Programming iMote Networks Made Easy

Michel Bauderon, LaBRI, bauderon@labri.fr
Stéphane Grumbach, INRIA-LIAMA, stephane.grumbach@inria.fr
Daqing Gu, FT R&D, daqing.gu@orange-ftgroup.com
Xin Qi, CASIA, qixin1984@gmail.com
Wenwu Qu, USTC&ISCAS, quww@ios.ac.cn
Kun Suo, CASIA, kunsuo@gmail.com
Yu Zhang, FT R&D, yu.zhang@orange-ftgroup.com

***Abstract*—In this paper, we report on our experiment with the deployment of a virtual machine, Netquest, which evaluates protocols written in a declarative language, over iMote devices. Netquest offers a new programming model for sensor networks with a high level of abstraction. Protocols and applications can be written in a simple rule-based language, which allows concise and application centric programming. The Netquest Virtual Machine is implemented on top of an embedded DBMS, SQLite, which carries on most of the computation. Our experiments over a small network of iMote devices demonstrate (i) that high-level programming abstraction can be provided on such devices, and (ii) that declarative applications for sensor networks can be developed easily.**

***Index Terms*—Programming abstraction, sensor networks, declarative networking.**

I. Introduction

Sensor networks are made possible by progresses in hardware technologies allowing affordable miniaturized devices with reasonable storage, computing and communication facilities. Their physical limitations, their intermittent availability, as well as the dynamics of the networks raise considerable challenges though for the development of applications. Moreover, many targeted applications require very complex, and sometimes data intensive, functionalities. One of the fundamental barriers today to their development is the lack of programming abstraction [15].

Several recent proposals have been made to circumvent this fundamental barrier and provide high-level programming abstraction to networks of devices. Some systems for sensor networks, such as TinyDB [13] or Cougar [5] offer the possibility to write queries in SQL over the data distributed in the devices. These systems provide solutions to perform energy-efficient data dissemination and query processing. A distributed query execution plan is computed in a centralized manner with a full knowledge of the network topology and the capacity of the constraint nodes, which optimizes the placement of subqueries in the network [16]. Declarative methods have been used also for unreliable data cleaning based on spatial and temporal characteristics of sensor data [8] for instance.

Solutions based on programming languages have been proposed as well, including, beyond standard approaches relying on java, those relying on functional languages, such as Flesk [14], or on rule-based languages. The later has been initially proposed to express communication network algorithms such as routing protocols [12] and declarative overlays [11]. We adopted this programming style, known as **declarative networking**, which relies on the recursive query languages developed in the 80's for database systems, and is extremely promising in the distributed context. It has been further pursued in [10], where execution techniques for Datalog are proposed. Distributed query languages provide new means to express complex network problems such as node discovery [2], route finding, path maintenance with quality of service [4], topology discovery, including physical topology [3], secure networking [1], or adaptive MANET routing [9].

We chose the Netlog language [7], which is well adapted to express protocols and applications over sensor networks. Netlog allows to express applications in terms of deductive rules of the form $head : -body$, which allow to program the behavior of nodes under events. A set of rules coding a protocol is deployed on each node of the network, and runs concurrently on each node. A rule allows to derive new facts of the form "$head$" of the rule, if its "$body$" is satisfied by the local data on the node. The derived facts can be either stored locally, or pushed to other nodes. Both of these actions constitute events that can trigger the firing of rules. The semantics of Netlog has been formally defined in terms of a distributed fixpoint semantics [7]. The examples given in the present paper show that a few rules are sufficient to program applications such as a routing protocol or a average query.

Netlog programs are interpreted by a machine, the Netquest Virtual Machine, which can be easily installed on heterogeneous types of architectures. It relies on a Relational Database Management System, RDBMS, which needs to be embedded on the devices. The Netlog programs are compiled into SQL queries, which form a kind of bytecode of the declarative programs. The main component of the Netquest Virtual Machine is its Engine, which loads the queries corresponding to protocols that can be triggered by the events on the node. It then loads them to the database for execution, and handles the results which are either updates to the local database, or facts to send to other nodes. The Netquest Virtual Machine also has a Router, which handles the communication with the outside world. It differs from classical routers, support as many routing protocols as have been programmed in Netlog, and relies on the database to find the routes to route information in the network.

The porting of Netquest over iMote devices raised some technical problems. First we had to use a RDBMS which is supported by the ARM processor. We thus chose SQLite, which necessitates to slightly revisit our compiler for Netlog programs which was initially designed for SQLServer and MySQL. The limitation of the MAC layer frame size also imposed to limit the size of packets whose payload had to be restricted to one fact at a time.

We tested the system over a small network of iMote devices with different types of programs, including, simple networking protocols, such as a proactive routing protocol, and the construction of a dominating set, as well as a query aggregating data distributed on the nodes. Our results show that the Netquest Virtual Machine runs well on the iMote devices, and that the programs result in expectable communication with no overhead.

The paper is organized as follows. In the next section, we introduce the rule-based language, Netlog, together with simple protocols. In Section III, we present the virtual machine interpreting the declarative protocols, while in Section IV, we describe its installation on iMote devices. Experimental results are presented in Section V.

II. Declarative network programs

In this section, we present informally the Netlog language [7] through some examples of simple protocols that will be used on the iMote testbed. Protocols are expressed by sets of declarative rules of the form $head : -body$, which are evaluated bottom-up, that is if the body is true, then the head is derived. The first protocol we consider, $Hello$, constructs symmetrical links on the network. It consists of three rules, which are running on each node of the network. They are fired in parallel to deduce new facts from the data available locally in the tables on each node.

Let us consider the evaluation on one particular node. The first rule, $Hello1$, is triggered by a time event $timeEvent('link')$ which is produced periodically on the node, and broadcasts a hello message. The symbol "!" in the body, implies that the fact will be deleted after being used by the rule, while the "↑" symbol in front of the deduced fact "$MsgHello(SelfAddr)$" in the head, implies that the fact will be send to the neighbors of the node, where it has been obtained. The variable $SelfAddr$ is interpreted by the node Id on which the rule is executed.

Rule $Hello2$ produces an acknowledgement, while rule $Hello3$ creates the link, upon reception of the acknowledgement. The relation $Link$ stores the neighbors of the node. The symbol ↓ in the head of rule $Hello3$ implies that the fact deduced will be stored locally on the node.

Hello protocol

$$\begin{aligned}
Hello1 &: \uparrow MsgHello(SelfAddr) : -!timeEvent('link').\\
Hello2 &: \uparrow MsgHelloAck(SelfAddr, n) : -\\
&\qquad !MsgHello(n).\\
Hello3 &: \downarrow link(n) : -!MsgHelloAck(n, SelfAddr).
\end{aligned}$$

We next consider two types of protocols to illustrate the power of the language. First, networking protocols, such as a basic proactive routing protocol, and a dominating set construction, fundamental for network organization. Second, a typical sensor network query, the computation of the average of some value, processed in a fully distributed manner, with the construction of a tree during the propagation of the query, on which a convergecast is then performed.

A. Network Organization

The next program constitutes a basic proactive routing protocol. It constructs a routing table, stored in the table $route$ on each node, which contains the attributes: dst: destination; nh: next hop; $hops$: number of hops; and sn: sequence number. Each node creates a route to itself when the program starts (rule $BPR1$). Then each node periodically broadcasts its route table to its neighbors (rule $BPR2$) and increases its local sequence number (rule $BPR3$).

Basic Proactive Routing Protocol

$$\begin{aligned}
BPR1 &: \downarrow route(SelfAddr, SelfAddr, 0, 0) : -\\
&\qquad !timeEvent('initialize').\\
BPR2 &: \uparrow MsgRoute(SelfAddr, dst, hops, sn) : -\\
&\qquad !timeEvent('broadcast'); route(dst, nh, hops, sn).\\
BPR3 &: \downarrow route(SelfAddr, SelfAddr, 0, sn1) : -\\
&\qquad !timeEvent('broadcast'); sn1 := sn + 1;\\
&\qquad !route(SelfAddr, \_, \_, sn).\\
BPR4 &: \downarrow route(dst, snd, hops1, sn) : -link(snd);\\
&\qquad \sim route(dst, \_, \_, \_); hops1 := hops + 1;\\
&\qquad !MsgRoute(snd, dst, hops, sn).\\
BPR5 &: \downarrow route(dst, snd, hops1, sn) : -link(snd);\\
&\qquad !route(dst, \_, \_, sn1); hops1 := hops + 1;\\
&\qquad !MsgRoute(snd, dst, hops, sn); sn > sn1.\\
BPR6 &: \downarrow route(dst, snd, hops2, sn) : -link(snd);\\
&\qquad !route(dst, \_, hops1, sn);\\
&\qquad !MsgRoute(snd, dst, hops, sn);\\
&\qquad hops + 1 < hops1; hops2 := hops + 1.
\end{aligned}$$

When a node receives these information, it first checks if there is any local route with the same destination as the one in the received message. If there is no such local route, a new route is created (rule $BPR4$). If there exists such route, it compares the sequence number of the route received with the one in its local route table. If the sequence number received is larger (rule $BPR5$), the node updates its local route table. If the sequence number is the same, it compares the number of hops of the routes, to select the shortest route (rule $BPR6$).

The next protocol constructs a connected dominating set.

Connected Dominating Set

$$\begin{aligned}
CDS1 &: \uparrow MsgCds(SelfAddr) : -!timeEvent('cds');\\
&\qquad SelfAddr = metaNode.\\
CDS2 &: \downarrow NodeS(1) : -!MsgCds(n); link(n); !NodeS(0).\\
CDS3 &: \uparrow MsgCds(SelfAddr) : -!MsgCds(n);\\
&\qquad link(n); !NodeS(0).\\
CDS4 &: \uparrow MsgAck(n, SelfAddr) : -!MsgCds(n);\\
&\qquad link(n); !NodeS(0).\\
CDS5 &: \downarrow NodeS(2) : -!MsgAck(SelfAddr, n); !NodeS(1).\\
CDS6 &: \downarrow timer('redo', 100000, 1, 1, 3) : -!MsgCds(n);\\
&\qquad link(n); !NodeS(0).\\
CDS7 &: \downarrow NodeS(3) : -!timeEvent('redo'); !NodeS(1).
\end{aligned}$$

It is a structure essential for the construction of network backbones. An initial node, $metaNode$, broadcasts information when the timer event "$timeEvent('cds')$" occurs (rule $CDS1$). When a node in state 0 (original state) receives the message, it (i) updates its state to state 1 (rule $CDS2$); (ii) relays the message (rule $CDS3$); (iii) sends an ACK message to the sender (rule $CDS4$); and (iv) sets a timer (rule $CDS6$). When a node receives an ACK message, it changes its state to state 2 (CDS node) (rule $CDS5$). Note that all these rules are applied in parallel. When the timer is timed out, the nodes in state 1, change to state 3, not CDS, (rule $CDS7$).

B. Sensor Aggregation

The next program performs the computation as well as the processing of an average query in a sensor network. It first propagates the query and constructs a tree in the network (module Query propagation), and then convergecast the results up the tree (module Converge cast).

Query propagation

$$\begin{aligned}
QP1:\ & \downarrow tree(src, n, hops1) : -!MsgQuery(n, src, hops);\\
& \quad \sim tree(src, \_, \_); link(n);\\
& \quad SelfAddr <> src; hops1 := hops + 1.\\
QP2:\ & \uparrow MsgQuery(SelfAddr, src, hops1) : -\\
& \quad !MsgQuery(n, src, hops); \sim tree(src, \_, \_);\\
& \quad SelfAddr <> src;\\
& \quad link(n); hops1 := hops + 1.\\
QP3:\ & \uparrow MsgFather(@n) : -!MsgQuery(n, src, hops);\\
& \quad link(n); \sim tree(src, \_, \_)\\
& \quad SelfAddr <> src; n <> src.\\
QP4:\ & \downarrow timer('leaf', 100000, 1, 1, 4) : -link(n);\\
& \quad !MsgQuery(n, src, hops);\\
& \quad \sim tree(src, \_, \_); SelfAddr <> src.\\
QP5:\ & \quad : -!MsgFather(SelfAddr);\\
& \quad !timer('leaf', \_, \_, \_, \_).
\end{aligned}$$

When a node first receives a query message, (i) it adds the sender of this message as its parent in the tree with root, src, (rule $QP1$); (ii) relays the query (rule $QP2$); (iii) sends an ACK message to the sender (rule $QP3$); and (iv) sets a timer "leaf" (rule $QP4$), which is eventually deleted when the node receives an ACK message (rule $QP5$). The symbol "@" in the head of rule $QP3$ implies that the facts derived are unicast.

The computation of the average climbs up the tree as a pair of values (average, q, and weight, num) as follows. When the timer event "leaf" times out on a leaf node, it unicasts its value to its parent node (rule $AVG1$). When a node receives an answer, it computes the new average value (rule $AVG2$ and rule $AVG3$). When the first answer arrives (rule $AVG2$), it computes the average of the local value and the received value for the subtree. It then adds the other arriving values with the previous result (rule $AVG3$). It also sets/updates a timer "trunk" (rule $AVG4$ and rule $AVG5$).

When the timer event "trunk" times out, the node sends its computation result to its parent node (rule $AVG6$) and generates a fact $ValueAvg(src, 0, 0)$.

Converge cast

$$\begin{aligned}
AVG1:\ & \uparrow MsgAns(@n, src, q, 1) : -!timeEvent('leaf');\\
& \quad tree(src, n, \_); Value(q).\\
AVG2:\ & \downarrow ValueAvg(src, q2, num2) : - \sim ValueAvg(src, \_, \_);\\
& \quad !MsgAns(SelfAddr, src, q, num);\\
& \quad Value(q1); q2 := (q * num + q1)/(num + 1);\\
& \quad num2 := num + 1.\\
AVG3:\ & \downarrow ValueAvg(src, q2, num2) : -\\
& \quad !ValueAvg(src, q1, num1);\\
& \quad !MsgAns(SelfAddr, src, q, num);\\
& \quad q2 := (q * num + q1 * num1)/(num + num1);\\
& \quad num2 := num1 + num.\\
AVG4:\ & \downarrow timer('trunk', 100000, 1, 1, 4) : -\\
& \quad \sim timer('trunk', \_, \_, \_, \_);\\
& \quad !MsgAns(SelfAddr, src, q, num);\\
& \quad SelfAddr <> src.\\
AVG5:\ & \downarrow timer('trunk', 100000, 1, 1, 4) : -\\
& \quad !timer('trunk', \_, \_, \_, \_);\\
& \quad !MsgAns(SelfAddr, src, q, num).\\
AVG6:\ & \uparrow MsgAns(@n, src, q, num) : -!timeEvent('trunk');\\
& \quad tree(src, n, \_); !ValueAvg(src, q, num).\\
AVG7:\ & \downarrow ValueAvg(src, 0, 0) : -!timeEvent('trunk');\\
& \quad !ValueAvg(src, q, num).
\end{aligned}$$

We will see in the next section that the Netlog programs are compiled into SQL queries, together with some meta-information, which are loaded on each node.

III. The Netquest system

To run Netlog programs on device nodes, we chose to develop a virtual machine, the **Netquest Virtual Machine** (NVM for short). Programs are compiled into a bytecode, which is interpreted by the NVM. As an intermediate language for the bytecode, we chose the classical data manipulation language SQL which shall be evaluated by a standard relational database management system (RDBMS). The RDBMS will also be used to store all necessary data : (i) the Netlog bytecode of programs; (ii) network information; as well as (iii) application data.

As a consequence, the virtual machine is a combination of a RDBMS together with functionalities to supplement the RDBMS, e.g., for communication, fixpoint computation, etc. The architecture of the NVM, as shown on Fig. 1, relies on three main components: a Router (handling the communication), an Engine (handling the execution of queries by the RDBMS) and the Local database. Each node runs its own instance of the NVM installed on top of an embedded RDBMS.

A. The Netquest Engine

The Engine is responsible for processing the facts received from the network and evaluating the local programs. It is composed of three modules, the *timer manager*, the *selector* and the *executor*.

- The *Timer manager* is the clock of the Netquest Virtual Machine. It manages all the timers.
- The *Selector* decides which program is triggered when a new fact is received from the network or if there are modifications to the local database.

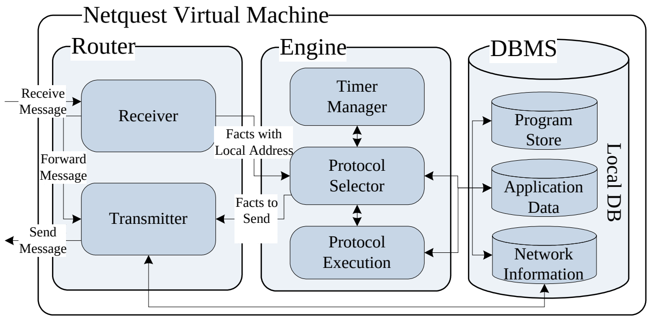

Fig. 1: The Netquest Virtual Machine

- The *Executor* manages the evaluation of the Netlog bytecode.

The evaluation of a program by the Engine uses an event-driven, run-to-completion approach. The firing of a program is triggered by facts received from other nodes or generated by the local Timer manager. These facts are called *triggering facts*. If a node receives a triggering fact when its Engine is currently evaluating a Netlog bytecode, this fact is stored in a cache by the Router, until the evaluation of the current Netlog bytecode terminates. These facts are then loaded by the Engine to trigger new programs.

The Selector checks which programs are triggered when there are new triggering facts. The triggering facts received from other nodes and the triggering facts generated by the local timer manager are treated differently.

For the triggering facts received from other nodes, the Selector checks the trigger table in the local database to find which programs are triggered by the new facts. Each protocol is loaded with its own trigger information when it is installed. The trigger information of a program is generated by the compiler during the compilation of its Netlog code.

The triggering facts are deleted after the execution is finished. If no program is triggered, the triggering facts are discarded directly.

When the triggering fact is a *timeEvent* fact generated by the local Timer manager, the Selector checks the body of the *timeEvent* fact to find which module of which program is triggered. The *timeEvent* fact has several system attributes which are invisible to users or applications. These system attributes specify the modules and programs triggered by *timeEvent* facts. If a program is triggered, the Executor is invoked to evaluate this program.

The Executor then loads the bytecode of the triggered modules, which consists of SQL queries, from the local database, and invokes the local DBMS to execute them. The triggered program is applied over the local database iteratively (fixpoint semantics) until no further modification can be made on the local database by this program. The evaluation of programs is optimized by an extended semi-naive algorithm for Netlog.

The facts deduced by the Executor are either inserted/deleted from the local database, or sent to other nodes depending upon the rules ($\uparrow$ or $\downarrow$ instructions).

B. The Router

The Router is the Netquest Virtual Machine's interface to the network. It interacts with the communication facilities of the device. It handles the emission, forwarding and reception of messages.

It unpacks messages addressed to the node and sends their content to the Engine, when receiving a message whose destination is the node itself. It also packs the facts received from the Engine and sends them to their destinations. It works as well as a forwarder. When the Router receives a message whose destination is not the node itself, it forwards this message to its destination (when it can find the route in the local database).

Before sending a message to another node, the Router checks the route table in the local database to find the next hop for the destination of the message. Otherwise this message is discarded.

IV. Porting Netquest over iMote

The Netquest Virtual Machine has been ported already in different environments. It runs in particular on the network simulator WSNet [6] under Linux, with the RDBMS MySql, and an implementation of the Engine and the Router in C++. The network simulator takes care of the network behavior (e.g., generating nodes, simulation of the execution of code on the nodes, etc.).

A second implementation was developped as a real time, single-computer, multi-threaded simulation using either SQLServer or MySQL as a RDBMS. The computation of each node as well as the routing of messages are done via system threads. Both Router and Engine are written in C♯. This implementation provides a comprehensive graphical user interface under MSWindows, including a syntax directed editor, a compiler, a visualisation tool and a debugger. It has been extended to nodes (smartphone) running MSWindows Mobile 6.0, SQLServer CE as a RDBMS (MySQL is not available) and (almost) the same C♯ code, communication via a Bluetooth connection, getting close to a real size test-bed for distributed algorithms written in Netlog.

All these implementation use fairly powerful systems (in terms of processing power, memory size, battery lifetime, etc.), full size RDBMS and general development tools and language. Even a smartphone is a very powerful system when compared to a sensor. Although there is no such thing as a standard environment or a standard node, a standard RDBMS, a standard language, no such thing as a generally available RDBMS (MySql is not available for mobiles: windows mobile, android, symbian), we could boil everything down to some common denominator and generate (almost) the same bytecode for all these implementations.

A. The Platform

Porting the Netquest system to a realistic test-bed consisting of a network of sensors, raised quite a lot of issues due to their hardware limitations.

We chose to use the iMote2 from Crossbow as hardware platform. It is built around the low power CPU PXA271 and integrates an 802.15.4 compliant radio chip TI CC2420. The PXA271 is based on an ARM architecture without the floating point instructions, and uses a seven-stage integer and eight-stage memory super-pipelined RISC architecture. A C++ development environment is available, allowing to use the C++ implementations of the Netquest Virtual Machine.

Some constraints result from the limitations of the hardware platform. The first one is due to the frame size supported by the platform. According to the IEEE 802.15.4 standard, the maximal MAC layer frame size is 127 bytes. When short address is used, the MAC header length of a data type frame is 11 bytes and 2 bytes CRC are also required by the MAC layer. So the maximal length left for Netquest packets is 114 bytes. Given that Netquest also requires some bytes for its header, the actual length for Netquest data will be less than 114 bytes. Netquest protocol data could consist of multiple facts and each fact could have multiple Attributes. The data is very likely to exceed the limit. We worked to make the Netquest packet header as compact as possible and decided to send only one fact at a time. If multiple facts have to be sent, they will be sent one by one with each one encapsulated in one packet.

The second constraint concerns the functionalities of the DBMS. Well-known RDBMS such as MySQL, Oracle or SQLServer are not supported by the ARM processor which is used in iMote2. We therefore selected a relational, open-source, ARM supported DBMS, SQLite, which is widely used in embedded systems and meets our requirement. SQLite is a software library that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. The source code for SQLite is in the public domain (see www.codeproject.com/KB/database/CppSQLite.aspx). Since SQLite is written in C and we used C++ to implement Netquest, a C++ interface wrapper for SQLite was needed as well.

When compared with ordinary RDBMS, SQLite has some limitations (let us compare with MySql for example):

- Some expressions supported by mySQL are not by SQLite, such as ALTER TABLE and Foreign Key which are not fully supported.
- There is no Boolean data type in SQLite.
- There is a Row Class in MySQL++ for instance, which manages rows from a result set. However, there is no similar class in SQLite. So SELECT cannot return the number of affected rows. In SQLite, the function sqlite3_changes() has an effect similar to mySQL_affected_rows() in MySQL, which returns the number of database rows that were changed or inserted or deleted by the most recently completed SQL statement on the database connection specified by the first parameter.

B. Software Architecture

To port Netquest on this hardware platform with the constraints presented above, we had to develop the following:

- A main program, which provides timers, and task scheduler services to Netquest, and calls Netquest's interface functions.
- A Communication module which let Netquest work with the cc2420 interface. The code will gather/scatter data from/to C++ objects in Netquest, encapsulate/decapsulate frame according to the IEEE 802.15.4 MAC frame format and send/receive frames to/from the physical world.
- A wrapper to enable the Engine to communicate with SQLite.

Data gathering code had to be added in the Transmitter of the Netquest Virtual Machine to collect data in related sub-objects, as well as encapsulation code to format the payload for MAC frame and then call the cc2420 transmit function to send frames out. In the Classifier, packet decapsulation code had to be added to get data from the MAC frame and then scatter the data into related sub-objects. In the main program, we setup a main loop to handle events and receive frames from the cc2420 interface. A sorted event list data structure is implemented and maintained.

After its start, the main program waits for the arrival of a network frame or the occurrence of a timer event. If a timer event occurs or a network frame arrives, the event or frame is sent to the handling module of Netquest. After handling the event or frame, the main program loops to handle the next event or frame. If the Netquest Virtual Machine has to send out something according to the execution of rules, it calls the Transmission module of the NVM, which in turn calls the real transmit function of the cc2420 interface. The transmit function is registered to Netquest as a callback function by the main program when it starts.

Netquest packets are directly encapsulated over the IEEE 802.15.4 MAC frame, since Netquest is protocol independent and the IP address is not required. Without IP/UDP layers, the packet size is reduced. To send and receive Netquest packets, we use PF_PACKET protocol family and SOCK_RAW raw socket. This kind of sockets can send the full 15.4 frame with the MAC header up to higher layers.

V. Experimental results

The declarative programs presented in Section II have been loaded on the iMote devices equipped with the Netquest system. We ran our experiments on a network of 5 to 20 nodes, positioned on a square area of 1.5 meter width. Each node uses the IEEE 15.4 non-Beacon mode on the Mac layer, and it uses channel 15 to communicate with others. The Tx power is set to 5mW and the Rx threshold is set to -30dbm.

A. Networking Protocols

We first ran the basic proactive routing protocol, BPR, presented in Section II on networks of increasing sizes. The protocol runs on the "*links*" created by the *Hello Protocol* creating symmetric links. Fig. 2 and Fig. 3 show the relationship between the number of packets sent/received and the network size respectively during the construction of the routing table. As expected, the number of packets sent grows linearly in the

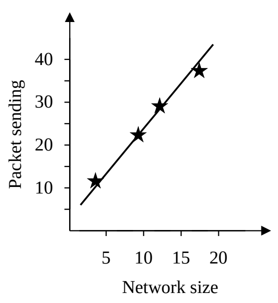

Fig. 2: Packets sent

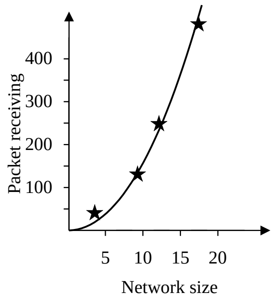

Fig. 3: Packets received

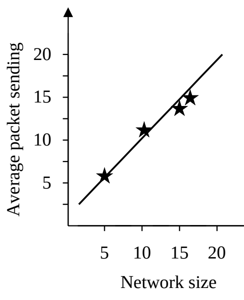

Fig. 4: Packets sent

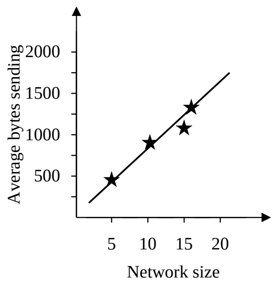

Fig. 5: Bytes sent

size of the network, as for standard routing proactive routing protocols. The number of packets received, grows faster due to the increased degree of the nodes on the testbed area resulting from the density, and the capacity of the routing table. Results for the Connected Dominating Set rules are similar.

B. *Value Aggregation*

Last, we ran the declarative average value computation program, consisting of the query propagation module, QP, and the convergecast module, AVG, on the same iMote networks with different sizes. Fig. 4 and Fig. 5 show now the relationship between the number of packets, respectively bytes, sent and the network size during the average value computation. Fig. 4 shows that the number of packets sent from each iMote increases linearly with the network size. Fig. 5 shows that the number of bytes sent also increases linearly with the network size. It should be emphasized that the resulting algorithm performs both the processing of the average query (construction of the tree), as well as its computation (convergecast along the tree), unlike most declarative approaches which process the query outside the network in a centralized manner.

VI. Conclusion

It has been shown how rule-based languages, such as Netlog, could be used to express network protocols as well as distributed applications in a declarative manner, with very concise programs (a few to a few dozens of rules). We have shown in this paper, that the Virtual Machine evaluating Netlog program could easily be embedded in different environments, including constraint devices such as iMote, as soon as an embedded DBMS was available. The simplicity of the code of the virtual machine results from the fact that it relies on the embedded DBMS for the program evaluation.

This experiment demonstrates that it is therefore possible to develop high-level programming abstraction for distributed systems such as sensor networks, relying on devices which suffer strong limitations on their CPU, and energy in particular. We plan in the near future, to deploy a network of larger size (100 nodes over an indoor or outdoor area), and test fundamental classes of sensor network applications, from networking protocols, including self-organization and routing, to applications, including monitoring and complex query evaluation.

References

[1] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *NETB'07: Proceedings of the 3rd USENIX international workshop on Networking meets databases*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[2] G. Alonso, E. Kranakis, C. Sawchuk, R. Wattenhofer, and P. Widmayer. Probabilistic protocols for node discovery in ad hoc multi-channel broadcast networks. In *Ad-Hoc, Mobile, and Wireless Networks, Second International Conference, ADHOC-NOW 2003 Montreal, Canada*, 2003.

[3] Y. Bejerano, Y. Breitbart, M. N. Garofalakis, and R. Rastogi. Physical topology discovery for large multi-subnet networks. In *INFOCOM*, 2003.

[4] Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi, and A. Sprintson. Algorithms for computing qos paths with restoration. *IEEE/ACM Trans. Netw.*, 13(3), 2005.

[5] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The cougar project: a work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.

[6] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *6th International Conference on Information Processing in Sensor Networks, IPSN*, pages 176–185, 2007.

[7] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain*, pages 88–103, 2010.

[8] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *Pervasive Computing, 4th International Conference*, pages 83–100, 2006.

[9] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 63–68, New York, NY, USA, 2008. ACM.

[10] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA*, 2006.

[11] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *20th ACM Symposium on Operating Systems Principles, Brighton, UK*, 2005.

[12] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA*, 2005.

[13] S. 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), 2005.

[14] G. Mainland, G. Morrisett, and M. Welsh. Flask: staged functional programming for sensor networks. In *ICFP*, pages 335–346, 2008.

[15] P. J. Marron and D. Minder. *Embedded WiSeNts Research Roadmap*. Embedded WiSeNts Consortium, 2006.

[16] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Twenty-fourth ACM Symposium on Principles of Database Systems*, pages 250–258, 2005.