Nano-CF: A Coordination Framework for Macro-programming in Wireless Sensor Networks

Vikram Gupta*†, Junsung Kim*, Aditi Pandya*, Karthik Lakshmanan*, Ragunathan (Raj) Rajkumar* and Eduardo Tovar†
*Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
†CISTER/ISEP, Polytechnic Institute of Porto, Portugal
{vikramg, junsungk, apandya, klakshma, raj}@ece.cmu.edu, emt@isep.ipp.pt

***Abstract*—Wireless Sensor Networks (WSN) are being used for a number of applications involving infrastructure monitoring, building energy monitoring and industrial sensing. The difficulty of programming individual sensor nodes and the associated overhead have encouraged researchers to design macro-programming systems which can help program the network as a whole or as a combination of subnets. Most of the current macro-programming schemes do not support multiple users seamlessly deploying diverse applications on the same shared sensor network. As WSNs are becoming more common, it is important to provide such support, since it enables higher-level optimizations such as code reuse, energy savings, and traffic reduction. In this paper, we propose a macro-programming framework called *Nano-CF*, which, in addition to supporting in-network programming, allows multiple applications written by different programmers to be executed simultaneously on a sensor networking infrastructure. This framework enables the use of a common sensing infrastructure for a number of applications without the users being concerned about the applications already deployed on the network. The framework also supports timing constraints and resource reservations using the Nano-RK operating system. *Nano-CF* is efficient at improving WSN performance by (a) combining multiple user programs, (b) aggregating packets for data delivery, and (c) satisfying timing and energy specifications using Rate-Harmonized Scheduling. Using representative applications, we demonstrate that Nano-CF achieves 90% reduction in Source Lines-of-Code (SLoC) and 50% energy savings from aggregated data delivery.**

***Index Terms*—wireless, sensor, network, macro-programming, coordination, aggregation**

I. Introduction

Wireless Sensor Networks (WSN) are increasingly being deployed for large-scale sensing applications such as building monitoring, industrial sensing, and infrastructure monitoring. Often, sensor networks are deployed in difficult terrains and it is expensive to reprogram all the nodes individually, seriously limiting the manageability and usability of sensing infrastructure. Several macro-programming schemes [1, 2, 3, 4, 5] have been proposed in the past to abstract away from low-level details of sensor networking such as radio communication, analog-to-digital converter (ADC) configuration, memory management and reliable packet delivery. Macro-programming systems typically provide a unified high-level view of the network, allowing programmers to focus on the semantics of the applications to be developed rather than understanding the diverse characteristics of underlying platforms.

Present day sensor nodes have the ability to support a combination of several sensors such as temperature, pressure, humidity, light, audio, accelerometer, and vibration. This variety of sensors attract researchers from different technical backgrounds to make use of a sensing infrastructure for their respective interests. Most contemporary sensor operating systems are designed to support multi-tasking and have APIs to access sensor-hardware but current macro-programming systems or sensor-networking middleware do not provide such capability. A framework, which supports multiple users to write independent applications and execute them seamlessly over a given sensor networking infrastructure, can be highly beneficial for sensor-network researchers and other interested users. Such a system should allow the users* to use the sensor network without them being concerned about other user's applications on the same network.

Many real-world deployments suffer from problems of limited usability and low involvement of users, either because *(a)* the sensors are expensive and it may not be practical to deploy them with ideal or desired density, or *(b)* middleware support to allow seamless deployment of applications is inadequate. The former is addressed in [6] by sharing sensors among multiple deployments through human involvement and to address the latter, design requirements for a middleware to support concurrent applications are outlined in [7]. A typical use-case of supporting multiple applications simultaneously can be conceptualized on a university test-bed deployment like Sensor Andrew [8]. Sensor Andrew is a sense-actuate infrastructure deployed across the Carnegie Mellon University campus. The test-bed is used for inter-disciplinary research ranging from link-layer protocol development to design and testing of applications such as building-energy estimation and social-networking support systems like neighbor discovery.

All the nodes in Sensor Andrew need to be programmed individually for supporting any new application. Furthermore, a user should contact a system administrator to reprogram the network. As this test-bed is an interdisciplinary effort, researchers from different technical backgrounds use the infrastructure for their needs. To help better understand the goal of a middleware framework on a sensor deployment, the following simple example can be used. Consider a task that monitors temperature and humidity in various buildings regularly, and reports it to

*We use the terms 'users' and 'programmers' interchangeably in this paper, as the proposed solution is designed to support the users who are interested in programming the sensor network.

a civil engineering researcher interested in constructing an airflow map of the building. In an another task, a building manager might be interested in using the same infrastructure for a high-priority deadline-based fire alarm system. Users from such diverse technical backgrounds may benefit from a middleware that helps them program the network at an abstract higher level. A conventional macro-programming framework, however, may not allow more than one user to independently program the sensor network for handling these kinds of applications simultaneously. Furthermore, supporting multiple applications pose additional challenge of coordinating tasks on every sensor node and scheduling radio transmissions. In this paper, we propose a framework called *Nano Coordination Framework (Nano-CF)*, to provide support to multiple programmers to write independent applications for a given sensing infrastructure. The framework seamlessly deploys those applications on the end nodes, and coordinates the packet delivery and data aggregation to reduce overall resource usage in the network.

The proposed framework has a global view of various applications running on the network and their mutual interactions, it batches the sensing tasks and radio-usage together, with the help of Rate-Harmonized Scheduling (RHS) [9]. RHS proposes a scheduling scheme to maximize the sleep duration of a processor in case of periodic tasks. In this paper, we adapt RHS to coordinate periodic radio usage tasks such that the packets from several tasks can be transmitted together and smaller packets can combined into larger ones . It is shown in the subsequent sections that significant savings in processor use and packet transmissions can be achieved through such a batching mechanism. In Nano-CF, multiple tasks are coordinated based on their timing parameters, within an allowable deviation as specified by the user. The major contributions of our proposed coordination framework are as follows:

1) It facilitates the use of a sensor-networking infrastructure by multiple programmers for multiple independent applications simultaneously.
2) It leverages the real-time and resource-centric features of underlying sensor-network operating system for providing low-latency response.
3) It also improves the network lifetime by clustering processor usage and radio communication.

The rest of the paper is organized as follows. Section II describes the related research and compares our framework with existing schemes. Section III explains the system design with the details of the hardware platform and the underlying operating system. Sections IV, V and VI describe various components of the framework in detail. These are then followed by an evaluation and discussion in Section VII. Finally, we provide conclusions and future work in Section VIII.

II. Related Work

Creating software for individual nodes can be challenging because of the diverse nature of hardware and operating systems used in sensor networks. Previous works such as [10, 11, 12] describe middleware for facilitating the development of sensor networking applications on individual nodes. On a larger or *macro* scale, macro-programming and reprogramming approaches can be classified as types of middleware for network-level application deployment.

Several macro-programming schemes proposed in the past provide abstraction from the node hardware and network intricacies. Query-based approaches for reprogramming sensor networks such as [13, 3] provide declarative programming expressions for processing data gathered by the sensor nodes. The approach in [13] allows the use of SQL-like queries for getting aggregated response from the sensor network in a transparent fashion. These approaches are advantageous in applications where frequent processing of new but simple queries is required. These schemes are convenient to use, but not very scalable, as individual programming of every node may be required to implement a new query. The other end of spectrum include reprogramming schemes [1] that involve sending application-specific virtual machines to reprogram every individual node. Reprogramming incurs a large overhead as multiple application packets may need to be sent to individual nodes. Compressing the size of reprogramming packets [14] and incrementally programming each node [15] are some of the techniques proposed for reducing this overhead. The frequency of reprogramming the nodes is typically lower than that of data communication, hence it is more beneficial to optimize the resource consumption during the normal use of the network.

High-level programming abstractions like the ones proposed in [2, 16, 4] allow the programmer to view the network as a set of abstract subnets based either on neighborhood, proximity to an event, sensor reading or a combination of the above. These abstractions allow convenient selection of nodes for reprogramming, data collection, and aggregation; thereby optimizing the overall communication in the network. Our proposed framework, Nano-CF, can easily make use of such abstractions for subnet selection and also support multiple applications simultaneously over the whole network.

Recently, there has been some interest in using sensing infrastructure for multiple concurrent applications. The scheme proposed in [17] describes a system for sharing sensors among multiple tasks running on each sensor node. The system also reduces the communication incurred at a sensor node by combining data from each sensing task. The redundancy in computation is minimized by optimally merging the data-flow graph corresponding to each task. Such an optimization, however, is limited to individual node. At a network scale, the interaction between multiple tasks on multiple nodes requires a higher-level framework for efficiently reducing the resource consumption in computation as well as communication.

An implementation to support multiple applications over a sensor infrastructure was demonstrated in [18]. Their sensor infrastructure, however, runs on embedded Linux devices making it impractical for many low-powered devices. Furthermore, their system does not provide any network-level abstraction for programming sensor nodes and users are provided virtual API's for different sensor networking operating systems. Many contemporary sensor node operating systems have support for multiple tasks [19, 20] and we aim to leverage this for deploying multiple applications through our proposed framework.

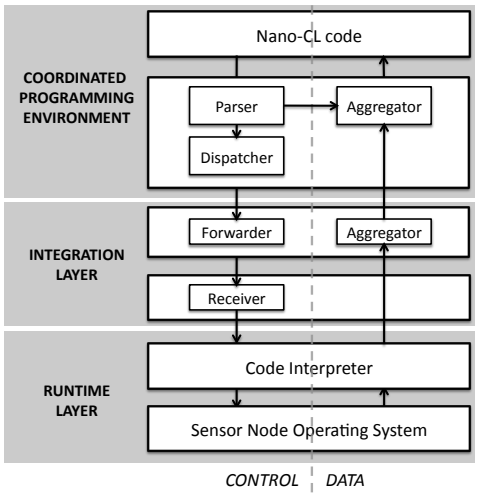

Fig. 1. Layered architecture of Nano-CF

III. System Design

Nano-CF (Nano Coordination Framework) is an architecture for macro-programming with coordinated operations in a WSN that encompasses multiple layers of a sensor networking system architecture, as shown in Figure 1. A main consideration behind the design of our framework is to support data-collection applications. Complex applications with actuation and distributed decisions can still be supported. However, the complexity of the resulting program can be quite high as the framework is not optimized for such programs.

A user interacts only with the top layer of our system, which we call *Coordinated Programming Environment* (*CPE*). For developing applications on the sensor network, a user only needs to write programs using Nano-Coordination Language (Nano-CL), we developed for Nano-CF. The remaining functionality of providing abstraction from the lower-layer networking and topology is handled by the CPE. The functions of the framework are divided into two main ways, *(i)* to handle *control* information including the reprogramming packets, and *(ii)* to collect *data* from sensors. All the three layers contribute towards exchange of control information and data gathering. The CPE provides a programming interface to allow the user to write, compile and send programs over-the-air. The CPE also returns the aggregated data corresponding to each the application independently to the user. The CPE consists of a parser and a dispatcher. The parser or *compiler* converts the functional definitions specified by the user to lower-level byte-codes and then the dispatcher sends them to the deployed sensor nodes. In this paper, we assume that multiple WSNs are connected to each other via the Internet, where each WSN is composed of sensor nodes with at least one gateway node. The communication from the CPE to the end nodes is handled through the *Integration Layer* (*IL*).

The Integration Layer encompasses all the nodes in the network and interfaces to the CPE through a gateway node. The gateway node implements a forwarder function to associate a specific task to a particular node and send corresponding programming packets to end nodes. An Aggregator module spreads over all three layers in Nano-CF to gather data from children nodes at a parent node in the subnets and finally present the result to the user interacting at the CPE layer. In addition to packet delivery and data aggregation, it is the function of the Integrator Layer to make sure that the timing properties of the applications specified by the user are delivered to the OS. Integration layer also handles the responsibility of batching tasks and packet transmissions together using RHS, details of which are provided in the later sections.

Runtime Environment of Nano-CF is implemented at each sensor node, above the operating system. At the lowest layer of this architecture, each node runs a byte-code interpreter to translate low-level instructions from the dispatcher to an executable form for the sensor node. Our architecture is highly portable because we only need to change the code interpreter to support different sensor network operating systems. Further implementation details along with the source code are provided at the project website [21].

A. Nano-RK: a Resource-centric RTOS for Sensor Nodes

Nano-RK [20] is a Real-Time Operating System (RTOS) designed and implemented to support resource reservations for wireless sensor nodes with a multi-hop packet transmission. By leveraging timing characteristics of Nano-RK, such applications can be easily offered through Nano-CF. Virtual energy reservations introduced in Nano-RK also help Nano-CF to manage energy consumptions in each sensor node. By setting reservation values of *(CPU, Network, Sensor)* for runtime environment in a sensor node, we can enforce the expected power consumption. Since most other popular sensor node operating systems do not support multi-tasking by design, our current implementation is limited to Nano-RK. We aim to support more operating systems as future extensions of this framework.

We used FireFly [22] sensor nodes with Nano-RK operating system for our framework. Each node has an Atmel ATmega1281 processor and a Texas Instruments CC2420 Transceiver for IEEE 802.15.4 compliant wireless communication. In addition, a custom sensor expansion card can be connected to the FireFly main board. In particular, the sensor expansion card offers voltage sensing, dual axis acceleration, passive infrared motion, audio, temperature, and light. These diverse sensors supported by the FireFly platform suit the goal of Nano-CF to deploy multiple applicatons simultaneously.

B. Routing and Link Layer

Since Nano-CF requires byte-codes to be transferred to each sensor node in a multi-hop networking environment, the framework requires the support of an underlying routing and link layer protocols. For the purpose of brevity, the details of the routing and MAC layer are not discussed in this paper. Our framework is flexible to operate over any transport layer as long as the gateway node is able to communicate with every node through unique addresses. Many sensor networks employ modified versions of routing protocols such as AODV [23] and DSR [24]. We used DSR-based routing protocol with multicast in this paper for our evaluation. Below the routing layer, the link-layer is crucial for data delivery. Time-Division

Multiple Access based RT-Link [25] and contention-based B-MAC [26] are two commonly used protocols with Nano-RK. In our implementation, we opted for B-MAC, but the operation of Nano-CF is independent of the lower layer protocol used.

Based on this fundamental architecture, we now describe the three main components of Nano-CF; Nano-CL in Section IV, the Integration Layer in Section V, and the Runtime Layer in Section VI.

IV. NANO-CL

We designed an imperative-style language called Nano-CL (Nano Coordination Language) that provides a unified interface to users for writing sensor networking applications. The language has been designed to meet the following design goals:

1) The language should provide an abstraction from the lower-level details of the sensor networking OS and radio communication.
2) The language design should facilitate the extraction of timing and communication properties from user-written applications.
3) The syntax of the language should be simple and easy for non computer-scientists to understand and program.

Each Nano-CL program is composed of two important sections: Job descriptor and Service descriptor, as shown in Figure 2.

```
JOB:
<service1> <nodes> <rate> <agg_func>
<service2> <nodes> <rate> <agg_func>
  . . .
END

SERVICE:
<service1> <return_type>
/* instruction 1 */
/* instruction 2 */
  . . .
END
```

Fig. 2. Format of a Nano-CL program

A. Service Descriptor

In Nano-CL, the user writes a service which is functionally equivalent to a task that is to be executed on each node. Nano-CL consists of a set of primitives and programming constructs which provide sufficient capability for programming the sensor nodes, as well as, an abstraction from the lower-level implementation details of the operating system and radio communication. Each service descriptor specifies the functionality of one task. The syntax for a service descriptor is similar to 'C'-like sequential programming, where the user can make use of pre-defined library functions to interact with the sensor hardware. The return value from the service corresponds to the data value that the user wishes to collect from the sensor nodes, and unlike the usual practice, more than one data value can be returned. The framework converts the user program in service descriptor into byte-codes, which are then sent over the wireless network to be interpreted and executed at each node.

B. Job Descriptor

A programmer can write multiple services and then each service can be mapped to a set of nodes in the job descriptor. The job descriptor section can have more than one service call where each call has the associated timing properties specified by the user. The timing properties include the periodic rate at which the service should repeat at each node and the maximum allowable deviation from the specified period. This deviation allows the framework to "batch" tasks together on sensor nodes, as well as, schedule the transmissions together to reduce the overhead associated with switching on/off the radio and processor. This coordination of tasks and packet delivery across the network is explained in Section V. Set of nodes given by `<nodes>` in the Job Descriptor section contains a list of all the nodes where the respective service should be executed. The nodes in the network should have unique identity, and are mapped to a given physical location in the network. The choice of having an explicit node-list to map the service is deliberate as Nano-CF can leverage adaptive selection of nodes using some of the techniques already proposed in literature such as [16, 27]. The routing layer can provide information about the network topology regularly, which can be used to dynamically select nodes in the job descriptor. The role of the `<agg_func>` is explained in more detail in the next section.

C. Nano-CL Compiler

The Nano-CL compiler (nclC) converts the source code consisting of services into byte-code streams. The compiler is designed with an aim to limit the byte-codes to a small subset of op-codes to allow the code-interpreter task on the end-node to have a small memory footprint. nclC adds metadata to the byte-code stream which helps the integration layer to extract information for batching the computation and radio usage on each node. It also specifies the timing properties for network-wide packet clustering. The metadata in the byte-code stream are generated from the information provided by the user in the **`rate`** section of the job descriptor, which consists of the period of the task and the allowable deviation from the period.

The following are the timing parameters handled by the compiler:

- **`T_srv:`** Repeat rate of the current service.
- **`Dev_srv:`** Allowable deviation in the repeat rate of the service.

The metadata are then sent along with the byte-code to individual nodes and are interpreted at the integration layer and the code interpreter.

D. Example Nano-CL Program

We provide a simple example of a Nano-CF program with two applications implemented using two services, shown in Figure 3. The aim of the first application is to find number of occupied rooms in a building. We use a small network of four nodes at locations `L1,L2,L3,L4` with one node in each room. We assume that each of the nodes have light, audio and temperature sensors, and are placed in such a way that occupancy of the room can be determined by one node. Service `occup_monitor` returns value `1` if room is

```
 JOB:
occup_monitor <L1,L2,L3,L4> <20s,5s> SUM
temp_collect <L1,L2,L3,L4> <50s,0s> NOAGG
 END

 SERVICE:
occup_monitor uint8
int16 light_sense, audio_level;
int32 sum;
int8 cnt, thresh;
sum = 0;
cnt = 10;
thresh=40;
for(i=1:cnt)
    light_sense = gets(LIGHT);
    audio_level = gets(AUDIO);
    sum = sum + light_sense/100;
    sum = sum + audio_level/100;
    wait(1s);
endfor
if(sum/cnt > thresh)
    return 1; // Return 1, if room is occupied
else
    return 0;
endif
  END

 SERVICE:
temp_collect uint16
return gets(TEMP);
 END
```

Fig. 3. An example Nano-CL program with two services

occupied or 0 otherwise. Occupancy†, in this example, is determined by comparing an average of 10 samples of weighted combination of light and audio levels in the room against a threshold. Various parameters in this example are chosen based on experimentation, and may not be applicable in all cases. In order to determine the number of occupied rooms from these locations of sensors, we use the SUM aggregation function. In second service temp_collect, the user just wishes to collect readings of temperature every 50 secs. Please note that each of the services in this example could be created and programmed to a given sensor infrastructure by multiple users independently using Nano-CF. We have shown these services in a single program for ease of presentation.

V. Integration Layer

The Integration Layer (IL) is responsible for byte-codes delivery, data aggregation, optimization of task execution and data transmission on the nodes in the network. This layer, shown in Figure 4, overlays across the gateway and the endnodes.

A. Byte-code Delivery

The Forwarder module on the gateway node forwards the byte-codes generated by Nano-CL to the Receiver module on the end nodes. The primary features related to byte-code transfer are routing table management and fault-tolerant packet delivery.

The routing table generated during the network initiation phase is used for communicating with the end nodes. Sequence numbers, end-to-end acknowledgements and packet retransmissions must be used because one missing packet may cause the network to malfunction due to missing instructions. When end nodes try to reply to a request from their gateway, a broadcast storm problem may occur. In order to avoid this problem, packets are scheduled with an offset as explained in next section. Each packet header contains additional fields like packet type, source address, destination address, application identifier, reprogramming packet sequence number, packet identifier and total number of packets for this application. We shall skip the further details of packet handling in Nano-CF because of space constraints.

B. Data Aggregation

The Integration Layer covers gateway node and end nodes, and links the programming environment to the runtime layer on end nodes. Nano-CF supports in-network data aggregation through the Aggregator module for reducing the packet overhead in the network. The aggregation scheme is defined by the user in the Job Descriptor of the program. In our current implementation, we support common commutative aggregation functions given in Table I. The Aggregator handles different functionality at different levels. At the leaf nodes, the job of the aggregator is to send its own data. At at an intermediate level in the network, it should combine its data with that from all the child nodes according to the specified aggregate function. If the function is NOAGG, then the intermediate node concatenates data from each of the child nodes along with the node id and forwards it towards the gateway. The aggregator module at the gateway node communicates directly with the CPE and provides the aggregated data to the user. Whenever the Receiver module receives a new reprogramming packet from the gateway node, it coordinates with the runtime layer to manage multiple applications from various users.

TABLE I
Aggregation functions supported in Nano-CF

Function	Description
MIN	Minimum value of data
MAX	Maximum value of data
SUM	Sum of all data
COUNT	Number of replies received
NOAGG	Forward all data to CPE

C. CPU/Data Coordination

The IL provides task/network coordination for saving energy on each sensor node. Because most WSN applications are periodic, we utilize Rate-Harmonized Scheduling (RHS) [9].

1) Notation and Assumptions: Suppose that each sensor node n_i is running a set of tasks, Γ_i, which is composed of m tasks, $\tau_1, \tau_2, ..., \tau_m$. Each task, τ_j, is represented by (C_j, T_j, D_j), where C_j is its worst-case execution time, T_j is period, and D_j is deadline. Tasks are arranged in a non-decreasing order of periods. The response time of τ_j is denoted

†We are providing a simple example for understanding purposes. Precision of monitoring is not a goal of this example.

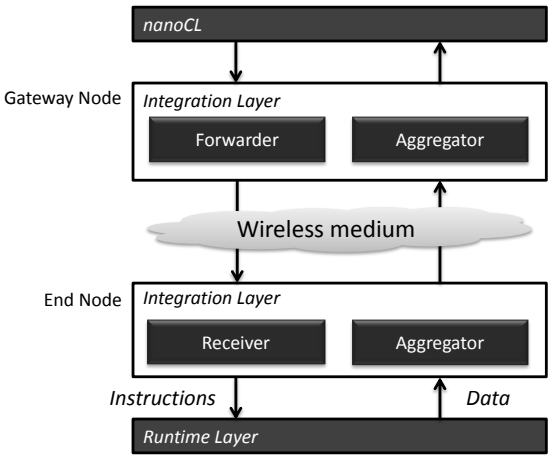

Fig. 4. Integration Layer architecture design and its relationship with other components

as R_j. In this paper, we assume that T_j is equal to D_j. Each task τ_j may also generate a B_j-byte packet ρ_j every $P_j \geq T_j$ time units. Thus, ρ_j can be represented as (B_j, P_j). A packet ρ_j is not dropped at the task level even if it may be lost in routing or the link layer. With P_j being the relative deadline for sending the packet ρ_j, a separate communication task in each sensor node is used to send these packets.

RHS clusters periodic tasks such that all task executions are grouped together in time to accumulate idle durations in the processor schedule. This accumulation helps a processor get a chance to go into a deep sleep state. This property is also applicable to packet transmissions, and sending bigger concatenated packets will consume less energy than sending multiple packets more frequently.

2) Composition with Rate-Harmonized Scheduling: Let packet respose time be the time duration from sensing the environment to the instant when the packet is delivered. Then, RP_j denotes the packet response time of ρ_j. The harmonizing period of tasks, $\mathbf{T_H}$, is chosen so that $\mathbf{T_H} = T_1$ if $\Psi = \emptyset$ and $\mathbf{T_H} = \frac{T_1}{2}$ if $\Psi \neq \emptyset$, where $\Psi = \{\tau_j | T_j < 2T_1, j \neq 1\}$, and $T_j < T_i$ satisfies if $j < i$.

Now we prove some properties of RHS with data clustering.

Lemma 5.1: If a packet is generated by every job of τ_j, the worst-case packet response time, RP_j, for any packet ρ_j is bounded by $2T_j$.

Proof: In the worst case, sensing the environment data can occur at the start of task execution, and packet delivery can occur at the end of task execution. Therefore, RP_j can be represented as $R_j + T_H - \epsilon$, where T_H is added because a packet delivery can be delayed for $T_H - \epsilon$ if a communication task just misses the harmonization boundary. If ϵ is infinitesimal and T_H is T_1 as the worst case compared to $\frac{T_1}{2}$, RP_j is $R_j + T_1$. Since R_j is bounded by T_j due to the implicit deadline of τ_j, $RP_j \leq T_j + T_1 \leq 2T_j$. ∎

Corollary 5.2: Any packet, ρ_j, generated by τ_j will meet its packet transmission deadline if $P_j \geq 2T_j$.

Proof: By Lemma 5.1, if $P_j \geq 2T_j$, ρ_j will be delivered within P_j. ∎

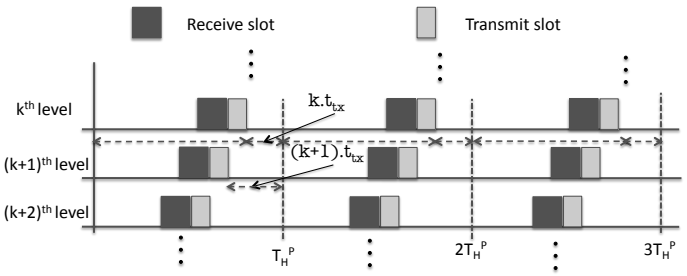

Fig. 5. Timeline of network-wide scheduling based on RHS at various hops in the network.

Theorem 5.3: If τ_c is the communication task and represented by (C_c, T_c), a set of given tasks, Γ, is schedulable if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_c}{T_c} \leq \frac{1}{4} \tag{1}$$

Proof: This follows from *Lemma* 5.2 and *Theorem* 4[‡] from [9]. ∎

The result from *Theorem* 5.3 can be pessimistic because we strictly applied the packet deadline. If $P_j \gg T_j$ for τ_j and ρ_j, Equation (1) can be changed into $\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_c}{T_c} \leq \frac{1}{2}$, which is the same as the result from [9].

3) Energy-Saving with RHS: By using RHS for tasks, a processor in a sensor node can go into a deep-sleep state more frequently (at $\mathbf{T_H}$ boundaries). Applying RHS to packet transmissions allows each sensor node to send a merged packet instead of sending packets whenever it has data in the queue. The amount of energy saved by using RHS for tasks can be obtained from the length of the deep-sleep period given in the form of (C_{sleep}, T_{sleep}) [9]. The amount of saved energy can be derived from the number of transmitted packets. The number of transmitted packets per unit time when we do not use RHS is given by $\sum_{i=1}^{n} \frac{1}{T_i} \lceil \frac{B_i}{B_{max}} \rceil$, and by $\sum_{i=1}^{n} \frac{1}{\mathbf{T_H}} \lfloor \frac{T_i}{\mathbf{T_H}} \rfloor \cdot \lceil \frac{B_i}{B_{max}} \rceil$ when we use RHS. Here, B_{max} is the maximum packet size. We will evaluate these energy savings later.

D. Network-wide batching using RHS

As has been emphasized in earlier sections, a network programming framework like Nano-CF provides a global view of the network where efficient scheduling for packet aggregation at multiple hops becomes feasible. An efficient approach to aggregate packets and schedule data has been proposed in [28], however, we use Rate Harmonized Scheduling across a network to save energy by reducing the frequent turning On or Off of the radio. The main reason to use RHS is that it can help batch tasks together even if the periods of the various network transmissions mismatch. We use RHS in a distributed fashion to batch packet events to be scheduled at the end of the

[‡]Theorem 4 from [9] proves that a taskset is feasible under basic rate-harmonized scheduling if $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 0.5$.

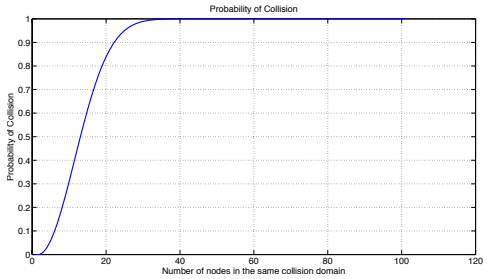

Fig. 6. Probability of collision in a network when the nodes present in the network can transmit uniformly at anytime within the period

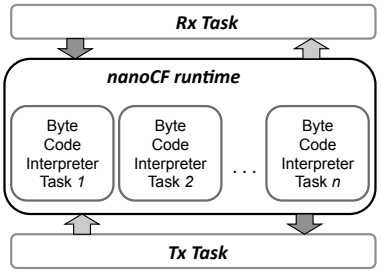

Fig. 7. Block diagram of the runtime layer of Nano-CF

harmonizing period T_H. However, if all the nodes in a subnet are scheduled to transmit at T_H, multi-hop aggregation of the packets can not be supported, and packet collision will be high. However, if the packet transmission at k^{th} hop can be offset to an earlier time, a parent node in the network can efficiently collect data from its children.

For efficient data collection, it can be deduced that each node should transmit at an offset given by:

$$\Omega_k = -(k \times t_{tx}) \tag{2}$$

where, Ω_k is the introduced offset of transmission from T_H, t_{tx} is the maximum amount of time a node uses its radio while transmission and k is the depth of the node in the tree. Equation 2 gives a simple schedule for packet transmissions in a multi-hop scenario. The nodes listen before they transmit as shown in Figure 5 and this schedule can be maintained with some coarse-grained time synchronization even over CSMA (Carrier Sense Medium Access) protocols. This allows collision free scheduling in the network, whereas if the nodes can transmit uniformly anytime within the T_H duration; the probability of collision of any two packets with n nodes is given by:

$$P_c = \frac{N!}{(N-n)! \times N^n} \tag{3}$$

where N is the number of possible slots in the harmonizing period, and is given by T_H/t_{tx}. Figure 6 shows the variation of P_c with respect to n for $N = 100$. It can be seen from the plot that the probability of collision quickly increases with the increase in number of nodes.

VI. Runtime Architecture

The runtime layer of the framework consists of routing, communication and execution of byte-code on individual sensor nodes. In our current implementation, each sensor node in Nano-CF uses Nano-RK. The runtime environment has three types of tasks running on the OS: Receive (RX) Task, Transmit (TX) Task, and a set of code interpreter tasks. There are pre-defined copies of the code-interpreter tasks on each sensor node, corresponding to the number tasks to be supported in the framework. The RX and TX task take care of reliable packet delivery and also implement the routing layer.

A. Routing

The framework requires at least a basic routing layer to ensure connectivity to all the nodes. In our current implementation we use a routing protocol similar to Dynamic Source Routing (DSR) [24], but the system is flexible with respect to the routing layer as long as the higher layer is able to address a node directly. The system can also support on-demand routing schemes, provided the user generates a topology-map of the network before reprogramming the network nodes. The topology map can be generated during the initiation phase and is beyond the scope of this paper. The user can also make use of `send()` and `receive()` primitives available in the language for developing ad-hoc routing schemes.

B. Code Interpreter

As shown in Figure 7 the code interpreter receives byte-code from the Nano-CL compiler through the *RX Task*. Table II lists the primary opcodes handled by the code interpreter. First, the code interpreter reads the metadata section of the byte-code and it saves the period (repeat rate) of the service **`T_srv`** and the deviation **`Dev_srv`** into local variables. The interpreter has a local instruction stack and it executes the byte-code corresponding to each instruction in a sequential manner. The interpreter maintains a local stack of variables and a return stack to support function calls. It repeatedly executes the byte-code with a period of **`T_srv`**, and also listens for any new byte-code packets for node reprogramming. If the code-interpreter task receives a signal from the *RX task* that it has received a new service to execute, it finishes the execution cycle of the current task, flushes the local stacks and copies the new metadata and instructions into the local memory and restarts the execution.

TABLE II
Commonly used Byte-Codes in Nano-CF

Instruction Type	OpCode
Declare/Assign:	`DECL, AEQ`
Arithmetic:	`ADD, SUB, MUL, DIV`
Comparison:	`GE, LE, EQ, GT, LT`
Constructs:	`IF, ELSE, FOR, GOTO`
I/O:	`SET, GET, TOGGLE, CLR`
Nano-RK:	`SEND, RECV, WAIT`
Macros:	`LABEL, SECTION, END`

VII. Evaluation

One of the key goals of a macro-programming framework is to provide usability to the end-user. It should provide enough features to allow a programmer to program the network for most

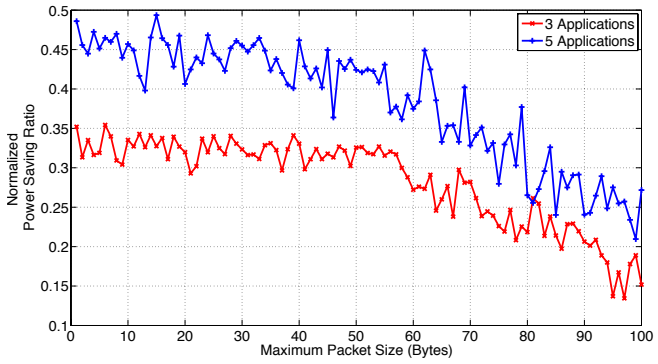

Fig. 8. Applying RHS in packet delivery allows each sensor node save the energy by reducing the number of packets to be sent.

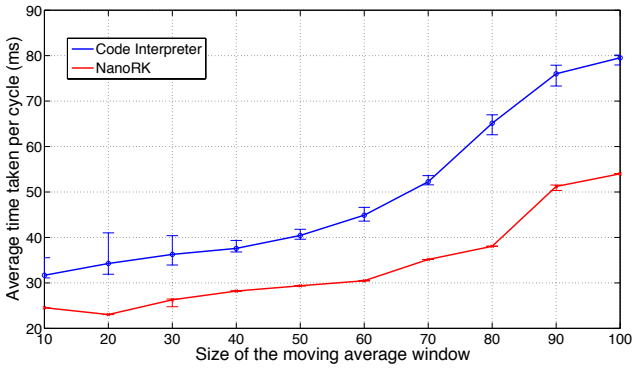

Fig. 9. Code Interpreter performance comparison with equivalent code running on Nano-RK. The micro-benchmark used in this experiment calculates the moving average of light samples. This figure shows the average time taken per cycle by the micro-benchmark along with the corresponding error bars.

applications while being transparent to the underlying complicated details. This trade-off of complexity with transparency is not trivial to evaluate, and is highly dependent on the concerned end-user. Another major important aspect of the framework performance is the overhead in timing and energy consumption. In this section, we will provide detailed evaluation of our framework with respect to the energy-savings, overhead of using a code-interpreter, and usability of our programming language in terms of Source Lines of Code (SLoC).

A. Energy Savings with Rate-Harmonized Scheduling

By clustering task executions anad packet transmissions on the FireFly sensor nodes with RHS, we can reduce the energy consumed on each node. The power consumption of various components of a sensor node is provided in Table III. As sensor nodes usually have low CPU utilization, we can guarantee even longer life expectation of each sensor node. Due to space constraints, we do not include results from task batching, as they are similar to those provided in the RHS manuscript [9].

The framework supports delaying the packet transmission and hence combining the packets together, which yields significant power savings by using the transceiver for shorter durations at lower duty-cycles. This effect is shown in Figure 8. The figure is obtained by estimating relative power savings for randomly generated packets having a constant Period P_j with varying the maximum packet size from each application from 1 to 100. Every data-point shows average after 50 iterations. When 3 applications are used, the energy consumption related to packet delivery can be saved up to 35%. In addition, if we use 5 applications, the amount of energy saving is increased up to 50%. As the maximum packet size increases, the effect of saving energy is decreasing. It happens because large packets may not be merged anymore. In addition, we can obtain opportunities to save more energy due to high possibilty of clustering packets from more number of applications. Aggregating packets together helps in reducing the number of packets transmitted in the network, which in turn reduces the channel contention and packet loss due to collision.

TABLE III
POWER CONSUMPTION INFORMATION OF THE FIREFLY PLATFORM

Power State	Current	Voltage
All Active	24.8 mA	3.0V
Both CPU and Radio Idle	0.20 μA	3.0V
CPU Active	6 mA	3.0V
Radio Tx	17.4 mA	3.0V
Radio Rx	18.8 mA	3.0V

B. Performance Evaluation of the runtime environment

The runtime environment consists of a code-interpreter which executes the Nano-RK instructions corresponding to the received byte-code. We evaluated the overhead of the code interpreter task with respect to compiled code running directly with Nano-RK on the FireFly. The application we tested is a quite computationally intensive task of finding a moving average of light sensor readings. The sensor sample over a window of given size are consecutively added and then divided by the size of the window. We observed the average time taken by the tasks to calculate the average with varying window size. A larger window size means more cycles for processing in the task and hence longer duration per cycle. Figure 9 shows the obtained results. It can be deduced from the plot that the code interpreter does not add much overhead to computationally intensive tasks. We found the percentage overhead of the code-interpreter with respect to native Nano-RK code to be about 55.80%. Sensor networking tasks typically involve less computation, this overhead is quite acceptable. As shown in Table III, the processor power is much lower than the communication power, where using Nano-CF results in 50% savings. In future work, we plan to investigate just-in-time compilation techniques that can reduce the overhead of executing interpreted code.

TABLE IV
COMPARISON OF NUMBER OF LINES OF CODE FROM EXAMPLE IN FIGURE 3

Application	Nano-CL	Nano-RK
Occupancy Monitoring	20	205
Temperature Collection	2	80

Nano-CL allows the programmers to write their applications in small services, where they do not need to consider the details of hardware setup and sensor configuration. We compare the typical number of lines of code a programmer is required to

TABLE V
COMPARISON OF FLASH MEMORY REQUIREMENTS FOR DIFFERENT APPLICATIONS RUNNING INDIVIDUALLY ON A SENSOR NODE

Application	Nano-CL(Bytes)	Nano-RK(Bytes)
Occupancy Monitoring	35306	27932
Temperature Collection	35306	29324

write for a particular application. Table IV gives comparison between the number of Source Lines of Code (SLoC) for the example in Figure 3 to similar applications implemented on Nano-RK. We can see the overhead in case of Nano-RK is more than a factor of 10. The number of SLoCs in Nano-RK programs are significantly higher because of the code required for task and hardware initialization. Comparison of memory footprints for these applications implemented using Nano-CF and directly on Nano-RK is provided in Table V. The flash memory required by a Nano-CL program on a sensor node is the same regardless of the application, as the program is copied into RAM during in-network programming.

VIII. CONCLUSION AND FUTURE WORK

The ability to program a sensor network for multiple simultaneous applications using a macro-programming framework is a desirable feature. In this work, we presented Nano-CF, a framework which allows sensor network programmers to write applications on the sensing infrastructure with a simple macro-programming language. We demonstrated the motivation behind supporting multiple independent applications through a macro-programming on a sensor network using the example of Sensor Andrew. With the proposed Nano-CF, we could save up to 50% of the communication energy when 5 applications are being used simultaneously on the sensor node. The code interpreter overhead was measured to be 55.80% on the average. However, the use of a code interpreter improves the portability and maintainability. Furthermore, Nano-CF macroprogramming allows the user to create applications with significantly reduced complexity. Compared to developing application directly on the sensor node operating system, we can implement same functionalities with only 10-15% of code lines.

For the future work, there is further scope of intelligently combining multiple application's source code to remove any redundancy across tasks, based on the timing properties and user specifications. We plan to support the automated composition of multiple applications together into one or more by identifying common functionalities in the code written by users. This would allow our framework to be more efficient in the usage of resources at both the node and network level by combining the common subparts of tasks and data packets.

REFERENCES

[1] Philip Levis and David Culler. Matè: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 85–95, San Jose, California, 2002. ACM.

[2] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 489–498, Cambridge, Massachusetts, USA, 2007. ACM.

[3] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[4] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, Brighton, United Kingdom, 2005. ACM.

[5] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A.F. Harris, and M. Zorzi. Synapse: A network reprogramming protocol for wireless sensor networks using fountain codes. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pages 188 –196, 2008.

[6] Nithya Ramanathan, Thomas Harmon, Laura Balzano, Deborah Estrin, Mark Hansen, Jenny Jay, William Kaiser, and Gaurav Sukhatme. Designing wireless sensor networks as a shared resource for sustainable development. In *in Information and Communication Technologies and Development*, 2006.

[7] Yang Yu, Bhaskar Krishnamachari, and V.K. Prasanna. Issues in designing middleware for wireless sensor networks. *Network, IEEE*, 18(1):15 – 21, 2004.

[8] Anthony Rowe Mario Berges Gaurav Bhatia Ethan Goldman Raj Rajkumar Lucio Soibelman James Garrett Jose Moura. Sensor andrew: Large-scale campus-wide sensing and actuation. *IBM Journal of Research and Development: Special Issue on Smarter Cities and Sensed Infrastructures*, 2010.

[9] Anthony Rowe, Karthik Lakshmanan, Haifeng Zhu, and Ragunathan Rajkumar. Rate-harmonized scheduling for saving energy. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 113–122, Barcelona, Spain, 2008. IEEE Computer Society.

[10] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Tinylime: Bridging mobile and sensor networks through middleware. *Pervasive Computing and Communications, IEEE International Conference on*, 0:61–72, 2005.

[11] Glauco Vasconcelos Mardoqueu Vieira Nelson Rosa Carlos Ferraz Eduardo Souto, Germano Guimares. A message-oriented middleware for sensor networks. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, 2004.

[12] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems*, 26:351–368, 2004. 10.1023/B:TELS.0000029046.79337.8f.

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

[14] N. Tsiftes, A. Dunkels, and T. Voigt. Efficient sensor network reprogramming through compression of executable modules. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pages 359 –367, 2008.

[15] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 25 – 33, 2004.

[16] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, Boston, MA, USA, 2004. ACM.

[17] Arsalan Tavakoli, Aman Kansal, and Suman Nath. On-line sensing task optimization for shared sensors. In *IPSN '10: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 47–57, Stockholm, Sweden, 2010. ACM.

[18] Christos Efstratiou, Ilias Leontiadis, Cecilia Mascolo, and Jon Crowcroft. Demo abstract: A shared sensor network infrastructure, 2010.

[19] Rzvan Musloiu-E Philip Levis Andreas Terzis Ramesh Govindan Chieh-Jan Mike Liang, Jeongyeup Paek. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys '09: Proceedings of the 7th international conference on Embedded networked sensor systems*, 2009.

[20] A. Eswaran, A. Rowe and R. Rajkumar. Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks. *IEEE Real-Time Systems Symposium*, 2005.

[21] http://www.nanork.org/wiki/nanocf.

[22] Rowe A., Mangharam R., Rajkumar R. FireFly: A Time Synchronized Real-Time Sensor Networking Platform. *Wireless Ad Hoc Networking: Personal-Area, Local-Area, and the Sensory-Area Networks, CRC Press Book Chapter*, 2006.

[23] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, pages 90–100, 1999.

[24] D. B. Johnson, D. A. Maltz, and J. Broch. DSR: the dynamic source routing protocol for multi-hop wireless ad hoc networks. *Ad hoc networking*, 5:139172, 2001.

[25] Rowe A., Mangharam R., and Rajkumar R. RT-Link: A Time-Synchronized Link Protocol for Energy-Constrained Multi-hop Wireless Networks. *SECON*, 2006.

[26] J. Polastre, J. Hill and D. Culler. Versatile low power media access for wireless sensor networks. *SenSys*, November 2005.

[27] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, San Francisco, California, 2004. USENIX Association.

[28] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 131–146, Boston, Massachusetts, 2002. ACM.