A Virtual Machine for Sensor Networks [*]

René Müller Gustavo Alonso Donald Kossmann
Department of Computer Science, ETH Zurich, Zurich, Switzerland
{muellren, alonso, kossmann}@inf.ethz.ch

ABSTRACT

Sensor networks are increasingly being deployed for a wide variety of tasks. Today, in these networks, the development, deployment, and maintenance of applications are performed largely ad-hoc. Existing platforms help somewhat but also introduce implicit trade-offs. In one extreme, low-level programming platforms and languages make programming cumbersome and error-prone. In the other extreme, declarative approaches greatly facilitate programming but restrict what can be done. In both cases, additional limitations include lack of support for concurrency, difficulties in changing applications, and insufficient abstractions from low-level details. This paper presents SwissQM, a virtual machine designed to address all these limitations. SwissQM offers a platform-independent programming abstraction that is geared towards data acquisition and in-network data processing.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Organization and Design; D.2.4 [**Distributed Systems**]: Distributed Applications; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Performance

Keywords

Wireless sensor networks, virtual machine, efficient bytecode representation, query processing, SwissQM

[*]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.

1. INTRODUCTION

Advances in several technologies (radio communication, sensors, and computing) are making sensor networks a viable option for monitoring the world at large. Lower manufacturing costs are also driving the proliferation of such networks, increasing both the number of sensors as well as the sensing resolution. Examples of existing deployments include tracking a large population of animals [14, 22], monitoring weather and climate [16, 28], monitoring of elderly people [26], and supply-chain management systems [2]. Even though these early prototypes are promising, the development, deployment, and maintenance of applications over sensor networks is still largely ad-hoc and remains an open problem.

1.1 Operating System Level Support

Initial support for programming in sensor networks was provided by systems such as TinyOS [11] and NutOS [24]. Unfortunately, the services provided by these operating systems are rather limited. For instance, TinyOS compiles the application into the operating system and deploys both as a single unit. With this approach, reprogramming or changing the application becomes an expensive and, for large deployments, rather complex task [22].

Although efficient, these systems do not provide adequate software abstractions. As an example, consider *Surge* [10], an application running on top of TinyOS. Using Surge, every network node samples one specific sensor periodically and sends the readings via multihop-routing to a sink node. The sensor to be sampled is hard-coded and cannot be changed once Surge is deployed. Only the sampling period can be modified at run-time. The trade-off here is clear. Surge consists of 400 lines of code, and requires only 17 kB Flash and 2 kB of SRAM memory (for data) on Mica2 [30] nodes. However, to change the application, the entire software stack of each mote needs to be replaced. This can only be done by physically redeploying the network.

A more powerful tool is *Deluge* [12]. Deluge allows sending the application image over the radio, thereby avoiding physical presence at each node. Although an improvement, Deluge still has its limitations. The image is split into several messages and reassembled during reception on every node. On the nodes, replacing the program image requires reprogramming the Flash memory, and this reprogramming consumes a relatively high amount of energy (writing a 16-byte block in Flash memory requires 900 μJ [22] for Mica2 at 3 V battery voltage) in addition to the energy consumption involved in broadcasting the image (sending a 36 byte message requires 216 mJ [22] per transmission). The life time of the network is also affected since Flash reprogramming can only be done a limited amount of times (around 10'000 times on Mica2). This limitation is also present in approaches that introduce a secondary, back-bone wireless network used to maintain the primary sensor network [1]. In both cases, the fundamental problem is that reprogramming is too coarse-grained, especially if only a small fraction of the application needs to be changed.

Overall, these platforms are indeed efficient, but they do not provide the proper software engineering abstractions for a mass deployment of long-lived sensor networks. In addition to limiting dynamic changes, none of these platforms support concurrency or the deployment of several applications, and these platforms make it difficult to maintain sensor networks with a long life-cycle.

1.2 High-level declarative support

A radically different approach to program sensor networks is provided by systems like TinyDB [21] or TASK [3] (which is based on TinyDB and provides a more complete toolsuite). TinyDB allows to program a sensor network using a SQL-like query language (in fact, only a subset of SQL is supported). TinyDB thus offers a declarative interface that abstracts from the low-level details of a sensor network and logically transforms the sensor network into a virtual set of relational tables. As a result, TinyDB is almost trivial to program as compared to OS-level platforms. Furthermore, TinyDB supports multi-programming (although, there is a limitation to two concurrent queries in the current implementation due to main memory constraints). Finally, TinyDB is naturally tailored to continuously changing the application (i.e., the query) running on the sensors. It is far more efficient in distributing applications, and it can support long lived deployments.

The problem with TinyDB (and TASK) is that it does not provide sufficient control over the network. TinyDB is also not extensible, since it is built directly on top of TinyOS and any extension to TinyDB requires rewriting or modifying the whole TinyDB system. Extensions are indeed required because SQL, the query language of TinyDB, is not Turing complete. For example, TinyDB limits in-network processing to data aggregation and it lacks support for user-defined functions. Hence, important functionality such as data cleaning or virtualisation [13] cannot be implemented on the current version of TinyDB.

1.3 Contributions

The two approaches described above represent the two extremes of a software stack for sensor networks. Systems like TinyOS are useful for abstracting the underlying hardware but they are not adequate for the development of applications. Systems like TinyDB significantly simplify the use of sensor networks but do not provide sufficient expressiveness and extensibility. What is missing between the two is a software layer that provides both runtime and development support. The runtime abstraction of this additional layer must be higher than that of low-level operating systems. The development abstractions should give full (Turing complete) expressiveness and flexibility.

Following these requirements, this paper presents and evaluates SwissQM (**S**calable **Wi**reles**S** **S**ensor network **Q**uery **M**achine), a virtual machine intended to act as the missing software layer between low-level operating systems and high-level programming platforms for sensor networks. SwissQM is a stack-based, integer virtual machine that runs programs written in a bytecode language specialised for data acquisition and data processing in sensor networks. SwissQM has a small footprint: 33 kB of Flash and 3 kB of SRAM memory (on the Mica2 platform). It consists of the bytecode interpreter, an operand stack for the stack-based virtual machine and a transmission buffer that is used to store message data and can also serve as temporary storage for programs. Programs can reserve memory space through a data structure called synopsis. The synopsis is used for, e.g., data aggregation and for maintaining state over several invocations of the program. The sensor nodes running SwissQM form a tree topology rooted at a gateway node that provides access to the sensor network. SwissQM is able to execute up to six QM programs concurrently. This limitation stems from the available memory of today's hardware for sensors. SwissQM can, in principle, run an arbitrary number of concurrent programs.

SwissQM provides a much better platform for developing sensor network applications than existing low-level systems. Compared to query-based systems it offers much greater flexibility and programmability. Developers, however, have a much cleaner and higher level interface than TinyOS and NesC [10] (an extension to the C programming language that reflects the encapsulation and execution model of TinyOS). Using a VM model also provides extensibility and customisation: the bytecode can be extended as the underlying hardware evolves. Compared to high level systems, SwissQM offers only a bytecode interface. Yet it makes it possible to design compilers and applications using standard tools. A system like TinyDB can be easily implemented on top of SwissQM and such an implementation naturally inherits additional features from SwissQM such as extensibility, efficiency, and support for user-defined functions, as well as sophisticated in-network processing. These features are the direct result of having the proper programming abstractions to characterise programs, of the benefits of a well characterised bytecode, and of an architecture that is easily extensible.

The remainder of this paper is organised as follows: Section 2 describes the architecture of SwissQM. Section 3 explains the programs that can be run on the SwissQM and gives examples. Section 4 presents the implementation of data aggregation in SwissQM. Program distribution and result routing is shown in Section 5. Section 6 discusses related work. Section 7 concludes the paper and discusses avenues for future work.

2. SYSTEM ARCHITECTURE

SwissQM is built primarily for battery operated sensing devices that communicate with each other through a wireless ad-hoc network. SwissQM runs on sensor networks, which consist of any number of sensor nodes and one dedicated gateway node that acts as the interface to the outside world. Each sensor node contains a (small) micro-processor and possibly several sensors, which measure internal and external signals.

2.1 Topology

To support arbitrarily large networks, SwissQM is based on multi-hop routing. Because SwissQM has been designed primarily for data acquisition and in-network processing, the communication pattern in the network is *multi-source to single-sink*. Like similar systems [10, 21], SwissQM establishes a spanning-tree over the network's connection graph with the root of the tree maintaining a direct link, e.g., over wire, to the gateway (Fig. 1).

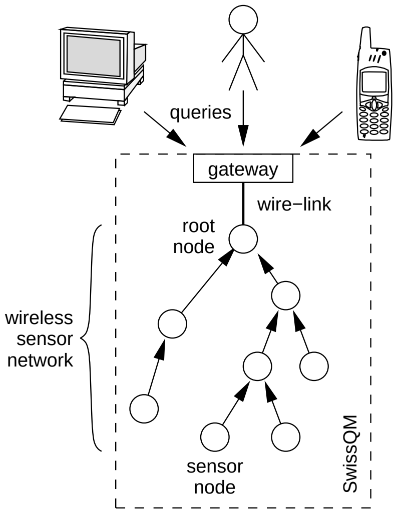

Figure 1: Tree topology of the sensor network with a gateway connected to the root node

2.2 Sensor Nodes

The sensor nodes run the *Query Machine* (QM) that interprets the SwissQM bytecode. The QM runs natively and is implemented in NesC [10] on top of TinyOS [11]. Currently, the system has been deployed mostly on the Berkeley *Mica2* node platform [30]. A Mica2 node has an 8 bit Atmel ATmega 128L processor with 128 kB Flash program memory and 4 kB SRAM for dynamic program data (stack, heap, etc.). The ATmega processor has a Harvard architecture that separates the data and program memories. Memory is accessed through 16-bit wide addresses. The flash memory uses word-addressing (i.e., two bytes) whereas the SRAM uses single byte addressing. Mica2 motes communicate through a proprietary radio module, a ChipCon CC1100 that operates on the 915 MHz ISM band. When running TinyOS, the size of a message is 36 bytes: seven bytes for the header and 29 bytes for the payload. We currently use Mica2 motes equipped with Mica sensor boards featuring a temperature sensor, a brightness (light) sensor, a microphone, and a tone detector.

2.3 SwissQM Gateway

The gateway is typically a more powerful device that is connected to the UART port of the root Mica2 node. The current implementation of the SwissQM gateway requires Java J2SE 1.4 or higher and runs on a Linux-based, 266 MHz ARM-type IXP420 processor with 32 MB RAM.

The gateway provides a single access point to the sensor network and a clean separation between the sensor network and the applications running on top of it. The external interface of the gateway can be anything, ranging from a simple RPC-based API to a Web service. Although not strictly

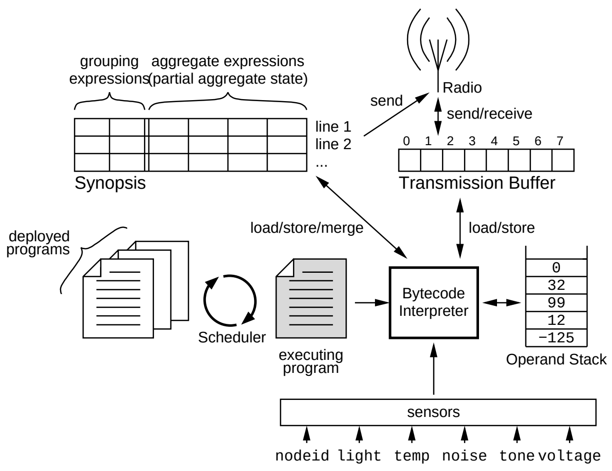

Figure 2: Query Machine Architecture

necessary, the gateway is intended as the place where the bytecode is generated (through a compiler or other means). Commands issued from the gateway can be used to stop and remove a given program, reset nodes, and control sensors; e.g., turn the LEDs of a given sensor node on and off to easily locate the node in a physical deployment. The gateway can also be used to cache sensor data and carry out additional processing that is more efficiently performed at the gateway than in the network. For example, data cleaning pipelines [13] are naturally implemented at the gateway. Additional examples of advanced processing that can be performed at the gateway are given in [23].

2.4 Virtual Machine

The query machine (QM) is a stack-based, integer virtual machine that runs on the sensor nodes (it does not run on the gateway). The main design choice was between a stack and a register-based machine. We chose a stack-based machine based on the results presented in [25]. A stack-based VM has typically smaller bytecodes for the same program than a register-based VM. Programs of a register-based VM needs less instructions than programs of a stack-based VM because no instructions are needed to put the operands onto the top of the stack. However, instructions can be represented in a much more compact way in a stack-based VM because the location of the operands is implicit, resulting in an overall more compact bytecode of programs for stack-based VMs. A compact representation of instructions is important in order to reduce the size of the bytecode for application programs. A small, compact bytecode consumes less memory and less bandwidth when it is disseminated, and it also increases the reliability of program dissemination.

Another design consideration is the number of instructions supported by the bytecode. There is an obvious trade-off between expressiveness and size: the more instructions are supported, the larger the encoding. Although SwissQM is more powerful than TinyDB, the bytecode of SwissQM programs is smaller than the size of the equivalent declarative programs used in TinyDB. As a result, SwissQM can run six concurrent programs on Mica2 nodes whereas TinyDB can run only two concurrent programs on the same hardware.

Fig. 2 illustrates the components of the QM: the *byte-*

code interpreter, the *transmission buffer*, the *stack*, and the *synopsis*. Programs are executed when a timer fires (at a configurable interval called *sampling period*) or when data from other nodes arrive. When a program is scheduled for execution, the corresponding *code section* (see below) is executed by the *bytecode interpreter*. The program stores the operands of its bytecode instructions on the stack. For example, the integer addition instruction `iadd` removes the two top-most elements from the stack, computes the sum, and pushes the result back on top of the stack. As another example, instructions reading sensors push the resulting data onto the stack.

The *transmission buffer* is an array accessed through an index. Programs write the payload data of result messages in the transmission buffer before initiating the transmission. Correspondingly, when a node receives a message, the QM copies the payload data of the incoming message into the transmission buffer before executing the corresponding SwissQM program. Additionally, the transmission buffer can serve as a direct-access temporary storage for programs. This feature further reduces the number of instructions in the program since it is more efficient to access this linear memory space than constantly manipulating the stack. This use of the transmission buffer is similar to the concept of local variables in the Java Virtual Machine.

The *synopsis* is a table used for data aggregation and, unlike the transmission buffer, to keep state between different invocations of the same program. The layout of the synopsis is explained in detail in Section 4.2.

2.5 Data Types

In order to reduce the number of instructions and the footprint size of the implementation, the query machine provides only a single data type: a signed 16-bit integer type. Boolean types used, e.g., in conditional branch instructions are interpreted following the C language rule, 0 for `false` and any other value for `true`. Floating-point types are currently not supported. When memory capacity increases in the future, floating-point support, i.e., a new data type and the corresponding instructions, can be added easily. The lack of floating-point processing is not too restrictive. First, most processors used in sensor networks do not feature a floating-point unit so that the computations need to be emulated in software anyway. Second, the raw data returned by sensors is typically an integer value from an A/D converter; sensors rarely generate floating point values. Third, the evaluation of floating-point expressions can easily be carried out at the gateway. For example, an average value of some sensor readings can be computed as the quotient of the sum and the count, which are both integer values.

Using a single data type has also additional advantages for efficient data manipulation. There is no type-checking involved at runtime, all data elements have the same size, and access to both the transmission buffer and the synopsis can be implemented with a simple element index rather than a byte address.

2.6 Memory Layout

It is important to differentiate between the SwissQM application itself and the user programs that are executed by SwissQM. Throughout the paper, the term *program*, represented using bytecode, refers to the latter. Both the SwissQM application and programs must be stored in memory, but the requirements are different.

The memory of sensor nodes typically consists of persistent memory (Flash) and volatile data memory (SRAM) and is organised as shown in Fig. 3. The SwissQM application code resides in the Flash memory. The volatile data memory in SRAM stores the global state of the SwissQM application. This memory region is further divided into two sections, the *initialised data section* (`.data`), which holds initialised global variables and constants, and the *uninitialised data section* (`.bss`), which holds uninitialised global variables of the SwissQM application. The content of the initialised data section is copied from Flash into SRAM during the boot process. The current implementation of SwissQM uses 3 kB of SRAM memory. The remaining 1 kB SRAM is used as stack for the SwissQM application.

SwissQM allocates a 384 byte sized heap in the volatile `.bss` section. When a new program is loaded, memory is allocated on this heap in order to store the dynamic data structures for this program as well as its bytecode. Every program has its own stack (16 bytes deep) and transmission buffer (16 bytes wide). If requested by a program, another 16 bytes can be allocated for the synopsis structure. Fig. 3 depicts the application heap for two concurrently executed programs; a synopsis is only allocated for Program 1.

SwissQM keeps a fixed list of six program descriptors. Each descriptor contains information about its associated program, e.g., the sampling period, the current epoch counter value, and the length of the program. Most importantly, the descriptor keeps a pointer to the program's data structures (stack, synopsis, and transmission buffer) on the heap. When a program is stopped, its program descriptor is invalidated and the heap memory deallocated. For managing the heap, a memory allocator is integrated into the SwissQM system. The complexity of this memory allocator is low because all memory allocation is static at the time when a program is loaded. That is, programs cannot dynamically allocate memory (there is no *malloc()* instruction in SwissQM). Since all references to heap data use an indirection, a simple compacting allocator was implemented for SwissQM because compacting the heap only requires updating a single reference for each program.

Footprint aside, the main reason to use only static memory allocation is to allow the gateway to perform program admission control. With static allocation, the gateway knows exactly how much memory a program needs and can thus forward to the network only as many programs as can physically run on the nodes. The gateway performs similar resource calculations for other parameters (timing, sampling intervals, etc.). This greatly simplifies the code at the nodes; programs do not need to check for any overflow condition and do not need to make allowances for thrashing situations.

2.7 Instruction Set

SwissQM uses a small subset of the integer and control instructions of the Java Virtual Machine (JVM) specification [19]. The current implementation of the system contains 59 bytecode instructions. 37 instructions are identical to the JVM specification. They include stack manipulation, arithmetic, logic, and control instructions such as conditional and unconditional jumps. The remaining 22 instructions provide access to the sensors or are used for accessing the synopsis and the transmission buffer. The complete list of all instructions along with a formal descrip-

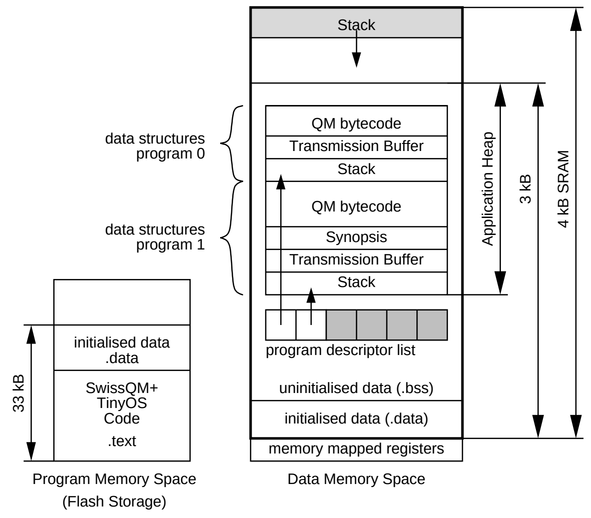

Figure 3: Memory layout in SwissQM with two programs (program 1 requiring a synopsis)

tion and execution duration is shown in Table 4 in the Appendix. Currently, each instruction is encoded in a single byte, which imposes a limit of 256 instructions. The total instruction length with operands can be one, two, or three bytes, depending on whether the instruction has implicit operands (e.g., `iadd`), an 8-bit immediate operand (e.g., `ipushb`=push byte immediate), or a 16-bit immediate operand (e.g., `ipushw`=push word immediate). The implementation of instructions is modular to enable extensibility. The modularity is achieved by organising instructions into groups according to their general functionality. Each group is implemented as a separate NesC component. The groups of the current SwissQM design and the space each group requires both in Flash and SRAM memory is shown in Table 4. The core of the QM, which consists of the bytecode interpreter and the associated logic, takes 23 kB of Flash memory. The implementation of the 59 instructions requires another 10 kB of Flash memory resulting in a total footprint of 33 kB for the current version of SwissQM. The extensibility of the instruction set depends on two factors. The space required in Flash memory and the space required in the SRAM memory. The former limits the number of instructions SwissQM can support. The latter affects how much space remains for programs. The current version implements a rather small instruction set. Since there are more than 80 kB of Flash memory available, it is possible to extend the initial instruction set with more powerful instructions. For instance, if an application ends up generating programs in which a long sequence of instructions appears often, it is possible to encode that sequence into a single bytecode instruction, as already suggested in [4]. As another optimisation, it is possible to extend SwissQM and add more sophisticated arithmetic operations, e.g., vector-operations or operations on the whole synopsis. This optimisation makes the programs more compact. Due to space constraints, this paper does not discuss such optimisations in detail. In general, such optimisations are application specific and involve language optimisations and memory trade-offs that must be taken into account by the compiler. Nevertheless, the possibility of extending the bytecode instruction set and implementing such optimisations is one of the key advantages of SwissQM.

2.8 Execution Time

Table 4 (in the Appendix) lists the average running time of each instruction currently implemented in SwissQM. The running time of each instruction was measured by instrumenting the SwissQM application code. For each bytecode instruction executed, the instrumented code sets an I/O register at the start and at the end. Setting a bit in an I/O register takes 2 clock cycles (i.e., 270 ns) on the ATmega128 micro-controller. Thus, the overhead of the instrumentation (which is included in the measurements) is small compared to the execution time of the actual bytecode instruction. The execution duration was measured as the resulting pulse length of the signal observed with an oscilloscope connected to the I/O port.

As shown in Table 4, most instructions are in the order of tens of micro-seconds. Exceptions are instructions for sending messages and certain instructions for reading sensor data. Both of these kinds of instructions are in the order of milliseconds. The execution times for the `send_tb` instruction that broadcasts the contents of the transmission buffer to the network, and for the `send_sy` instruction that broadcasts the synopsis, depend on the amount of data that needs to be sent. Furthermore, since a contention-based MAC layer for the radio communication is used, additional random delay is introduced. For example, we measured 22–42 ms for sending a full transmission buffer (`send_tb` instruction). The execution time for sensor sampling instructions depends on the type of sensor. For the sensors on the *Mica sensor board*, the execution times for the sampling instructions vary between 0.3 and 3.4 ms. Reading the light and temperature sensors uses a single A/D conversion and requires about 0.3 ms. Accessing the microphone takes longer (3.4 ms) because the A/D converter is accessed 10 times and the largest value obtained is returned. Reading system attributes (e.g., `get_nodeid` and `get_parent`) is faster and in the order of microseconds because no physical sensors need to be accessed.

3. QM PROGRAMS

QM programs are written in SwissQM bytecode and are interpreted by the virtual machine running on the nodes.

3.1 Program structure

A program is identified by its program number. Messages generated by the program contain this identification number to allow nodes to correlate messages and programs. Data acquisition tasks in sensor networks are performed according to one of the following two sequences of well defined phases. The first sequence is used when no in-network aggregation takes place, i.e., a node operates only on its own data and forwards it. The phases in this sequence are *sampling*, *processing*, and *sending*. The second sequence is applied when nodes aggregate their data with the data received from other nodes before forwarding the result. This second sequence has four phases: *sampling*, *processing*, *merging received data*, and *sending*. The program structure in SwissQM is divided into three sections that are combined to implement these different processing phases:

Delivery section: The instructions of the delivery section of a program are periodically executed in response to a timer event. Associated with the delivery section is an execution interval, called *sampling period* that determines the time between two invocations of this section. The delivery section is intended to be used for instructions that sample the sensors and generate a stream of data tuples that is sent towards the gateway node. Result messages generated in the delivery sections contain an *epoch number* that associates the data of the message with a particular invocation of the delivery section. The epoch number is incremented after the execution of the delivery section is completed. By default, the epoch number is always added to the data sent by an application. An additional argument specifies whether the synopsis is to be implicitly cleared at the end of each execution (section argument `"epochclear"`) or whether the `clear_sy` instruction must be used to explicitly clear the synopsis (section argument `"manualclear"`). The delivery section is the only mandatory section of a valid QM program.

Reception section: This section is executed when a node receives a message from any of its children. The section is optional and is used to intercept the message, e.g., in order to aggregate data obtained from the children. If no reception section is present, a node simply forwards the data to its parent.

Init section: This section is executed once at the beginning before the delivery and reception sections are executed for the first time. It is optional and can be used to initialise the synopsis.

3.2 Basic sampling and sending

A first example of a SwissQM program involves a simple program that samples the temperature once every minute. Every node reports the temperature and its node ID. Intermediate nodes simply forward the messages from their children. The corresponding QM program is as follows:

```
1 .section delivery, "@60s"
2      get_nodeid        # read the node's ID
3      istore     0      # store it at pos. 0
4      get_temp          # read temperature sensor
5      istore     1      # store it at pos. 1
6      send_tb           # send transmission buffer
7
8 .section reception
9      send_tb           # forward tuple from child
```

Lines 1 and 8 declare the two sections (delivery and reception) defined in this QM program. In the *delivery* section the node ID is read and pushed on the operand stack (Line 2). This node ID is copied to the first position of the transmission buffer (Line 3). Next, the temperature sensor is read and its value is copied into the second position of the transmission buffer (Lines 4 and 5). The content of the transmission buffer is sent to the parent node (Line 6). The QM knows the size of the transmission buffer from the indices used in previous instructions to copy data into the transmission buffer.

In this example, the *reception* section is unnecessary because the default behavior of simply forwarding incoming messages from the children to the parent is sufficient. For presentation purposes, Lines 8 and 9 of the example program show how this default behavior can be implemented in a SwissQM program. To give another simple example for the use of a reception section, a program can filter out messages from its children by not forwarding them; i.e., by removing the `send_tb` instruction from Line 9. The bytecode of this whole example program requires 8 bytes and can be disseminated to the nodes of the sensor netword using a single message.

This simple example already illustrates two important properties of SwissQM. First, any code section can write to the data transmission buffer. As a result, SwissQM must take care of race conditions. SwissQM avoids race conditions by executing each code section in an atomic way. This mutual exclusion is enforced through global variables. Second, several obvious optimisations are possible. For instance, a node could merge its data with the data received from its children into a single message. This approach reduces the number of messages even if no aggregation is carried out. For brevity, this paper does not describe such optimisations in detail.

3.3 Control instructions

A second example illustrates how to generate and propagate events. In this example, the nodes sample their light and temperature sensors every 10 seconds. If at least one of these readings exceeds a given threshold, the node reports its ID. A possible use of this program is the detection of fire and its localisation (the position of a node is known at the gateway). The program is as follows:

```
1  .section delivery, "@10s"
2        get_light          # read light sensor
3        ipushw     900     # push light threshold value
4        if_icmpgt  send    # jump to send if greater
5        get_temp           # read temp sensor
6        ipushw     500     # push temp threshold value
7        if_icmpgt  send    # jump to send if greater
8        goto       skip    # skip sending
9  send: get_nodeid         # get the node's ID
10       istore     0       # store it at pos. 0
11       send_tb            # send transmission buffer
12 skip:
```

The code is largely self-explanatory. The *reception* section has been omitted because the default behaviour is sufficient. The bytecode of this program consists of 18 bytes and can be disseminated using two messages. An important aspect of this example is the execution time. When a message is sent, the program can take up to 65 ms; the bulk of the time is spent by the `send_tb` instruction in Line 11 as the MAC layer needs to be accessed and the data sent over the low bandwidth radio. If no message is generated, the program takes 22 ms. In this case, most of the time (10 to 20 ms) is spent by TinyOS, which enforces a delay between two samples of different sensors. TinyOS enforces such a delay in order to compensate for a hardware problem on the Mica2 platform. The TinyOS code specifies a 10 ms delay; in practice, however, the delay can be up to 20 ms due to timing issues. This type of complex overhead calculations for the programs is one of the reasons to perform the planning of the execution of concurrent programs at the gateway. The bottom line is that these subtle but crucial problems are one of the reasons to use SwissQM in the first place: Having a well characterised bytecode language makes the cost analysis feasible and the predictions over the resources consumed by a program sufficiently accurate.

	description	function	sample period	bytecode size	# fragment messages
p_0	raw light	$y_k = l_k$	2 s	4 B	1
p_1	raw temperature	$y_k = t_l$	2 s	4 B	1
p_2	arith. operation	$y_k = \frac{1}{2}(l_k + t_k)$	4 s	8 B	1
p_3	max. temperature	$y_k = \max_{i=0,\dots,i} t_i$	4 s	15 B	1
p_4	EWMA filter	$y_k = \alpha y_{k-1} + (1-\alpha) l_k$	4 s	16 B	1
p_5	window average	$y_k = \frac{1}{5}\sum_{i=0}^{4} t_{k-i}$	8 s	46 B	3

Table 1: Six QM programs used for multi-program execution experiment

3.4 Keeping state

A third example illustrates how a program can keep state across different invocations. As pointed out in the previous subsections, such state cannot be kept in the transmission buffer because the transmission buffer is flushed after the execution of a code section. To keep state across invocations of a code section, a program must declare the use of a synopsis. In this example, every node samples its light sensor every 5 seconds. In order to smooth noise from the sensors, a user-defined function with an *exponential weighted moving average* (EWMA) filter is applied to the readings (forgetting factor $\alpha = 0.8$), i.e., the filter produces $y_k = \alpha y_{k-1} + (1-\alpha)u_k$, where y_k is the new computed value, y_{k-1} is the value produced in the last iteration, and u_k is the actual sensor reading. The nodes send the value of y_k and their ID.

```
 1 .section init
 2    get_nodeid         # get the node's ID
 3    istore_sy    0     # store it in synopsis pos. 0
 4    get_light          # sample u0
 5    istore_sy    1     # store it as y0 in synopsis
 6
 7 .section delivery, "@5s","manualclear"
 8    get_light          # sample uk
 9    iload_sy     1     # read y(k-1)
10    isub               # uk - y(k-1)
11    ipushb       5     # push 5 on stack
12    idiv               # (uk - y(k-1))/5
13    iload_sy     1     # read y(k-1)
14    iadd               # y(k-1) + (uk - y(k-1))/5
15    istore_sy    1     # update synopsis y(k-1) <= yk
16    send_sy            # send synopsis
```

In addition to the *delivery* section, this program also contains an *init* section that is called once before the *delivery* section is run for the first time. In this example, the synopsis is used to store the filter output y_k. It also shows that the synopsis can be used as buffer for the result message to be sent (Line 16). An interesting optimisation that a compiler could take advantage of is to store data that must be sent in every invocation but never changes (e.g., the node ID) in the synopsis. In the program, the node ID is setup once in the *init* section (Lines 2–3) when the synopsis is first initialised. Lines 4 and 5 sample the sensor and store it in the synopsis to provide an initial value y_0. Since the synopsis state has to be kept between invocations of the *delivery* section, the argument `"manualclear"` is specified in the section declaration (Line 7). This declaration specifies that the program does use a synopsis. In the *delivery* section the light sensor is sampled (Line 8), and y_{k-1} is read from the first position of the synopsis (Lines 9 and 13). Lines 10–14 compute y_k such that the errors introduced by integer arithmetic are minimal. The synopsis is updated (Line 15) and the complete synopsis (node ID at Position 0 and y_k at Position 1) is sent to the parent (Line 16). The size of the complete program is 19 bytes and fits into two messages.

The execution of the *delivery* section requires 23–43 ms. The execution time without `send_sy` is only 940 μs. Thus, computation costs are negligible compared to the cost of sending a message. This observation is a clear indication that the processing capacity of the nodes can be used without really interfering with energy consumption or the time-budget of a program. In general, the amount of computation is limited by the amount of memory. Again, one of the important advantages of SwissQM is that SwissQM makes memory consumption (only static allocation) as well as time and energy consumption of programs predictable so that all planning can be carried out at the gateway.

3.5 Concurrent programs

A direct consequence of the optimised use of resources and the use of a well characterised bytecode to represent programs is that SwissQM can support a higher level of multi-programming than more traditional platforms for programming sensor networks. As part of the evaluation of the platform, we have performed an experiment in which six different programs are concurrently run on a single node. The programs are characterised in Table 1; l_k represents a light reading, t_k a temperature reading. Each program computes a different function of the light and temperature readings. Each program uses a different sampling period, but all programs produce a single value y_k for each epoch k.

Like all other experiments reported in this paper, this experiment was performed on a Mica2 node. The node and its sensors was placed in a room with a lamp. The experiment ran for four hours in the afternoon until dusk in order to capture a drop in temperature and light. The lamp was turned on and off during two intervals in order to produce clearly identifiable changes in light and temperature. Fig. 4 shows the data obtained by programs p_0, p_1, p_2, and p_3. The results produced by programs p_4 and p_5 are similar to those of p_0; these results are shown in Fig. 5 (using a different scale to highlight the differences).

This experiment illustrates several important advantages of SwissQM. First, the curves confirm observations made in [13]: There is a great deal of noise in sensor data. This noise can be seen in the graph for p_0 in Fig. 5. This noise can be reduced by applying filter functions such as p_4 and p_5. The efficient in-network implementation of these functions motivates the design of a powerful and flexible platform for programming sensor networks such as SwissQM. Programs

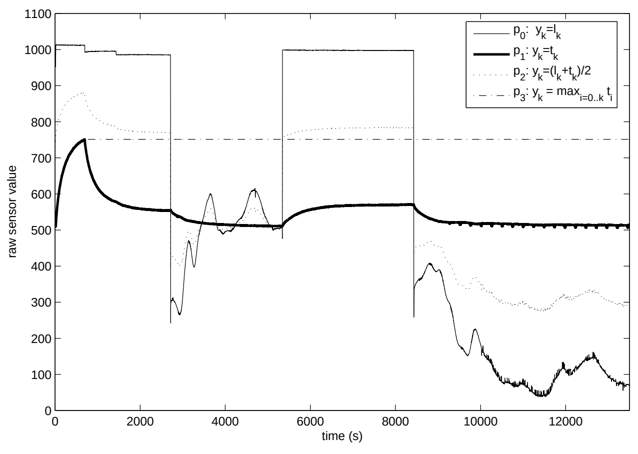

Figure 4: Complete results for p_0, p_1, p_2 and p_3

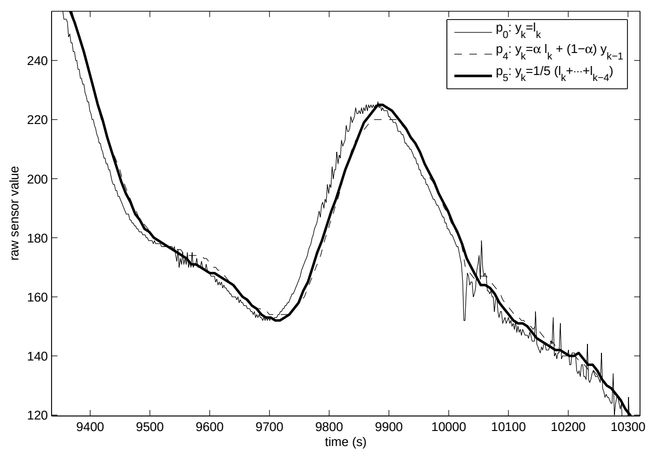

Figure 5: Detailed zoom for programs p_0, p_4 and p_5

that capture and send sensor data are simply not enough. In practice, the need for keeping state and specifying complex control flow in a program is even larger. This experiment is greatly simplified. In practice, programs oversample in order to get a better signal and the programs only send a fraction of the captured value (after application of aggregation and filtering functions) through the sensor network.

The second advantage illustrated is this experiment is the ability of SwissQM to run programs concurrently. The implementation of the six programs in this experiment was carried out manually (not using a high-level language) and purposefully in a naïve way. As a result, the execution is extremely inefficient: Every program samples the light and temperature sensors separately; the readings are not shared between the programs. A clever compiler at the gateway would sample both sensors every two seconds and compile much more efficient bytecode that allows the sharing of the sensor data in all programs. Such an approach is described in [23]. Nevertheless, despite these inefficiencies, the six programs run perfectly well and without interfering with each other, which shows the robustness of the SwissQM engine. Furthermore, SwissQM provides the right abstractions in order to perform the optimisations (e.g., those described in [23]), if these optimisations become crucial.

4. IN-NETWORK DATA PROCESSING

In-network data processing has the potential to reduce the number of messages sent to the gateway [20]. The most relevant form of in-network processing is data aggregation. SwissQM implements it using the synopsis and a specialised instruction: `merge`.

4.1 Synopsis

In the current implementation of SwissQM, the synopsis is a fixed size buffer of 16 bytes allocated on demand when a program is loaded. As shown in the previous examples, SwissQM has several instructions to manipulate and send the synopsis. The synopsis can be used in two ways. In *raw mode*, the synopsis is regarded as an array of 16-bit elements, like the transmission buffer, that can be accessed over an element index. The instructions `iload_sy` and `istore_sy` are used to load data from the synopsis onto the stack and store data from the stack into the synopsis. The example shown in Section 3.4 uses this mode of operation. In *managed mode*, the synopsis is accessed through the `merge` that combines data from the transfer buffer with the synopsis.

4.2 Merge Instruction

The `merge` instruction is used to express a complex operation with a single bytecode instruction. As such, it is an example of the type of powerful instructions that can be included in SwissQM to capture what otherwise would be a repetitive sequence of instructions (since the type of aggregation performed tends to be similar across many applications). `merge` is a parameterised instruction that implements the aggregate operations shown in Table 2. Which one it performs depends on the parameters that it finds on the stack. In Table 2, the state column shows the 16-bit variables used for each type of aggregation. Following the TAG [20] approach, aggregation involves three functions. The *initialiser* function of an aggregate is used to create the initial aggregation state, e.g., the state that represents the data sampled by a node from its own sensors. This state is then merged with the aggregation state received from its children using the *merger* function. The initialiser and merger functions run on the nodes. In SwissQM, the finaliser function runs at the gateway. The finaliser computes the final value of the aggregation from the received data (see example below and Fig. 6). Although not strictly necessary in all cases implemented so far, running the finaliser at the gateway allows more complex aggregation functions to be implemented without having to worry about the impact of such operations on the sensor nodes (e.g., lack of floating point support or complex math processing).

4.3 Merge example

The example illustrates the use of the `merge` instruction and how query processing à la TinyDB can be implemented in SwissQM. By generalising this implementation, it is easy to see how a system like TinyDB could be implemented on top of SwissQM. Consider the following TinyDB query:

```
SELECT parent, MAX(light)
FROM sensors GROUP BY parent
SAMPLE PERIOD 10s
```

Aggregate	State	Initialiser	Merger	Finaliser
`COUNT`	c	$c = 1$	$c = c_1 + c_2$	$= c$
`MAX`	m	$m = expr$	$m = \max(m_1, m_2)$	$= m$
`MIN`	m	$m = expr$	$m = \min(m_1, m_2)$	$= m$
`SUM`	s	$s = expr$	$s = s_1 + s_2$	$= s$
`AVG`	(s, c)	$(expr, 1)$	$(s_1 + s_2, c_1 + c_2)$	$= \frac{s}{c}$
`VARIANCE`	(s, t, c)	$(expr, expr^2, 1)$	$(s_1 + s_2, t_1 + t_2, c_1 + c_2)$	$= \frac{t}{c} - \frac{s^2}{c^2}$

Table 2: Aggregate operations for `merge` instruction

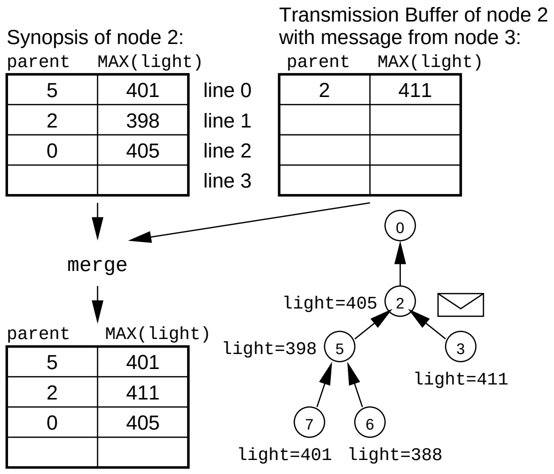

Figure 6: Merging aggregation state from the transmission buffer to the local synopsis

This query returns the maximum light value among all nodes that have the same parent node in the tree. Leaf nodes simply send their synopsis containing the light value sampled every ten seconds and the ID of their parent. Intermediate nodes send a synopsis that includes: (1) its own parent and its own light value, (2) a pair node ID – `MAX(light)` for every non-leaf node in its subtree. In the query, `parent` is the grouping expression and `MAX(light)` the aggregate expression. In general, there can be more than one grouping expression that forms the aggregation groups, as well as multiple aggregate expressions.

In "managed mode", the synopsis is organised as a table. In this example, each line of the synopsis contains information about one group, i.e., a parent ID and the maximum light reading for that ID. Fig. 6 shows in its lower right corner an example routing tree and the synopsis of node 2. Furthermore, it shows that node 3 sent its synopsis to node 2 so that the synopsis of node 3 is stored in node 2's transmission buffer. Using its synopsis and the transmission buffer, node 2 refreshes its synopsis, thereby updating the entry for parent 2. The update of the local synopsis is carried out by the `merge` instruction. Next, node 2 sends its (refreshed) synopsis to node 0, which in turn carries out the same calculations in order to report the final result to the gateway.

This TinyDB query and the data merging algorithm described in the previous section can be translated into the SwissQM bytecode program shown below.

```
1  .section delivery, "@10s","epochclear"
2     get_parent          # get ID of this node's parent
3     istore      0       # store it as group expression
4     get_light           # read light sensor
5     istore      1       # store it as partial agg. state
6     iconst_2            # aggregate type MAX=2
7     iconst_1            # number of aggregate expressions
8     iconst_1            # number of group expressions
9     merge               # merge transmission buffer
10    send_sy             # send synopsis
11
12 .section reception
13    iconst_2            # aggregate type MAX=2
14    iconst_1            # number of aggregate expressions
15    iconst_1            # number of group expressions
16    merge               # merge transmission buffer
```

The aggregation state added by a node is created in the delivery section. Since the `merge` instruction always merges data from the transmission buffer to the local synopsis, the transmission buffer must be prepared before the `merge` instruction is invoked. The same layout must be used in the transmission buffer as in the synopsis. The initial aggregation state consists of a single line with two 16-bit elements, the ID of the node's parent and the reading from the light sensor. The transmission buffer is prepared in Lines 2–5. The `merge` instruction looks for its parameters on the stack. The first parameter at the top of the stack is the number of grouping expressions (in the example, it is 1, the parent ID). The second parameter, as we proceed down the stack, is the number of aggregate expressions (in the example, it is 1, `MAX(light)`). The third parameter is a numeric code indicating the aggregation operation to perform (1=`COUNT`, 2=`MAX`, etc.). If there are more aggregate expressions (the second parameter is > 1), the stack contains one entry indicating the operation for each aggregate expression[1]. These parameters are pushed onto the stack in Lines 6–8. In order to merge the partial aggregation state received from the children, the `merge` instruction is executed a second time in the reception section (Lines 13–16). A message to the parent containing the aggregation state of a node (i.e., the synopsis) is only sent in the delivery section; incoming messages are merged into the synopsis but never forwarded to the parent. The delivery section is declared with the `epochclear` directive; `epochclear` specifies that the synopsis is to be cleared when the delivery section completes. The bytecode of this program is 15 bytes in size and can be sent in one program message. For comparison, TinyDB, disseminates this query in three messages using 168 bytes.

[1]The same mechanism is used in C to implement open parameter lists.

4.4 Managing aggregation

The aggregation example above can be used to illustrate several of the complex system problems associated to in-network processing. The first one is that, as it is done in the example, the synopsis is only sent in the delivery section. This prevents a synopsis being forwarded every time a child sends its data. For the procedure to be correct, when a delivery section of a node is executed, it should have received all the synopses of its children. This leads to a staged execution schedule in which nodes deeper in the tree need to be activated before their parents. In other words, the delivery section must be scheduled *earlier* the deeper a node is in the tree, such that the execution of the delivery section of a node overlaps with the time its parent is listening for results. This requires synchronised program execution of all nodes in the tree. In SwissQM nodes constantly exchange routing information (see Section 5.1). With this information, a node knows the depth of its position in the tree. Each node then uses a simple algorithm for shifting its schedule according to its position in the tree. Such a mechanism is needed by any implementation of the TAG approach [20] for in-network data aggregation. The advantage of SwissQM is that the timing characterisation can be made more precise because of the well characterised building blocks involved.

The second problem is what to do when a node runs out of space and is not able to store the whole aggregation state. In the example of the previous subsection, the aggregation state becomes larger as the aggregation proceeds up the tree. If the network is deep, the whole aggregation state possibly does not fit in the synopsis of a node that is at a high level in the routing tree. As mentioned earlier, SwissQM does not support dynamic memory allocation and that for good reasons: Given the memory constraints of today's generation of sensor nodes, dynamic memory allocation would not solve the problem anyway; instead it would make things worse because the resources consumed of a QM program could not be predicted. The SwissQM solution to this problem is as follows: Once the synopsis of a node is full, the `merge` operation notices this and simply forwards all the synopses it receives without merging. The final aggregation is then performed at the gateway. In such cases, the SwissQM approach to in-network aggregation reduces the number of messages invoked by nodes at the lower layers of the routing tree and SwissQM does as well as possible. In contrast, TinyDB can only implement an "all" or "nothing" approach for in-network aggregation.

The execution time of the `merge` instruction depends both on the size of the synopsis and the state stored in the transmission buffer. Nevertheless, and in spite of its complexity, the `merge` instruction is not too expensive. Initialising the synopsis (i.e., merging the transmission buffer with a previously empty synopsis) requires about 90 μs depending on the grouping and aggregation expressions. Each additional aggregation line adds another 30 μs to the overall execution time. In other words, merging m lines in the transmission buffer with an n-line local synopsis takes approximately $nm \cdot 30$ μs.

5. NETWORK

5.1 Topology Management

The SwissQM routing tree is formed using the *Mint* routing protocol [29] and is used to send result data back to the gateway. Every node in the tree maintains a link to a parent node closer to the root. Thus, a result message is sent to the parent node, which will then forward the message to the next node closer to the root. The protocol establishes a link quality estimator that is based on a *window mean with exponential weighted moving average* (WMEWMA) of the success rate. The success rate for a link is estimated from the number of packets successfully received over that link divided by the number of expected packets for a given period of time. The number of packets is obtained by routing messages that contain routing table entries the nodes periodically exchange (in SwissQM, every four seconds). These tables are then used by the nodes to select their parent nodes. In [29], the authors identified the WMEWMA as a very reliable and robust estimator for the link-quality. An implementation of the Mint protocol is available in the TinyOS distribution (it is also used by TinyDB).

A novel aspect of SwissQM is the embedding of clock synchronisation information into the routing messages of the network layer. This piggy-backing avoids the cost of separate time-synchronisation messages such as those used in TinyDB. Clock synchronisation plays a key role in program scheduling, particularly when aggregation is involved. Every routing message is timestamped by the sender. Whenever a routing message is received from its parent, a node adjusts its clock to the time found in the timestamp of the received message minus an average transmission delay (18 ms, a parameter of the deployment). Our experiments so far show that this simple protocol is accurate enough. An alternative would be to use a more accurate but more complex protocol such as TPSN [8] that requires an explicit two-way message exchange.

5.2 Program Dissemination

QM programs are split into several *fragment messages*. Every program message contains the identification number of the program as well as an enumeration number of the fragment. The first fragment message contains meta-data of the program, i.e., the length of the init, delivery, and reception section, the sampling period, and a number of program flags such as the presence of a synopsis, the synopsis-clear mode (*"manualclear"* or *"epochclear"*) as well as the time when the delivery section is invoked the first time. This program header information uses 10 bytes in the first fragment, leaving 16 bytes for the bytecode of the program. Starting with the second and following fragments, the fragment messages only contain the program and the fragment ID (2 bytes), leaving 24 bytes for the program bytecode. The fragment messages are sent as payload over the broadcast layer. The payload size of a broadcast message is 26 bytes. The broadcast layer adds a sequence number to the message. A sequence number in the broadcast message guarantees that a node rebroadcasts a message exactly once. The size of a broadcast message header is three bytes. A broadcast message is stored as the payload of a TOS message, the basic message type provided by TinyOS. The size of a TOS message is 36 bytes (the default message size on TinyOS). The message mapping between the layers is shown in Fig. 7.

Since wireless ad-hoc networks expose a high bit-error rate, messages are often corrupted and/or lost. SwissQM uses two mechanisms to alleviate the effects of lost messages that contain program fragments. The first mechanism is based on a timeout for program reception and the second

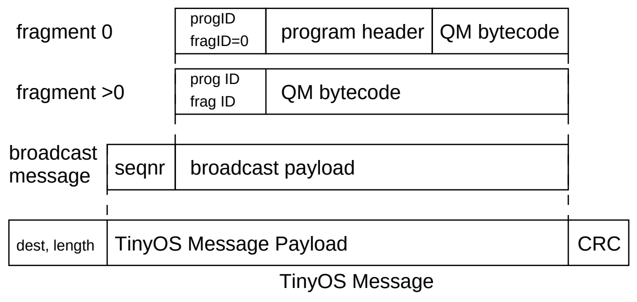

Figure 7: Message mapping: fragment messages → broadcast messages → TinyOS messages

on snooping result messages.

When the first fragment message is successfully received, the number of outstanding fragments is computed from the lengths of the program sections contained in the first fragment. Next, the dynamic program structures (stack, transmission buffer, synopsis) are allocated on the heap and a timer is started. When this timer times out (currently after 2 s) before all fragments of this program are received, a node sends a *program request* message to its neighbours and restarts the timeout timer. The request message contains the program ID and the missing fragments encoded as a bit-mask (16 bits). A node that receives this message and has the specified program, generates the requested fragments for the program. The node then sends each fragment in a *program reply* message back to the requesting node. The reply message also contains the current epoch count and the relative offset to the start of the next scheduled execution. This allows a node to join the execution even if the program execution was started in the meantime.

In addition to this mechanism and in order to accommodate nodes that join late, nodes snoop on the result messages. If a node sees a result message with a program identifier it is not aware of, it sends out a request for all program fragments. Since it is unable to determine the number of fragments of the program, it sets all bits in the 16-bit fragment bitmap of the request message. As soon as it receives fragment 0, it will recompute this mask from the section lengths found in that fragment. This snooping approach is also used in TinyDB.

The difficulty of getting a program disseminated to all nodes is the reason why SwissQM places so much emphasis on compact bytecode and the use of highly expressive instructions, such as `merge`.

5.3 Experiment Program Distribution

This section analyses the program distribution in a large scale network. In order to perform this experiment under controlled conditions as well as due to the lack of such a large number of sensor nodes, the experiment is run in the TinyOS simulator (TOSSIM) [18] rather than on real Mica2 nodes. The simulator is run on a standard Intel-based PC. The advantage of TOSSIM and the NesC compiler is that the same software running on the Mica2 nodes can be executed natively on the platform that runs the simulator. TOSSIM is able to accurately simulate the network behaviour of many nodes. The simulator uses a probabilistic bit error model for the radio links.

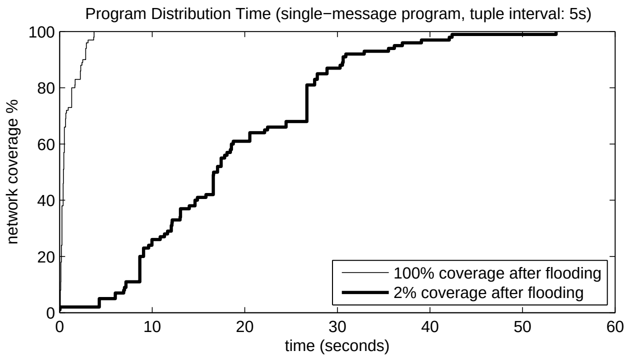

Figure 8: Program coverage during program distribution over time

For brevity, we only report on one experiment that is relevant to understand some of the issues in the design of SwissQM. The experiment consists of a 10x10 grid of 100 sensors with a 4.5 m spacing between the nodes. The average transmission range is set to 15 m. The bit error rates used in the radio links are computed based on empirical data [9] and based on the distance between two nodes. The root node is located in one corner of the grid and injects a single-message program into the network. In this experiment, we measured how long it takes to distribute a program that fits into a single message. We ran a statistically significant number of experiments and picked the best and worst case. The results are shown in Fig. 8, which shows the percentage of nodes that received the program as a function of time. In the best case, all nodes get the program in about 5 s. In the worst case, only 2% of the nodes (the root node and one of its neighbours) successfully received the program during the initial flooding phase. All other nodes must resort to the snooping protocol in order to realise that they have missed the program and then request it from other nodes. For the program in Fig. 8 clearly shows the steps in the network coverage after every 5 seconds, i.e., when a result message is generated, nodes overhear it and realise that they have not received the program. In the worst case, 100% coverage is reached after 53.6 seconds.

These experiments illustrate a well known limitation of ad-hoc wireless networks: the difficulty of achieving reliable communication. Again, this lack of reliability is the reason why SwissQM puts so much emphasis on compact bytecode. If the results are so poor for a single message, distribution of a program that requires 2, 4, or even more messages is quite challenging. In fact, in our experiments such programs almost never reached 100% coverage and often reach only a small fraction of the network. Note that this is a problem of the network. SwissQM (or other platforms) cannot solve it, it can only minimise its effects by minimizing the bytecode and running protocols that are aware of such failures.

To illustrate the advantages of a design like SwissQM over existing systems, we compared the program sizes of TinyDB and SwissQM. The interesting thing about this comparison is that TinyDB, unlike SwissQM, uses a (SQL-like) declarative representation of programs; in theory, a declarative representation should be more compact. In all our experiments, however, the highly optimised bytecode of SwissQM results

Query	TinyDB		SwissQM	
	msgs	bytes	msgs	bytes
`SELECT nodeid,light,temp FROM sensors`	3	168	1	20
`SELECT nodeid,light FROM sensors WHERE temp<512 AND nodeid>50`	5	280	2	30
`SELECT MAX(light) FROM sensors`	2	112	1	22
`SELECT parent,MAX(light) FROM sensors GROUP BY parent`	3	168	1	25

Table 3: Number of messages (and bytes) transmitted for SwissQM programs and TinyDB queries generating the same data

in programs that are not only more expressive but also far more compact. TinyDB uses a rather verbose representation format. One message is used for each field and clause expression. Table 3 shows the sizes of several TinyDB queries and the sizes of the equivalent SwissQM programs. The queries shown cover the entire design space of TinyDB. Overall, SwissQM programs use less messages and less bytes than the corresponding TinyDB queries. In addition, TinyDB developers increased the TinyOS message size to 56 bytes, whereas SwissQM uses 36 bytes (default on TinyOS). Hence, SwissQM not only uses less messages, but also smaller messages. Larger message sizes have the problem that they increase the message loss rate. Results would be worse for 56 byte messages, since the probability of message interference is higher.

6. RELATED WORK

There is a substantial and increasing body of work on sensor networks. The most relevant work to this paper is recent work pointing out the limitations of current sensor data acquisition systems and proposing new abstractions as well as a processing mechanism to build applications. For instance, it is well known that deployment, monitoring, and debugging is cumbersome in existing sensor networks. Hence, [1] proposes the use of a secondary, administrative backbone network to monitor and maintain the primary sensor network. Nodes in the primary network are wired to nodes in the secondary network. The nodes in the secondary network perform all the monitoring and administrative tasks freeing up resources on the primary nodes. This proposal, however, still assumes that the programming on both the primary and secondary network is low-level, with all the limitations that this implies. It has also been pointed out that data cleaning and virtualisation are crucial in building realistic applications on top of sensor networks [5, 13, 27]. This observation is indeed true and one of the most important motivations for SwissQM. Existing platforms like TinyDB or TASK do not support in-network data cleaning and programming it in NesC is very cumbersome. For instance, [31] propose to use wavelets for online data cleaning; implementing such proposals is only feasible on top of a system like SwissQM. Systems like HiFi [7] explore the processing of data streams at higher layers and have indicated that many optimisations would be possible if data processing operators could be pushed down all the way into the sensor network. Again, implementing such a push-down of higher layer functionality (filters, statistical operations, arbitration, spatial and temporal aggregation) in a dynamic manner is only possible using a programmable platform like SwissQM. None of this can be done on existing systems.

Similar memory and processing power constraints are also present in the JavaCard Virtual Machine for smart cards. The JavaCard technology provides a runtime environment for a subset of the Java programming language. However, in contrast to sensor networks, energy consumption is not an issue, as the smart cards are powered externally though the reading device. Additionally, since application bytecode is sent reliably via a wire rather than radio, efficient program representation and a specialised instruction set are less important. JavaCards resemble SwissQM in that they can contain multiple user applications at the same time. One important different, though, is that they are designed for a simple request-response interaction pattern between card and reader. Interaction in SwissQM is more complex. It involves multiple entities in a network.

The execution of the delivery section relates to code execution done by the S machine of *Giotto* [15] and the reception section to code interpreted by the E machine. However, as we are not aiming at hard-realtime systems, we can allow arbitrary code in the delivery section. It is possible to build SwissQM using systems like VVM [6]. However, that would have implied having to implement into SwissQM a lot of the functionality provided by TinyOS in terms of interaction with sensors and network management. Alternatively, we could have used *Maté* [17]. Unlike SwissQM, which is an efficient, Turing complete virtual machine for data acquisition applications, Maté is a generic tool for building application-specific virtual machines. The price for this generality is efficiency. For example, a Maté program similar to *Surge* that collects data and reports it to the sink node requires 108 bytes [17] and it is distributed using 7 messages. The same program in SwissQM takes 7 bytes and fits into a single message. Maté also lacks multi-programming and does not support data aggregation. The authors in [17] speculate that a one-word heap is sufficient. As the aggregation examples used in this paper illustrate, a one-word-sized heap is not sufficient for generic data aggregation.

7. CONCLUSION

This paper presented SwissQM, a virtual machine for sensor networks. SwissQM is intended to fill the gap between existing low-level tools such as TinyOS/NesC and higher application layers. SwissQM is not a sensor network solution by itself, it is a platform for developing sensor network applications. It offers a well defined and extensible bytecode language that can be used as the target platform for compilation from higher programming languages. It also offers a virtual machine environment that handles all the low-level details of running applications on sensor nodes. SwissQM offers significant advantages over what is currently available. The SwissQM bytecode is the most compact and space efficient representation of programs for sensor networks that we are aware of. In the paper, we showed the advantages of this compact representation, both in terms of multi-programming support and reliable distribution of programs across the network.

As part of ongoing work, we are currently implementing duty-cycle strategies for the sensor nodes (where the well characterised nature of the bytecode allows us to obtain very precise estimates of when a node has to be active and when it can be put in low-power mode), compilers for various high-level languages (XQuery, SQL, Java), individual node addressing (so that nodes can selectively run programs), and porting SwissQM to a variety of hardware platforms.

8. REFERENCES

[1] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. In *SenSys 2004*, pages 291–292, 2004.

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

[3] P. Buonadonna, D. Gay, J. M. Hellerstein, W. Hong, and S. Madden. TASK: Sensor network in a box. Technical Report IRB-TR-04-021, Intel Research, January 2005.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, 2000.

[5] E. Elnahrawy and B. Nath. Online data cleaning in wireless sensor networks. In *SenSys 2003*, pages 294–296, 2003.

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

[7] 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.

[8] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *SenSys 2003*, pages 138–149, 2003.

[9] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research, March 2002.

[10] 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 *PLDI 2003*, pages 1–11, 2003.

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

[12] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys 2004*, pages 81–94, 2004.

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

[14] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGOPS Oper. Syst. Rev.*, 36(5):96–107, 2002.

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

[16] K. G. Langendoen, A. Baggio, and O. W. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *WPDRTS 2006*, page 8, 2006.

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

[18] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys 2003*, pages 126–137, 2003.

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

[20] 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.

[21] 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.

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

[23] R. Müller and G. Alonso. Shared queries in sensor networks for multi-user support. In *MASS 2006*, 2006.

[24] E. S. Nut/OS. `http://www.ethernut.de`.

[25] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. In *VEE 2005*, pages 153–163, 2005.

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

[27] A. Terzis, R. Burns, and M. Franklin. Design tools for sensor-based science. In *Embedded Networked Sensors (EmNets 2006)*, 2006.

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

[29] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys 2003*, pages 14–27, 2003.

[30] Mica2 mote platform, Xbow. `http://www.xbow.com`.

[31] Y. Zhuang and L. Chen. In-network outlier cleaning for data collection in sensor networks. In *CleanDB, Workshop in VLDB 2006*, 2006.

APPENDIX

Table 4 lists instructions 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 $\Rightarrow$ respectively. α is used to designate the remainder of the stack that is left unchanged by the instruction.

Group	Instruction	Description	Execution Duration	Bytes in Flash	Bytes in SRAM
Stack Instructions				1562	0
	`dup`	$\alpha, x \Rightarrow \alpha, x, x$	15.2 μs		
	`dup_x1`	$\alpha, y, x \Rightarrow \alpha, x, y, x$	43.6 μs		
	`dup_x2`	$\alpha, z, y, x \Rightarrow \alpha, x, z, y, x$	59.0 μs		
	`dup2`	$\alpha, y, x \Rightarrow \alpha, y, x, y, x$	51.0 μs		
	`dup2_x1`	$\alpha, z, y, x \Rightarrow \alpha, y, x, z, y, x$	67.0 μs		
	`dup2_x2`	$\alpha, w, z, y, x \Rightarrow \alpha, y, x, w, z, y, x$	83.0 μs		
	`pop`	$\alpha, x \Rightarrow \alpha$	14.4 μs		
	`pop2`	$\alpha, y, x \Rightarrow \alpha$	23.6 μs		
	`swap`	$\alpha, y, x \Rightarrow \alpha, x, y$	37.6 μs		
	`iconst_0`	$\alpha \Rightarrow \alpha, 0$	12.0 μs		
	`iconst_1`	$\alpha \Rightarrow \alpha, 1$	12.0 μs		
	`iconst_2`	$\alpha \Rightarrow \alpha, 2$	12.0 μs		
	`iconst_4`	$\alpha \Rightarrow \alpha, 4$	12.0 μs		
	`iconst_m1`	$\alpha \Rightarrow \alpha, -1$	12.0 μs		
	`ipushb <int8>`	$\alpha \Rightarrow \alpha, \text{sign\_ext}(i)$	13.4 μs		
	`ipushw <int16>`	$\alpha \Rightarrow \alpha, i$	13.4 μs		
Arithmetic and Logic Instructions				1056	0
	`iadd`	$\alpha, y, x \Rightarrow \alpha, y + x$	41.0 μs		
	`isub`	$\alpha, y, x \Rightarrow \alpha, y - x$	41.0 μs		
	`imul`	$\alpha, y, x \Rightarrow \alpha, y * x$	42.0 μs		
	`idiv`	$\alpha, y, x \Rightarrow \alpha, \lfloor y/x \rfloor$	71.0 μs		
	`irem`	$\alpha, y, x \Rightarrow \alpha, y \bmod x$	73.0 μs		
	`ineg`	$\alpha, x \Rightarrow \alpha, -x$	30.8 μs		
	`iinc`	$\alpha, x \Rightarrow \alpha, x + 1$	30.0 μs		
	`idec`	$\alpha, x \Rightarrow \alpha, x - 1$	30.8 μs		
	`iand`	$\alpha, y, x \Rightarrow \alpha, y \& x$	40.4 μs		
	`ior`	$\alpha, y, x \Rightarrow \alpha, y \vert x$	41.6 μs		
	`inot`	$\alpha, x \Rightarrow \alpha, \tilde{}x$	31.2 μs		
Control Instructions				1278	0
	`if_icmpeq <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y = x$	34.2 μs		
	`if_icmpneq <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y \neq x$	36.2 μs		
	`if_icmplt <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y < x$	35.0 μs		
	`if_icmple <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y \leq x$	36.4 μs		
	`if_icmpgt <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y > x$	34.4 μs		
	`if_icmpge <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y \geq x$	35.2 μs		
	`ifeq <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x = 0$	25.2 μs		
	`ifneq <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x \neq 0$	23.4 μs		
	`iflt <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x < 0$	22.6 μs		
	`ifle <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x \leq 0$	25.8 μs		
	`ifgt <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x > 0$	23.8 μs		
	`ifge <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x \geq 0$	25.8 μs		
	`goto <int8>`	$\alpha \Rightarrow \alpha$, jump always	6.5 μs		
Buffer Instructions				1528	44
	`iload <int8>`	$\alpha \Rightarrow \alpha,$ `buf[i]`	20.8 μs		
	`iload`	$\alpha, i \Rightarrow \alpha,$ `buf[i]`	29.4 μs		
	`istore <int8>`	$\alpha, x \Rightarrow \alpha$ and `buf[i]` $= x$	23.4 μs		
	`istore`	$\alpha, x, i \Rightarrow \alpha$ and `buf[i]` $= x$	31.0 μs		
	`iload_sy <int8>`	$\alpha \Rightarrow \alpha,$ `synopsis[i]`	21.8 μs		
	`iload_sy`	$\alpha, i \Rightarrow \alpha,$ `synopsis[i]`	30.2 μs		
	`istore_sy <int8>`	$\alpha, x \Rightarrow \alpha$ and `synopsis[i]` $= x$	23.8 μs		
	`istore_sy`	$\alpha, x, i \Rightarrow \alpha$ and `synopsis[i]` $= x$	31.4 μs		
	`send_tb`	send transmission buffer	22–42 ms		
	`send_sy`	send synopsis	22–42 ms		
Sensor Instructions				1854	5
	`get_nodeid`	$\alpha \Rightarrow \alpha, nodeid$	12.1 μs		
	`get_parent`	$\alpha \Rightarrow \alpha, parent$	13.4 μs		
	`get_light`	$\alpha \Rightarrow \alpha, light$	342 μs		
	`get_temp`	$\alpha \Rightarrow \alpha, temp$	348 μs		
	`get_noise`	$\alpha \Rightarrow \alpha, noise$	3.38 ms		
	`get_tone`	$\alpha \Rightarrow \alpha, tonecount$	3.38 ms		
	`get_voltage`	$\alpha \Rightarrow \alpha, batteryvoltage$	804 μs		
Merge Instructions				1064	0
	`merge`	$\alpha, \text{aggop}_m, \ldots, \text{aggop}_1, m, n \Rightarrow \alpha$	# transmission		
		n: number of grouping exprs	buffer lines		
		m: number of aggregates	$\times$ # synopsis lines		
		aggop_i: ith agg. operation	$\times 30\,\mu$s		
	`clear_sy`	clear synopsis	6.9 μs		

Table 4: The complete SwissQM instruction set