SwissQM: Next Generation Data Processing in Sensor Networks [*]

System Overview

Rene Mueller Gustavo Alonso Donald Kossmann

Department of Computer Science
ETH Zurich
Zurich, Switzerland
{muellren, alonso, kossmann}@inf.ethz.ch

ABSTRACT

Sensor networks are becoming an important part of the IT landscape. Existing systems, however, are limited in two fundamental ways: lack of data independence, and poor integration with the higher layers of the data processing chain. In this paper we present SwissQM, a next generation architecture for sensor networks built to address these two limitations. SwissQM combines a virtual machine at the sensors with a powerful gateway as access point to the system. The flexibility of SwissQM opens up the doors to sensor networks with richer functionality, data model independence, optimised performance, and smooth integration into the rest of the architecture. SwissQM supports adaptability (e.g., push down strategies), multiple users and applications, high query turn-around, extensibility (e.g., user defined functionality), and optimised use of resources. The paper describes SwissQM, the features it provides, and examples of its use.

1. INTRODUCTION

Thanks to Moore's law, the IT industry has seen computers getting smaller, cheaper, and more powerful during the last three decades. Now and in the coming years, computers are increasingly being enhanced with powerful sensing devices. Computing devices extended with sensors are also being combined into *sensor networks* to tackle larger data acquisition tasks, e.g., monitor the behaviour of a large population of animals [14] or monitor the climate of a large territory [17]. Commercial applications of sensor networks include supply chain management [2], support of elderly people [16] and security facilities for manufacturing sites and homes [1].

Nevertheless, building and deploying sensor networks remain elusive and difficult tasks. Most existing deployments use application specific, *hard-coded* software. Typically, there is little support for data independence, high level abstractions, multiple users, and, above all, integration with the higher layers of the data processing chain. Systems are also very rigid on how sensors can be programmed and what the sensor network can do.

Several research projects are addressing some of these limitations. Most relevant to the work reported in this paper are three projects at UC Berkeley: TinyOS, TinyDB, and HiFi. TinyOS [6] is an operating system for small sensing devices based on event processing and a C derivative called NesC. It is designed to operate on sensing devices with limited storage, processing capabilities and battery life times. TinyDB [13] is built on top of TinyOS and provides a declarative query interface (SQL dialect) to the sensor network as well as some capabilities for in-network data processing and aggregation. Finally, the HiFi project [5] proposes a staging architecture to acquire, clean, and aggregate data acquired through applications such as TinyDB. This and work such as [7] indicate that higher-level abstractions and more powerful platforms at the sensor network level are needed.

The XTream project at ETH Zurich is a recently established research effort that aims at developing a comprehensive platform for supporting the entire data cycle in sensor networks: from the data acquisition to the data processing and storage, including deployment, optimisation, routing, and embedding within end user devices. This paper describes one of the key components of XTream: SwissQM (**S**calable **WI**rele**S** **S**ensor network **Q**uery **M**achine). In short, SwissQM is intended as the next generation architecture for data acquisition and data processing in sensor networks. Its main objectives are to provide richer and more flexible functionality at the sensor network level, a more powerful and adaptable interface to the outside world, data independence, query language independence, optimised performance in a wider range of settings than current systems, and smooth integration with the rest of the data processing architecture. SwissQM is based on a specialised virtual machine that runs optimised byte code rather than queries. As a result, SwissQM does not make any assumptions about the query language used (e.g., SQL or XQuery), about the deployment strategy of the underlying sensor network (e.g., one single network or multiple networks), and can easily provide highly efficient multi-user support. Compared to

[*]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.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3^{rd} *Biennial Conference on Innovative Data Systems Research (CIDR)*
January 7-10, 2007, Asilomar, California, USA.

existing systems, SwissQM provides a *generic* high-level, declarative programming model (it can support both SQL and XQuery) and imposes no data model (e.g., relational or XML). SwissQM is also Turing-complete and supports user defined functions, window queries, complex event generation at the sensor level, an extensible instruction set, sophisticated optimisations, sensor over-provisioning, and overlapping but distinct sensor networks. All these features make it possible to implement in SwissQM many and important optimisations that are either not possible or rather cumbersome to implement in existing systems.

2. SWISSQM

2.1 Design Considerations

SwissQM has been designed with several requirements in mind:
(1) Separation of sensors and external interface: SwissQM should not implement any particular query language. The programming model should be independent of the query language used. It must also be dynamically adaptable. As a result, the sensor nodes should not contain application specific functionality (e.g., the ability to parse SQL or join operators). Such functionality is treated as a dynamically deployable extension.
(2) Dynamic, multi-user, multi-programming environment: SwissQM should not impose restrictions on the query submission and change rate, nor in the number of queries that can be run concurrently (beyond the inherent limitations of the underlying hardware).
(3) Optimised use of the sensors: The only processing at the sensor nodes should be that related to capturing, aggregating, and forwarding data. Anything else should be there only because it has been pushed down from above. This increases the memory available for data and leaves room for more queries and/or more sophisticated processing such as event generation or user defined functions.
(4) Extensibility: SwissQM should be programmable to include the ability to implement user defined functions and the ability to push down functionality from higher data processing layers. Extensibility also refers to SwissQM itself: It should be possible to dynamically extend and modify the behaviour of SwissQM as needed.

2.2 A SwissQM Sensor Network

A SwissQM sensor network is built using a *gateway node* and one or more *sensor nodes.* The gateway node is assumed to have sufficient computing power and no energy or memory restrictions. The gateway acts as the interface to the system. The sensor nodes are assumed to be resource constrained devices, running on batteries. The sensors perform the actual data acquisition. In this paper we concentrate on the *Query Machine* (QM), the virtual machine that runs on the sensor nodes, and we will only explain the functionality of the gateway when needed (see [15] for an example of the type of processing occurring at the gateway).
The sensor nodes are organised into a tree that routes data towards the root, where the gateway node is located (Fig. 1). This is the same strategy as used in, e.g., TinyDB, since a tree facilitates in-network aggregation and reduces the amount of routing data a node has to keep. Details about creation and maintenance of the routing tree are beyond the scope of this paper. Here we assume that every node has a link to a parent closer to the gateway. We also assume there is a single tree although SwissQM is being built to support multiple independent trees and individual node addressing.

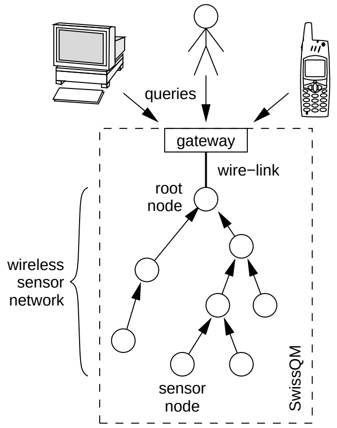

Figure 1: Sensors nodes are organised in a tree topology with the root node connected to the gateway

2.3 The Sensor Network Gateway

Requirements 1 and 2 have led to a more radical separation between the functionality of the sensor nodes and the gateway that is typically encountered in sensor networks. Since all sensor networks require such a gateway, SwissQM has been designed to exploit the gateway as much as possible. Unless dictated by optimisation strategies (e.g., minimisation of data transfer), everything that can be done at the gateway is done there rather than at the sensor nodes. Thus, the gateway provides all external interfaces, as well as query optimisation and compilation facilities. At the sensor nodes one finds only the code that is strictly needed to capture, aggregate, and propagate the data. Any additional code, e.g., user defined functions, is located at the sensor nodes only if explicitly pushed down by the gateway. The resulting architecture has considerable advantages. SwissQM can support a wide variety of interfaces (as dictated by requirement 1, e.g., SQL, XQuery, Web services) and these interfaces can change over time without requiring changes to the code in the sensor nodes. Sophisticated optimisation strategies can be implemented at the gateway without affecting the performance of the sensor nodes. The gateway is also the natural place to implement data cleaning pipelines and virtualisation such as those described in [7].

充分利用gateway，尽量将操作和优化位于gateway，支持多种上层接口。
上层接口的变化不会影响底层sensor node的代码

sensor node只负责感知通信等基本功能以及gateway分发的特定功能代码

The gateway works as follows: The gateway processes *user queries* and replies with data streams. The user queries can be expressed in various languages. The gateway processes and combines the user queries into a smaller, more optimised subset of *virtual queries.* The virtual queries are expressed in an internal format suitable for multi-query optimisation, query merging, subexpression matching, window-processing optimisation, etc. The optimised virtual queries are then transformed into *network queries* expressed in the byte code understood by the sensor nodes. This three-tier mechanism virtualises the sensor network and permits multiquery optimisation (requirement 2) across user queries for a more efficient use of the sensor network [15]. Thanks to the virtual

用户请求-->
内部虚拟请求格
式-->
传感器可理解的
字节码网络请求

三层的传感器网络虚拟请求机制进行多查询优化，使得用户可以更有效的利用传感器网络
虚拟请求可以使不同interface提交的请求也可以做到多请求优化

用于保持聚合查询执行的状态

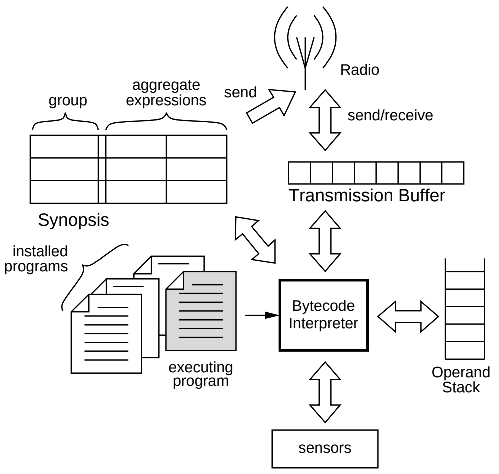

Figure 2: Query Machine Components

query step, it is also possible to use multi-query optimisation on queries submitted through different interfaces, e.g., through SQL and XQuery.

2.4 The Query Machine

运行在sensor node上的VM

指令集：java vm的指令子集以及一些专门指令

(减小程序的代码量)

QM只支持16bits整型以及布尔型，不支持浮点型，可以在网关上进行相应的类型转换，尽量优化sensor node端的性能

Requirements 3 and 4 motivated us to implement the sensor nodes as a stack-based virtual machine, the *Query Machine* (QM) (the name emphasises the dual role of query engine and virtual machine). The instruction set in the QM is a small subset of the Java Virtual Machine [11] extended with specialised instructions to reduce the size of the programs. The QM uses a uniform type to simplify the implementation and reduce the footprint of the QM. All data types are represented as 16-bit signed integer types (Booleans are represented using C-style semantics). Floating point types are not supported to keep the size of the instruction set as small as possible (the necessary conversion can be easily done at the gateway and that way all operations at the QM are on integers—see requirement 3). The QM includes the following components shown in Fig. 2: An *Operand Stack* that stores 16-bit elements used as operands and results of instructions. A *Transmission Buffer* that temporarily holds data that are to be sent or were received but not yet processed. It is accessed through an index over the 16-bit data elements. A *Synopsis* data structure is used to keep state for the execution of aggregation queries. A set of *Sensors*. A set of *QM Programs* expressed in QM byte code (produced at the gateway from the user submitted queries). The current version of the QM is 7.5 thousand lines of NesC on top of TinyOS 1.1. It runs on Berkeley Mica2 motes (Atmel ATmega 128L micro-controller, 128 kB Flash memory for program storage and 4 kB of SRAM for dynamic program data such as stack and heap). The core of the QM (byte code interpreter and associated logic) takes 23 kB of Flash memory.

2.5 QM Bytecode

Table 1 shows the basic set of instructions of the QM. The length of a QM instruction is 1, 2 or 3 bytes, depending whether it has zero, a one-byte or a two-byte immediate field. The top group in Table 1 includes stack-manipulation operations (e.g., dup, swap), arithmetic (e.g., iadd) and control instructions (e.g., if_icmple). The complete instruction set of SwissQM is shown in Table 2 in the appendix. The mnemonics and the semantics are identical to their counterparts defined in the Java Virtual Machine Specification [11]. The remaining four groups are query specific extensions. The load/store group includes instructions for accessing the transmission buffer and the synopsis. The sensing instructions access the physical sensors or system parameters and push the obtained value onto the stack. Two transmission instructions are provided: send_tb sends the content of the transmission buffer whereas send_sy sends the current aggregation state stored in the synopsis. The aggregation instruction merge is described in Section 2.7. Following requirement 4, the instruction set is modular. As need dictates (or device capabilities evolve), additional instruction classes can be added or removed (e.g., more sophisticated aggregation or floating point support). Additionally, when SwissQM is ported to a different platform the instructions of sensing instruction class have to be adapted in order to reflect the physical sensors (humidity, barometric pressure, acceleration, etc.) that are available on that particular platform.

Table 1: Instruction set of the Query Machine

	# instructions
stack operations	16
arithmetic operations	11
control instructions	13
load/store instructions	9
sensing instructions	7
transmission instructions	2
aggregation instructions	1
Total	**59**

As mentioned, the core of the QM requires 23 kB of Flash memory. The 59 instructions require another 10 kB. The full QM with the complete instruction set uses 33 kB Flash and 3 kB SRAM (including space for dynamic program data). For comparison, TinyDB uses 65 kB of Flash and 3 kB of SRAM memory. As in all sensor networks [13], the program execution is dominated by the time used to send data. A send_tb instruction requires 25 ms for a single result message (22 bytes) if no contention occurs at the network MAC layer. The execution of a sampling instruction requires about 400 μs. The execution of all other QM instructions takes approximately 50 μs or about 400 CPU cycles.

2.6 QM Programs

For the purposes of this paper we consider a network query to be equivalent to a QM program. The translation process from a virtual query to a QM program is through *templating*, a well-known compiler technique. During the translation phase the query is optimised. The optimiser precomputes constant parts of expressions and also identifies common subexpressions in the selection-clause list, the group-by list and in the selection-predicate. By identifying common subexpressions and reordering the computation of the expressions the final program size can be reduced considerably for complex queries.

QM programs consist of three code sections, each executed at different times. The *init* section is executed once upon program start. It is used to initialise state in the synopsis before the program starts producing the first data tuple. The *delivery* section is executed once per sampling period. This is where the sensors are sampled, data gathered and merged in the local synopsis. The *reception* section is executed upon arrival of a message from a child. It extracts the data from the child's message and merges it with the local synopsis. As an example, consider the following XQuery expression where sensor nodes send their ID to the gateway when the temperature exceeds 60°C. The temperature sensor is sampled periodically every 10 seconds.

```
for $n in xt:sample($sensors, 10s)//node
where $n/temp gt 60
return $n/nodeid
```

In this query `xt:sample` is a user defined function in the XTream scope that takes a URI (the sensors) and sampling period as input and returns a sequence of elements. The XML schema defines the sensor network as a sequence of `<node/>` elements, of which each has a `<nodeid/>` and a `<temp/>` element. In XQuery `gt` denotes the greater-than relation. The delivery and reception sections of the corresponding QM program are as follows:

```
 1 .section delivery, "@10s"
 2      get_temp           # read temperature sensor
 3      ipushb     60
 4      if_icmple  end     # skip if temp≤60
 5      get_nodeid         # read node's ID
 6      istore     0       # store it at pos. 0
 7      send_tb            # send transmission buffer
 8 end:
 9
10 .section reception
11      send_tb            # forward tuple from child
```

The temperature sensor is sampled and the reading pushed onto the stack (`get_temp`, line 2). In line 3 (`ipushb`), the constant 60 is pushed onto the stack. The sensor reading and the constant are then compared (`if_icmple`, line 4). If the temperature is ≤ 60 the execution is resumed at label `end`, which marks the end of the section. Otherwise the node's identification number (deploy-time parameter) is read (`get_nodeid`, line 5) and stored at position zero in the transmission buffer (`istore 0`, line 6). The last instruction in the *delivery* section (`send_tb`, line 7) sends the transmission buffer, i.e., the ID, to the parent node. When a node receives a message from one of its children the message is placed in the transmission buffer and the *reception* section is executed. In this example, the data from the transmission buffer is immediately forwarded (`send_tb`, line 11). This is the default behaviour (if the *reception* section is not present, the data from the child is simply forwarded). In general, the *reception* section indicates how to merge the data from the child with the local synopsis. Taken together the two program sections in the example take 10 bytes of program memory.

2.7 In-network aggregation

Our approach to in-network aggregation follows the TAG idea of TinyDB [12]. We use a tree topology to forward the data and to aggregate along the path to the gateway. In general, aggregation is applied together with grouping. For instance, consider the following query:

```
SELECT building, floor, AVG(temp), MAX(light)
FROM sensors
GROUP BY building, floor SAMPLE PERIOD 10s
```

The aggregate expressions `AVG(temp)` and `MAX(light)` are computed for each `building-floor` pair. The query thus has two grouping expressions and two aggregate expressions. For `AVG` the aggregation state is a pair $\langle c, s\rangle$ consisting of the sum s and the number of summands c. For `MAX` the aggregation state is a single value. Aggregation is implemented with the `merge` instruction. The `merge` instruction is parameterisable and has the form

$$\texttt{merge}(n, m, \mathrm{aggop}_1, \ldots, \mathrm{aggop}_m) \quad .$$

The first parameter is the number of grouping expressions n. The second parameter m is the number of aggregation expressions that follow. aggop_i are constants that specify the aggregation operations. n, m, and the aggregation type constants are pushed on the stack before calling the `merge` operation. The following aggregation operations are currently supported: `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`, `VARIANCE`. The required stack layout for the `merge` instruction as well as the definition of the aggregation operations is shown in Table 2 in the appendix. For the example above, the code excerpt for merging is as follows:

```
ipushb   MAX
ipushb   AVG
ipushb   2
ipushb   2
merge
```

An example of a merging step for a complex query is given in Section 3.1 and Fig. 3. The synopsis is a table dynamically built for each program. It contains one row for each group, in the example it is one row for each `building-floor` pair. The layout of the local synopsis and the transmission buffer are implicit and established by the code in the *delivery* section when the data captured from the sensors is written to the transmission buffer[1]. Merging occurs both in the *delivery* and the *reception* section. In general, data from the children is merged as it arrives whereas the local data is added at the end of the sampling period. The `merge` instruction performs a nested-loop join of the table in the local synopsis and the table in the transmission buffer and overwrites the local synopsis. By default (as in the example) the synopsis is cleared automatically at the end of each sampling period. For temporal aggregation over several sampling periods (window queries) clearing the synopsis can be overridden. Explicit clearing of the synopsis is implemented with the `clear_sy` instruction.

2.8 Multiprogramming

We provide multi-query support on two layers: first, by merging different user queries into a virtual query (see 2.3) and second, by multi-programming in the QM. Multi-programming in the QM is done through sequential execution of the programs. The execution duration of a program is typically short compared to the sampling interval. Program execution including data capturing and merging is in the

[1] The data received from a child is written by the delivery section of that child and thus will match the format established by the *delivery* section of the parent.

order of microseconds. Data transmission is in the order of milliseconds. Sampling periods are in the order of seconds or larger. Thus, without any further optimisation, it is possible to run a number of programs even with the shortest sampling period of one second. For instance, in the example shown in Section 3.1 the execution of the *reception* section takes 27 ms on Mica2 sensor nodes, with a sampling period of 30 seconds. From a CPU point of view there is room for over 1000 such queries. Of course, there is a trade-off between the number of programs that can be executed and the memory available to each program, i.e., the size of the synopsis for storing aggregation state, the size of the stack and the transmission buffer. When combined with query-merging and multi-query optimisations it is possible to support a relatively large number of user queries (in [15] we run over hundred user queries).

2.9 Program Propagation

One of the unspoken limitations of ad-hoc sensor networks is the restricted message size. Large messages increase collisions and the probability of transmission errors on an already unreliable medium. Different radio platforms provide different message sizes. For example, the ZigBee MAC layer (IEEE 802.15.4) used on MicaZ and Telos nodes defines a maximum message size of 102 bytes. We use Mica2 nodes a with a proprietary ChipCon radio transceiver and a gross message size of 36 bytes. With the bytes used by the MAC, broadcast, and routing layers of TinyOS, the application message size is only 24 bytes. Larger message sizes are very convenient for data processing and query propagation. For instance, TinyDB uses TinyOS packets of 49 bytes instead of the smaller packets of 36 bytes. This 30% increase in message length is one of the reasons why such large data losses are observed in TinyDB deployments [7]. We intend to explore this issue in more detail in the future. In SwissQM we opted for the smaller 36 bytes message size. This is one of the reasons why we strive for compact byte code and the use of a complex instruction set (e.g., the `merge` operation). Programs that are too long are transmitted from the gateway in several messages, called *fragment messages*. We use a sophisticated protocol to guarantee reliable program distribution (beyond the scope of this paper and highly dependent on the type of deployment one wants to achieve; the default in SwissQM is to use TinyDB's *single network* paradigm where all the nodes do exactly the same; this is rather limiting but simplifies the dynamics of reliable program distribution and adding and removing nodes).

For comparison, TinyDB provides a query processing engine that is run on the sensor nodes. A query is distributed from the gateway into the network. TinyDB does not distribute the "raw" query string directly, instead, the query is split at the gateway according to the fields and expressions in the selection-clause, the where-predicate and the grouping list. For each sensor attribute accessed and each expression in the query one query message is sent. Everything else being equal, our byte code dissemination mechanism is more efficient than the query dissemination mechanism of TinyDB, especially for small queries.

3. EXAMPLES

3.1 Conventional Streaming Queries

The following query is specified in a streaming variant of SQL (as introduced in [13]) and is used to analyse the correlation between temperature and brightness (light) readings. Assume that the light sensor produces only unprocessed *raw* values from the A/D converter. Further assume that the light sensor has an offset reading that needs to be accounted for. The goal of the query is to average temperature readings of sensor nodes that have similar brightness readings. The groups are built by first removing the offset and then forming bins by applying *integer division*.

```
SELECT (light-512)/10, AVG(temp)
FROM sensors GROUP BY (light-512)/10
SAMPLE PERIOD 30s
```

This simple query illustrates very well key differences between SwissQM and TinyDB. The query contains expressions to be computed as part of the query evaluation. TinyDB does not encode complete expression trees. Instead, all expressions must match a fixed format of the form:

$$\begin{array}{ll} & \langle \mathit{attribute} \rangle \\ | & \langle \mathit{aggregate} \rangle \big(\langle \mathit{attribute} \rangle \, \langle \mathit{operation} \rangle \, \langle \mathit{constant} \rangle \big) \end{array}$$

Therefore the range of expressions supported by the current version of TinyDB is limited. In SwissQM, the query is translated into byte code. Using traditional compiler techniques expression trees are translated into a short sequence of byte code instructions. Moreover, the query can easily support arbitrarily complex expressions (within the inherent limits of memory available to QM programs) since we do not impose a hard-coded expression format. In order to compare with TinyDB, we simplified the grouping expression from $(\texttt{light} - 512)/10$ to `light`. TinyDB uses three query messages (120 bytes in total) to disseminate the query. In our case the corresponding QM program requires only 20 bytes (two fragment messages). The byte code for the original query is as follows:

```
.section delivery, "@30s"
   get_light
   ipushw      512
   isub
   ipushb      10
   idiv
   istore      0       # store group expression
   get_temp
   istore      1       # sum := temp
   iconst_1
   istore      2       # count := 1
   ipushb      5       # agg: AVG = 5
   iconst_1            # number of agg expr: 1
   iconst_1            # number of grp expr: 1
   merge
   send_sy

.section reception
   ipushb      5       # agg: AVG = 5
   iconst_1            # number of agg expr: 1
   iconst_1            # number of grp expr: 1
   merge
```

Fig. 3 shows the layout of the synopsis for this example. A group is identified by the value of the grouping expression. The aggregation state for `AVG` consists of a sum/count pair. Fig. 3 also shows the layout of the transmission buffer as

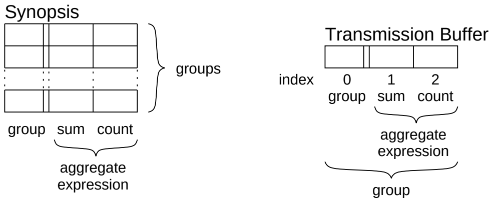

Figure 3: Layout of synopsis and transmission buffer (during delivery section) for the grouping aggregation query

is used in *delivery* section. Here the aggregate state contributed by the own sensors is prepared and merged to the local synopsis in `merge`. When the reception section is executed the transmission buffer contains the partial aggregation state received from the child. The size of the program including the complex grouping expression is only 27 bytes (two fragment messages).

3.2 In-Network Event Generation

Assume that a sensor network deployed in a building is used to detect potential fire related alarms. It is reasonable to assume that the presence of a fire is correlated with a sudden increase in temperature. In the following example the IDs of the nodes whose temperature reading increased by more than 10% during the last 10 minutes are returned. The corresponding query is:

```
SELECT s.nodeid
WHERE s.temp>1.1*w.mintemp AND s.nodeid=w.nodeid
FROM sensors AS s,
   (SELECT nodeid, MIN(temp) AS mintemp
    FROM sensors [RANGE 10min]
    GROUP BY nodeid) AS w
SAMPLE PERIOD 2min
```

In the inner query, a window of the temperature reading is created for every node. The query returns the smallest temperature values observed during the last 10 minutes for every node. These values are then compared with the current temperature reading in the outer query. First, observe that the window is maintained locally for each node, i.e., no aggregation state needs to be exchanged. Second, in contrast to a centralised approach where the event detection is done at the gateway, a message needs to be sent to the gateway only if the predicate in the where-clause is satisfied.

The QM program implementing this query uses a window that contains the last five temperature samples. This window is advanced every two minutes, resulting in a total window width of 10 minutes. The window entries are stored in a five element ring buffer. The buffer is implemented as an array and an index `next_insert` that is used to determine where to insert the next element. Fig. 4 shows the layout of the synopsis where the ring buffer is stored. The array occupies positions 0–4, the `next_insert` index position 5. The bytecode listing of the corresponding QM program is shown below.

The synopsis is initialised in the *init* section. Initially, the ring buffer elements are set to 32767. This is the largest positive value that can be represented using a 16 bit signed integer type.

```
# initialise synopsis
# set syn[0..4]:=MAX (0x7fff)
.section init
    ipushb    4       # i := 4
l1: dup
    iflt      l2      # exit if i<0
    dup
    ipushw    0x7fff  # push MAX
    swap
    istore_sy         # syn[i] := MAX
    idec              # i := i-1
    goto      l1
l2: pop
    iconst_0          # next_insert := 0
    istore_sy 5

.section delivery, "@2min","manualclear"
# find min(syn[i], i=0..N-1)
    iload_sy  4       # establish invariant
    ipushb    4       # setup variant

 # loop variant i on top of stack, invariant below
 # stack content: min(syn[j],j=i..4), i
l3: dup
    iflt      l5      # exit loop if i<0
    dup_x1            # → i,x,i
    iload_sy          # → i,x,syn[i]
    dup2              # → i,x,syn[i],x,syn[i]
    if_icmplt l4      # → i,x,syn[i] jump if x<syn[i]
    swap              # → i, min(x,syn[i]),max(x,syn[i])
l4: pop               # → i,min(x,syn[i])
    swap              # → min(x,syn[i]),i
    idec              # i := i-1
    goto      l3
l5: pop
# stack content at the end of loop: min(syn[i],i=0..4)

# insert new element into window
    get_temp          # read new sensorvalue
    dup               # → min_temp,temp,temp
    iload_sy  5       # get next_insert
    istore_sy         # syn[next_insert] := temp

# advance next_insert
    iload_sy  5       # get next_insert
    iinc              # next_insert := next_insert+1
    ipushb    5
    irem
    istore_sy 5       # next_insert := next_insert%5

# current stack content: .., min,temp
    swap              # →temp,min
    dup               # → temp,min,min
    ipushb      10
    idiv              # → temp,min,min/10
    iadd              # → temp,min+min/10
    if_icmple   l6    # skip if temp≤min+min/10
    get_nodeid        # read nodeid
    istore      0
    send_tb           # send nodeid
l6:
```

The *reception* section is invoked every two minutes, i.e., when the window needs to be advanced. Then the minimum

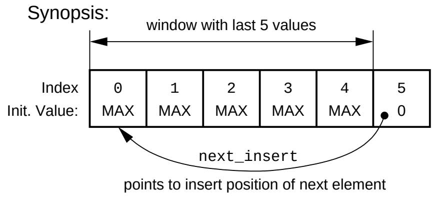
Synopsis:
window with last 5 values
Index
Init. Value:
0
1
2
3
4
5
MAX
MAX
MAX
MAX
MAX
0
next_insert
points to insert position of next element

Figure 4: Layout of synopsis for example

value of the ring buffer elements is determined (lines 18–35). Then the temperature sensor is sampled and the value obtained stored at the next insert position of the synopsis (lines 39–42). Afterwards, the next insert position is advanced (lines 45–49). Lines 52–57 evaluate the predicate $temp > 1.1 \cdot mintemp$. If the predicate evaluates to true, an event is detected and the node sends its ID to the root.

The size of the byte code is 62 bytes which can be disseminated in 3 fragment messages. The execution of the *delivery* section takes 40 ms on our Mica2 implementation. The *reception* section is empty as data does not need to be aggregated between nodes.

3.3 Adaptive Sampling

In principle, it is sufficient if sensor nodes only send data when the observed physical phenomena change. This allows answering queries directly at the gateway rather than fetching every tuple from the network. A similar idea has already been proposed in [3] where a statistical model is run at the gateway. Whenever the prediction of the model does not reach the confidentiality specified in the query, the gateway actively requests additional data from the sensors in order to update the model parameters and thus increase the quality of the prediction. In [3] the model at the gateway decides when to acquire additional data. As an extension of this idea, one can consider running a replica—or at least a simpler, less complex model—at the sensor nodes that allows the node to decide on its own when the model at the gateway is outdated and requires new data to update its parameters. By moving from a pull-based to a push-based mode of operation, the number of messages can further be reduced since no explicit requests have to be sent.

We illustrate how a trivial strategy for *adaptive sampling* can be implemented in SwissQM. The idea is very simple: increase the sampling rate if the phenomenon changes rapidly, otherwise decrease the sampling rate. The following pseudo code and corresponding bytecode show how to implement adaptive sampling.

```
int period = 0;
int count  = 0;
int oldval = getlight();

executeEvery2min() {
  int newval;
  if (count == 0) {
     newval = getlight();
     if (abs(newval-oldval)>oldval/10) {
        // change > threshold → decrease period
        if (period > 0) {
           // don't change period if already fastest
           period = period - 1;
        }
     } else {
        period = period + 1; // increase period
     }
     oldval = newval;
     send(getnodeid(),newval);
     count = period;
  } else {  // no sampling required
     count = count - 1;
  }
}
```

```
# initialise synopsis
.section init
    iconst_0
    istore_sy 0  # period := 0
    iconst_0
    istore_sy 1  # count := 0
    get_light
    istore_sy 2  # oldval := getlight();

.section delivery, "@2min", "manualclear"
    iload_sy  1  # load count
    ifneq     l1 # jump if count ≠ 0
    get_light    # → light
    dup          # → light,light
    iload_sy  2  # → light,light,oldval
    isub         # → light,light-oldval
    dup          # → light,light-oldval,light-oldval
    ifge      l3 # skip ineg if ≥ 0
    ineg         # → light,abs(light-oldval)

l3: iload_sy  2  # → light,abs(light-oldval),oldval
    ipushb    10 # → light,abs(light-oldval),oldval,10
    idiv         # → light,abs(light-oldval),oldval/10
    if_icmple l4 # within limit then increase period
    iload_sy  0  # → light,period
    dup          # → light,period,period
    ifeq      l5 # skip if already fastest
    idec         # decrement period
    dup          # → light,period-1,period-1
    istore_sy 0  # period := period-1
    goto      l5
l4: iload_sy  0  # increment period
    iinc
    dup          # → light,period,period
    istore_sy 0  # period := period+1

    # stack content: light,period
l5: istore_sy 1  # count := period
    dup          # → light,light
    istore_sy 2  # oldval := light

    istore    1  # send nodeid and light
    get_nodeid
    istore    0
    send_tb
    goto      l2 # go to end of section

l1: iload_sy  1  # sampling skipped
    idec         # count := count-1
    istore_sy 1
l2:
```

As the pseudo code shows, the program applies adaptive sampling on the light sensor. The routine `executeEvery2min` (line 5) is scheduled every two minutes, which determines the shortest possible sampling period. For larger sampling periods a variable `count` is introduced. When the routine is scheduled the variable is decremented (line 22). Sampling is only done when `count` reaches zero (line 8). Thus, all sampling periods are multiples of 2 minutes. The initial value assigned to `count` is the current sampling period stored in `period`. This period is increased or decreased depending on the difference between two consecutive samples `newval` and `oldval`. When the difference is larger than ±10% the `period` will be decremented (line 9 in the pseudo code). Otherwise the period is increased. After each sample the node sends its ID an the sensor value to the root (line 19).

The length of the resulting QM program is 64 bytes. It can be sent using three program messages. In the bytecode, the state kept by the program is stored in the three global variables, which are placed in the synopsis: `period` at position 0, `count` at position 1, and `oldval` at position 2. The initialisation is done in the *init* section.

4. CONCLUSIONS AND OUTLOOK

In this paper we propose SwissQM as the next generation architecture for data acquisition in sensor networks. In spite of operating in small and resource constrained devices, SwissQM opens the doors to richer functionality, data model independence, optimised performance, and smooth integration into the rest of the architecture. The two key design decisions in SwissQM are the separation between gateway and sensor nodes, and the implementation of a virtual machine at the sensor nodes, rather than a query processor. This gives us Turing completeness, independence at the sensor side of the query language used, independence of the user data model, and the necessary extensibility for pushing user defined functions and complex aggregation operations all the way down to the sensors.

4.1 Related Work

SwissQM borrows many ideas from several existing systems, mainly TinyDB [13]. The relation between TinyDB and SwissQM has already been commented upon throughout the paper. Beyond TinyDB, the most related work is that done on virtual machines for small devices.

In [4] the authors note that the use of a single, general purpose virtual machine (and execution model) cannot suit every problem domain. They suggest a *Virtual Virtual Machine* (VVM), a universal virtual machine, that can run different bytecodes for domain-specific virtual machines. Virtual machines for sensor networks have been proposed later in [9]. The difference to this previous work is that the SwissQM is a virtual machine specialised in data processing rather than a generic platform for application development. This makes SwissQM less general but better optimised.

Similarly, *Giotto* [8] is a runtime environment for embedded control systems. It is composed of two virtual machines. The first one, called *Embedded Machine* (E machine), processes external events. The second, called *Scheduling Machine* (S Machine), interprets code for which a temporal execution order is specified. Thus, the execution of SwissQM's delivery section relates to code execution done by the S machine whereas the code of the reception section would map to the E machine. However, and unlike Giotto, we are not aiming at hard-realtime systems. Hence we allow arbitrary code in the delivery section, which gives us more expressiveness.

Compiling queries into bytecode for virtual machines was also proposed in [10]. However, in the version available for download only a subset of the features described in the paper seems to be implemented. In particular, it is not clear how in-network aggregation is implemented. Furthermore, the system implemented aims at emulating TinyDB. SwissQM tries to expand beyond what has been achieved with TinyDB.

4.2 Future Work

SwissQM is a key element of the XTream project at ETH Zurich. With the features of SwissQM presented in this paper, we are now in an excellent position to implement in a realistic setting various optimisation techniques such as the maximisation of the life time of the sensor network if the network is over-provisioned. We are also exploring the impact of using XML at the gateway interface and the appropriate language constructs needed to make XQuery capable of dealing with sensor data streams and complex event generation.

Ultimately, SwissQM will be at the heart of the *RAISE* (Redundant Arrays of Independent SEnsors) scenarios targeted by XTream. The motivation for RAISE is quite simple: On the one hand, the prices for sensors are dropping rapidly; on the other hand, it is still very difficult to provision a sensor network correctly. The idea of RAISE is to simply over-provision the sensor network. That is, rather than calculating exactly which sensors, batteries, and computing power are needed at which place, RAISE suggests to do a rough estimate and then take, say, twice or three times as many sensors. This approach imposes risks and opportunities at the same time. SwissQM offers the flexibility required to program the sensors such that the risks can be avoided and the opportunities exploited. For instance, redundant sensors that serve the same purpose (e.g., provide the same sensor readings at the same or comparable locations) can coordinate in order to maximise their battery life-time; if the sampling period is 10 seconds, for instance, two redundant sensors can take their turns and each sensor can stay asleep for 20 seconds. Another example involves using adjacent sensor to sample different environment variables rather than using all sensors to sample everything (as it is currently done). In both examples, the critical success factor supported by SwissQM is that each sensor that runs SwissQM can be addressed and programmed individually. The exact protocols for RAISE are one important avenue for future work. It should be clear, however, that a flexible and powerful programming model as provided by SwissQM is necessary in order to implement these optimisations.

5. REFERENCES

[1] Alarm, wireless security systems. `http://www.alarm.com`.

[2] I. Bose and R. Pal. Auto-ID: managing anything, anywhere, anytime in the supply chain. *Commun. ACM*, 48(8):100–106, 2005.

[3] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *VLDB Journal*, 14(4):417–443, 2005.

[4] B. Folliot, I. Piumarta, and F. Riccardi. A dynamically configurable, multi-language execution platform. In *Proceedings of 8th ACM SIGOPS European Workshop*, pages 175–181, 1998.

[5] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, pages 290–304, 2005.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of ASPLOS-IX 2000*, pages 93–104, 2000.

[7] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *Proceedings of PERVASIVE 2006*, pages 83–100, 2006.

[8] C. M. Kirsch, M. A. A. Sanvido, and T. A. Henzinger. A programmable microkernel for real-time systems. In *Proceedings of VEE 2005*, pages 35–45, 2005.

[9] P. Levis and D. E. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS 2002*, pages 85–95, 2002.

[10] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of NSDI 2005: 2nd Symposium on Networked Systems Design & Implementation*, pages 343–356, 2005.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition, 1998.

[12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[13] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[14] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of WSNA 2002*, pages 88–97, 2002.

[15] R. Müller and G. Alonso. Efficient sharing of sensor networks. In *Proceedings of MASS 2006*, 2006.

[16] A. Sixsmith and N. Johnson. A smart sensor to detect the falls of the elderly. *IEEE Pervasive Computing*, 3(2):42–47, 2004.

[17] C. Tschudin, D. V. Muhll, S. Gruber, and I. Talzi. Permasense project, University of Basel. `http://cn.cs.unibas.ch/projects/permasense`.

APPENDIX

A. SWISSQM INSTRUCTION SET

Table 2 lists all instructions that currently implemented for SwissQM. For each instruction the state of the operand stack before and after the invocation is specified on the left and the right of ⇒ respectively. α is used to designate the remainder of the stack that is left unchanged by the instruction. The encoding that is used to specify the aggregates for the `merge` instruction is also shown.

	Instruction	Description
Stack	dup	$\alpha, x \Rightarrow \alpha, x, x$
	dup_x1	$\alpha, y, x \Rightarrow \alpha, x, y, x$
	dup_x2	$\alpha, z, y, x \Rightarrow \alpha, x, z, y, x$
	dup2	$\alpha, y, x \Rightarrow \alpha, y, x, y, x$
	dup2_x1	$\alpha, z, y, x \Rightarrow \alpha, y, x, z, y, x$
	dup2_x2	$\alpha, w, z, y, x \Rightarrow \alpha, y, x, w, z, y, x$
	pop	$\alpha, x \Rightarrow \alpha$
	pop2	$\alpha, y, x \Rightarrow \alpha$
	swap	$\alpha, y, x \Rightarrow \alpha, x, y$
	iconst_0	$\alpha \Rightarrow \alpha, 0$
	iconst_1	$\alpha \Rightarrow \alpha, 1$
	iconst_2	$\alpha \Rightarrow \alpha, 2$
	iconst_4	$\alpha \Rightarrow \alpha, 4$
	iconst_m1	$\alpha \Rightarrow \alpha, -1$
	ipushb <int8>	$\alpha \Rightarrow \alpha, \mathrm{sign\_ext}(i)$
	ipushw <int16>	$\alpha \Rightarrow \alpha, i$
Arithmetic and Logic	iadd	$\alpha, y, x \Rightarrow \alpha, y + x$
	isub	$\alpha, y, x \Rightarrow \alpha, y - x$
	imul	$\alpha, y, x \Rightarrow \alpha, y * x$
	idiv	$\alpha, y, x \Rightarrow \alpha, \lfloor y/x \rfloor$
	irem	$\alpha, y, x \Rightarrow \alpha, y \bmod x$
	ineg	$\alpha, x \Rightarrow \alpha, -x$
	iinc	$\alpha, x \Rightarrow \alpha, x + 1$
	idec	$\alpha, x \Rightarrow \alpha, x - 1$
	iand	$\alpha, y, x \Rightarrow \alpha, y \& x$
	ior	$\alpha, y, x \Rightarrow \alpha, y \vert x$
	inot	$\alpha, x \Rightarrow \alpha, {}^{\sim}x$
Control	if_icmpeq <int8>	$\alpha, y, x \Rightarrow \alpha$, jump if $x = z$
	if_icmpneq <int8>	$\alpha, y, x \Rightarrow \alpha$, jump if $x \neq z$
	if_icmplt <int8>	$\alpha, y, x \Rightarrow \alpha$, jump if $x < z$
	if_icmple <int8>	$\alpha, y, x \Rightarrow \alpha$, jump if $x \leq z$
	if_icmpgt <int8>	$\alpha, y, x \Rightarrow \alpha$, jump if $x > z$
	if_icmpge <int8>	$\alpha, y, x \Rightarrow \alpha$, jump if $x \geq z$
	ifeq <int8>	$\alpha, x \Rightarrow \alpha$, jump if $x = 0$
	ifneq <int8>	$\alpha, x \Rightarrow \alpha$, jump if $x \neq 0$
	iflt <int8>	$\alpha, x \Rightarrow \alpha$, jump if $x < 0$
	ifle <int8>	$\alpha, x \Rightarrow \alpha$, jump if $x \leq 0$
	ifgt <int8>	$\alpha, x \Rightarrow \alpha$, jump if $x > 0$
	ifge <int8>	$\alpha, x \Rightarrow \alpha$, jump if $x \geq 0$
	goto <int8>	$\alpha \Rightarrow \alpha$, jump always
Buffers	iload <int8>	$\alpha \Rightarrow \alpha, \texttt{buf[i]}$
	iload	$\alpha, i \Rightarrow \alpha, \texttt{buf[i]}$
	istore <int8>	$\alpha, x \Rightarrow \alpha$ and $\texttt{buf[i]} = x$
	istore	$\alpha, x, i \Rightarrow \alpha$ and $\texttt{buf[i]} = x$
	iload_sy <int8>	$\alpha \Rightarrow \alpha, \texttt{syn[i]}$
	iload_sy	$\alpha, i \Rightarrow \alpha, \texttt{syn[i]}$
	istore_sy <int8>	$\alpha, x \Rightarrow \alpha$ and $\texttt{syn[i]} = x$
	istore_sy	$\alpha, x, i \Rightarrow \alpha$ and $\texttt{syn[i]} = x$
	clear_sy	clear synopsis
	send_tb	send transmission buffer
	send_sy	send synopsis
Sensors	get_nodeid	$\alpha \Rightarrow \alpha, \mathit{nodeid}$
	get_parent	$\alpha \Rightarrow \alpha, \mathit{parent}$
	get_light	$\alpha \Rightarrow \alpha, \mathit{light}$
	get_temp	$\alpha \Rightarrow \alpha, \mathit{temp}$
	get_noise	$\alpha \Rightarrow \alpha, \mathit{noise}$
	get_tone	$\alpha \Rightarrow \alpha, \mathit{tonecount}$
	get_voltage	$\alpha \Rightarrow \alpha, \mathit{batteryvoltage}$
Aggregation	merge code agg 1 COUNT 2 MAX 3 MIN 4 SUM 5 AVG 6 VARIANCE	$\alpha, \mathrm{agg}_m, \ldots,$ $\mathrm{agg}_1, m, n \Rightarrow \alpha$ n: number of groups m: number of aggregates agg_i: code of aggregate i

Table 2: The SwissQM instruction set