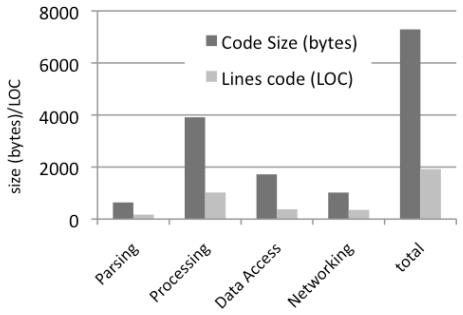

Fig. 7. Code Size for TinyStreams Parts

Fig. 8 shows the time taken for the query 'select value, nodeID from adc0', and the query 'select sum(value) from adc0' to execute, selecting either flash or RAM resident streams with varied sizes. The execution time increaed approaximately linearly with the number of tuples, and the computation of an aggregation expression increases execution time by only a small amount of time. This is as expected, since aggregation is done incrementally over an aggregation structure, with low extra overhead.

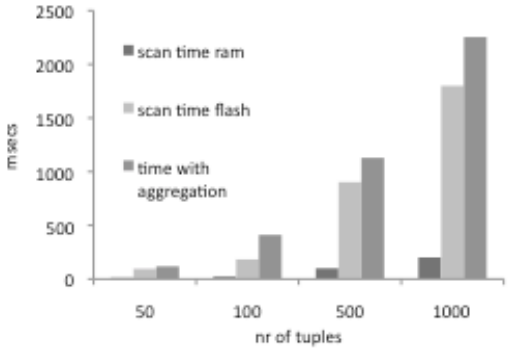

Fig. 8. Query Processing Times (msecs)

Table 2 shows the runtime memory requirements of TinyStreams, not counting the space occupied by each stream metadata and stream data.

	Static RAM (bytes)	Heap (bytes)
Query Parser	83	20
Storage	55	60
Communication	110	100
Query Processor	890	102
Total	1138	282

Table 2. Runtime Memory Requirements

5 Conclusion

We have proposed a TinyStreams model and engine for dealing with data in networked sensor systems with embedded sensing devices. We have shown how the approach allows data storage, retrieval, processing and routing. Memory and file system storage is abstracted into a stream management layer and a producer-consumer stream model allows networked configuration and processing with ease. The major advantage of the approach is that it allows users to specify what each node of a sensor network with computation-capable devices should do and how the data should be routed in a simple manner.

6 References

1. P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In Proceedings of the Second International Conference on Mobile Data Management, 2001.
2. H. Dai, M. N., and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys), Baltimore, MD, USA, Nov. 2004.
3. Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Rethinking data management for storage-centric sensor networks. In Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, Jan. 2007.
4. S.Madden, M.Franklin, J.Hellerstein, andW.Hong. TinyDB: an acquisitional query processing system for sensor networks. ACM Trans- actions on Database Systems, 30(1):122–173, 2005.
5. G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In Proceedings of the International Conference on Em- bedded Networked Sensor Systems (ACM SenSys), Boulder, Colorado, USA, Nov. 2006.
6. S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In Proceedings of the International Conference on In- formation Processing in Sensor Networks (ACM/IEEE IPSN), Cam- bridge, MA, USA, Apr. 2007.
7. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: Design and implementation of interoperable and evolvable sensor networks. In Proceedings of the International Conference on Em- bedded Networked Sensor Systems (ACM SenSys), Raleigh, NC, USA, 2008.
8. P.Pucheral, L.Bouganim, P.Valduriez, and C.Bobineau. PicoDBMS: Scaling down database techniques for the smartcard. The VLDB Jour- nal, 10(2-3):120–132, 2001.

9. N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In Proceedings of the International Conference on Information Processing in Sensor Networks (ACM/IEEE IPSN), San Francisco, CA, USA, Apr. 2009.
10. Streambase URL, 2012:www.streambase.com
11. Esper URL, 2012:esper.codehaus.org
12. Karl Aberer , Manfred Hauswirth , Ali Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks", in Mobile Data Management (MDM), Germany, 2007.
13. 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).
14. D. Gay, P. Levis, R.V. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, San Diego, California, USA: ACM, 2003, pp. 1-11.
15. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, IEEE Computer Society, 2004, pp. 455-462.
16. G. Mainland, M. Welsh, and G. Morrisett, Flask: A Language for Data-driven Sensor Network Programs, Harvard University, Tech. Rep. TR-13-06, 2006.
17. Mottola L., ""Programming Wireless Sensor Networks: From Physical to Logical Neighborhoods".PhD Thesis," Politecnico di Milano (Italy)., 2008.
18. R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming Wireless Sensor Networks Using Kairos," Distributed Computing in Sensor Systems, 2005, pp. 140, 126.
19. Bakshi, V.K. Prasanna, J. Reich, and D. Larner, "The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems," Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services, Seattle, Washington: USENIX Association, 2005, pp. 19-24.
20. R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," Proceedings of the 6th international conference on Information processing in sensor networks, Cambridge, Massachusetts, USA: ACM, 2007, pp. 489-498.
21. J. Liu, M. Chu, J. Reich, and F. Zhao, "State-centric programming for sensor-actuator network systems," Pervasive Computing, IEEE, vol. 2, 2003, pp. 50-62.
22. M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1, San Francisco, California: USENIX Association, 2004, pp. 3-3.
23. N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," SIGPLAN Not., vol. 42, 2007, pp. 200-210.
24. M.J. Ocean, A. Bestavros, and A.J. Kfoury, "snBench," Proceedings of the 2nd international conference on Virtual execution environments - VEE '06, Ottawa, Ontario, Canada: 2006, p. 89.
25. S. Li, S. Son, and J. Stankovic, "Event Detection Services Using Data Service Middleware in Distributed Sensor Networks," Telecommunication Systems, vol. 26, 2004, pp. 368, 351.