ProFuN TG: A Tool for Programming and Managing Performance-Aware Sensor Network Applications

Atis Elsts, Farshid Hassani Bijarbooneh, Martin Jacobsson, and Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden

***Abstract*—Sensor network macroprogramming methodologies such as the Abstract Task Graph hold the promise of enabling high-level sensor network application development. However, progress in this area is hampered by the scarcity of tools, and also because of insufficient focus on developing tool support for programming applications aware of performance requirements.**

We present ProFuN TG (Task Graph), a tool for designing sensor network applications using task graphs. ProFuN TG provides automated task mapping, sensor node firmware macrocompilation, application simulation, deployment, and runtime maintenance capabilities. It allows users to incorporate performance requirements in the applications, expressed through constraints on task-to-task dataflows. The tool includes middleware that uses an efficient flooding-based protocol to set up tasks in the network, and also enables runtime assurance by keeping track of the constraint conditions.

We show that the adaptive task reallocation enabled by our approach can significantly increase application reliability while decreasing energy consumption: in a network with unreliable links, we achieve above 99.89 % task-to-task PDR while keeping the maximal radio duty cycle around 2.0 %.

I. Introduction

Wireless sensor network (WSN) application developers are faced with a number of challenges in the process of moving from a specification of a system to its functioning deployment:

- Create a tailored firmware image for each sensor node in the network depending on its hardware components and software configuration.
- Partition the model of the network in logical regions based on node properties (configuration and other).
- Set up specific application-level tasks on sensor nodes in the network; control and change them during the network's lifetime to improve the reliability and energy efficiency of the system, e.g. after node failures, sensor hardware failures, and radio link failures.
- Determine the mappings from these tasks to nodes that have good probability to satisfy the performance requirements of the application.
- Throughout the application's lifetime, either assure the user that it is still meeting its performance requirements, or report their violations.

These challenges are typically solved in an application-specific way; an approach that is both tedious and error prone.

In this paper, we build on the dataflow programming paradigm and adopt the Abstract Task Graph (ATaG) [1] WSN macroprogramming methodology. We implement ATaG in ProFuN TG [1], a tool that addresses the needs of sensor network programming, deployment and maintenance. ProFuN TG not only allows users to describe the functionality of an application with a task graph, but also comes with support for mapping these task graphs on network nodes, for macrocompilation of their code, and for their deployment both on simulated and real networks.

We go beyond the original specification of the ATaG-based compilation framework [2] and enable *performance-aware* ATaG applications. Our tool supports user-defined application-level performance requirements that are expressed in form of constraints on delay and packet delivery rate (PDR), and set on dataflows between tasks. In this way, we join together existing ideas about runtime assurance through performance monitoring [3] with high-level programming support for WSN.

This extension has two implications. Firstly, at the design stage, these requirements are used by the task mapping algorithm to rule out potential task mappings that have insufficient probability to satisfy them. Secondly, during run time, these requirements are used to enable efficient (i.e. reactive, rather than continuous) feedback from the network to the central system. We implement a middleware for the runtime support; it sets up tasks in the network, manages task-to-task communication, and determines whether the conditions of the constraints hold, enabling runtime assurance through maintenance alert notifications. If configured to do so, it also periodically collects application performance statistics in the central system. The alerts and statistics are used for adaptive task remapping with the dual purpose to satisfy the constraints and to optimize the system.

We show that the desire for efficiency is not a reason to eschew high-level programming. Using trace-based evaluation on a 17-node network with unreliable links, we show that the tool can set up a unique task on each of nodes in less than 30 seconds, and, by making use of task reallocation on redundant sensor nodes, keep task-to-task PDR above 99.89 % with the average radio duty cycle close to 1.0 % and the maximal duty cycle around 2.0 %. To the best of our knowledge, this is the first implementation of an ATaG macrocompiler for the low-power *msp430*-class motes, as opposed to existing toolkits that generate Java byte code [4] and have significantly higher runtime requirements.

[1] http://parapluu.github.io/profun/

ProFuN TG is customizable and flexible: we allow the users to define their own tasks and their own task mapping functions, and we include support for joining together tasks written in several distinct programming languages.

The focus of this paper is on the ProFuN TG tool itself. For a more complete technical explanation and evaluation we refer the reader to the accompanying technical report [5] and to our previous work [6] on this topic that covers the middleware and automated reasoning aspects in more detail. The rest of this paper includes a conceptual background (Section II); a high-level description of the tool (Section III); its architectural overview and a brief description of its main components (Section IV); an evaluation of the runtime performance (Section V), and a comparison with related work (Section VI).

II. Conceptual foundations

A. Programming model

The core concept of ATaG is the *task graph* (Fig. 1), a user-defined graph where vertices correspond to abstract tasks and edges denote dataflows between these tasks. An *abstract task* is a clearly defined chunk of functionality with a fixed interface, such as the number of inputs and outputs. It is similar to a function in most programming languages. However, tasks communicate exclusively by message passing; they do not share state and cannot execute other tasks by using synchronous function calls. Tasks are annotated with properties, such as its firing rule (*periodic* or *event-based*), its firing period, and the number of copies to instantiate. Each abstract task is instantiated on one or more sensor nodes. An *abstract dataflow* is a link that connects a pair of abstract tasks. All dataflows have *scope*: a property that restricts the maximal distance (in number of intermediate hops or network regions) between the source and destination in a communicating pair of instantiated tasks. A dataflow may also have several *constraint* properties (Section II-C), a *number of retransmissions* property, a *datarate* property, and others.

ATaG is a hybrid programming model: the high-level specification is visual and declarative, while the low-level code inside the tasks typically is textual and imperative: the code of the predefined tasks of ProFuN TG is written in C. However, we also include support for tasks written in another, declarative, WSN application-specification language SEAL [7].

ProFuN TG provides a number of predefined task types in several categories: *sensors*, *actuators*, *data processing* tasks, and *other data I/O*. While users are free to introduce their own types of tasks of any category by extending the tool, the two predefined *function* task types are unique in the sense that they can be used to include application-specific code in task graph instances by using just the visual interface of the tool. To do that, the user has to provide the name of the function, its properties (such as the number of inputs and outputs), and its code. The code of a *C function* task consists of several separate blocks, possibly empty, instantiated as separate C language functions. The functions are: *initialization*, called when the task is created; *periodic action*, called when a timer with the task's period expires; *data item received*, called when a new input appears; and *cleanup*, called on termination. All types of tasks, including the predefined ones, share this division of runtime functionality.

The second type of supported function task is ProFuN TG is the *SEAL function*. SEAL is a WSN-specific node-level programming language that is compiled to C. SEAL comes with its own middleware library that implements commonly required functions, such as logical and arithmetic operations, data filtering and aggregation functions. Developing tasks in SEAL has a number of benefits:

- it does not require knowledge of the middleware C API;
- the user is not required to partition the code into several distinct functions; this is done behind the scenes by the SEAL compiler;
- it is comparatively much simpler to implement advanced functionality, e.g. conditional self-rescheduling of the task;
- some classes of potential programming errors are completely prevented. This includes some errors that would freeze the whole sensor node; for example, it is not possible to write a non-interruptible loop in SEAL.

B. Network model

ProFuN TG allows the user to interactively create and refine a model of the network and its environment (Fig. 2). In this way the user is able to incorporate not only his initial knowledge and design assumptions (coming from maps and plans, on-site surveys and remote sensing, expert opinions etc.), but also on-going measurements of the network and its environment.

The core of a network model is a set of sensor nodes connected with radio links. The location of each node is specified visually, by placing it on a background map. A node also has a number of other properties, such as its hardware platform and hardware components. In addition, user defined properties (in *name:value* syntax) can be set. For example, the user may specify one or more location properties, such as the room and the building in which the node is located.

Each radio link has a number of properties that describe its quality (e.g. transmission success probability and transport-layer delay). In the absence of explicit configuration, link existence and quality parameters are estimated by a network simulator. They can also be manually entered by the user, or collected from the network by observing its performance.

We do not restrict these properties to their mathematically expected values (averages), but instead recognize that they are random variables, best described by probability distributions. For a motivating example, consider two imperfect network links with the same average PDR, one of which has bursty packet loss, while the other has independently distributed packet loss. It is clear that the first one is likely to have far higher variance. Now consider a constraint that bounds the maximal delay on a dataflow over a link. Since the end-to-end delay of retransmitted packets is dependent on PDR, this difference must be taken into account for this constraint.

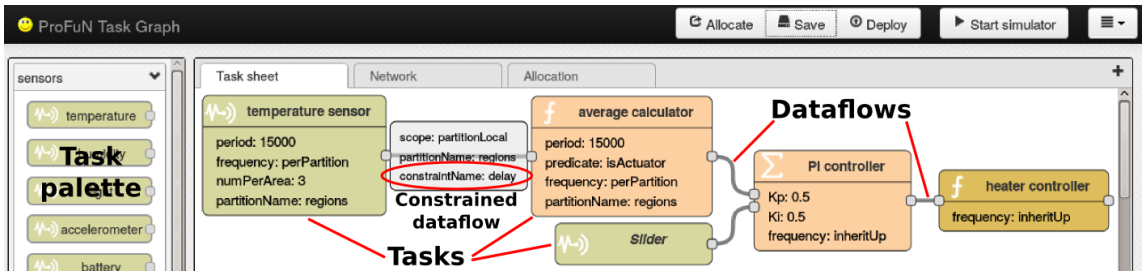

Fig. 1: **The task graph view of ProFuN TG**, showing a heating control application

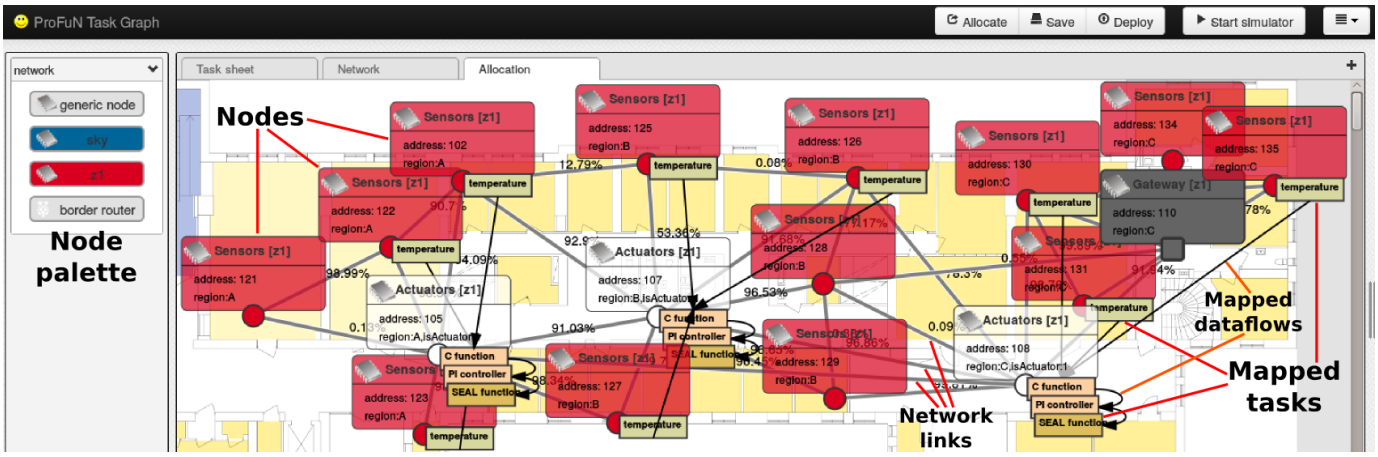

Fig. 2: **The network view of ProFuN TG**, showing the instantiated task graph of the application mapped on a test network

C. Constraints

One of the novel features of ProFuN TG is its support for end-to-end data flow constraints between source and destination tasks. A constraint is described by the minimal acceptable probability P with which it is predicted to hold in the task-mapping stage, and a bound C. For an example, let us take $P = 0.98$ and $C_{\text{delay}} = 3000$ & $C_{\text{PDR}} = 90\,\%$, respectively:

$$\mathbf{P}(\text{Delay} < 3000\,\text{ms}) \geq 0.98$$
$$\mathbf{P}(\text{PDR} > 90\,\%) \geq 0.98$$

These constraints serve two roles:

- Predictive: the mapping algorithm takes the constraints into account and will not produce mappings that violate them.
- Diagnostic: the runtime system continuously tests whether constraint conditions are met. If this test fails on a node, that node notifies the central system, which then reallocates tasks.

III. ProFuN TG functionality

Consider an example application: an indoor heating control system (Fig. 1). This application has two sensing tasks: *temperature* and *slider* (for desired T° adjustments), one actuation task: *heater*, and several data processing tasks: a function that *averages* multiple sensor values, and a *PI (proportional-integral) controller* [8] that outputs the intensity of the required heating, using the difference between the input data and a reference value describing the desired T°.

We assume the application is deployed in a building with several regions (in this example, a region is defined as several co-located rooms), each of which has several sensor nodes.

The requirements of this application include:

- several types of nodes should be supported: a node equipped with sensors and a node connected to a heater actuator;
- each region should have at least one active heater task;
- each PI controller should receive input averaged from at least three temperature tasks located in the same region;
- the delay between the sensor tasks and the processing tasks should be $< 30\,\text{s}$ with at least 99.5 % probability;
- the energy consumed by the network should be minimized, as long as the constraints above are satisfied.

The bulk of the support that ProFuN TG provides to the user can be separated in three distinct stages: *node setup*, *task graph design & mapping*, and *task setup & maintenance*. The tool is designed for and supports iterative development process (Fig. 3), as observing the operation of a real network will lead to changes in the network model, additional functional requirements to modification of the task graph, and hardware failures to reallocation of tasks.

A. Node setup

In the node programming stage firmware images are created and programmed on network nodes. ProFuN TG helps to automate this process by allowing the user to describe the platform of each network node (from a pre-defined palette),

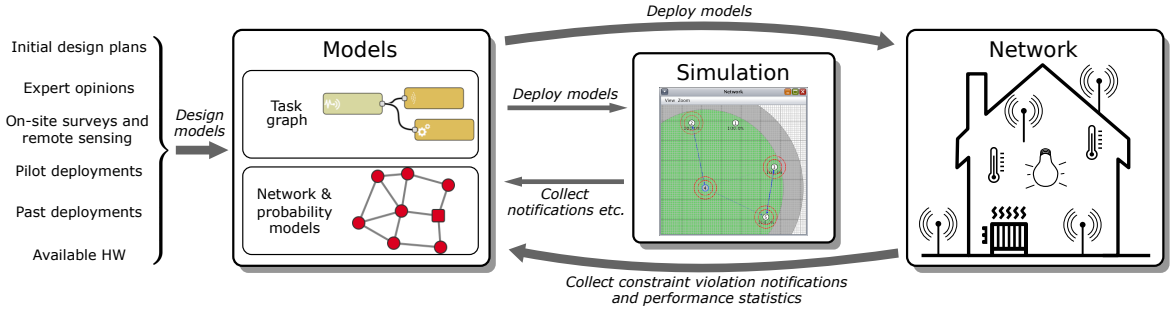

Fig. 3: Using ProFuN TG for iterative development workflow

and then to configure the nodes — to specify their hardware components, their constant properties, and the default values of their variables. Both template-based configuration of multiple nodes at once and configuration of a single node are supported. Taking into account this configuration, ProFuN TG produces a firmware image for each node and automatically deploys it on directly connected nodes; over-the-air reprogramming is an expected future extension of our system.

In the example application, the user starts by defining templates for sensing nodes and actuating nodes. The descriptions of these templates contain the list of enabled hardware component drivers and may also contain properties common to all or most of nodes, such as the identifier of the building. Then the user defines constant properties on each node, for example, region identifiers (a scalar property `region`), and also variables, such as the desired temperature in a region.

B. Task graph design and mapping

In this stage the user designs the application by connecting a number of abstract tasks with specific functionality. Subsequently, the tool maps the task graph to a model of the network. The result is a mapping from the desired application-level functionality to the available nodes.

Tool support is beneficial when specific relations are desired between the tasks. For instance, in the example application the user may want at least *three* distinct temperature sensor nodes sending data to each heater node. ProFuN TG allows to configure these high-level relations easily (i.e., once per network, not once per each pair of nodes), as well as to enforce that these relations are met everywhere in the network. Furthermore, it allows to map tasks only to nodes with a specific configuration. To do that, the user writes a binary *predicate* for a task (i.e., a logical expression on node's properties); the predicate is evaluated at the design time and operates as a filter to produce the set of nodes eligible to host the task.

It is typical for WSN applications to have specific task-to-task reliability requirements. In the general case it is not possible to reduce these requirements to a simple metric such as number of hops between nodes, because there are situations when a single bad link fails to deliver acceptable PDR, whereas a multihop path consisting of several good links succeeds. ProFuN TG combines the user-defined constraints with user-defined network model to automatically determine optimal mappings of task pairs that are within bounds of these constraints.

Once the set of nodes suitable for a task has been decided, that abstract task is mapped on one or more of these nodes (depending on the number of copies desired) — in other words, it is instantiated in the model.

In the example application, the user creates a task graph for the application (Fig. 1) that includes predefined tasks for a temperature sensor and a PI controller. The parameters of the controller (K_p and K_i) as well as its default reference value all are set from the ProFuN TG interface; the reference value can also be changed by user input at runtime, for example by using a slider sensor. The user also implements a task for averaging readings of multiple sensors (including application-specific decisions on how to handle missing and faulty data), and task that takes the output of the PI controller and turns it into a PWM signal (SEAL code in Listing 1). The PWM signal is used to control the intensity of heating, assuming a simple heater device that only has two states (*on*/*off*).

Listing 1: Implementation of the heater actuator task

```
1  // input is received from the PI controller task
2  input TaskGraph(heaterIntensity);
3  // define mapping from 16-bit range to 0%..100% range
4  define heaterIntensityPercent
5          map(heaterIntensity, 0, 0xffff, 0, 100);
6  // define the output pin and mode
7  define HeaterPWM AnalogOut, port 1, pin 2;
8  // read the last received value from cache,
9  // apply the mapping transform, and output result
10 read heaterIntensityPercent, period 15s, out HeaterPWM;
```

Then the user creates a new constraint describing the bound on the maximally acceptable delay, and puts it on the dataflow between the temperature sensor and the average calculator.

Subsequently, the user specifies *task frequency* properties. First he must partition the network in regions. Partitioning is done by writing a Python expression on node properties (these expressions are evaluated by the Python interpreter). In the example, the expression is simply `region`; however, in contrast to the original ATaG framework [2], more complex expressions are supported, including arithmetic and tuple expressions; for example, the pair `(building_number, room_number)` is a valid expression — given that both of these properties are defined on each node. Each of the (possibly many) partitioning expressions completely separates the network in non-overlapping regions; all nodes in each region have the same value of the evaluated expression.

Once the partitioning expression is written, it is named (for example, "rooms") and subsequently referenced by its name.

The user completes his design by specifying that all dataflows are region-local, and, for each task, the number of desired copies per region: three-per-region frequency for temperature tasks, one-per-region for all other tasks.

C. Task setup and maintenance

When all the abstract tasks have been mapped on the network model in a way that satisfies the user, he issues the *"Deploy"* command that creates runtime state for the instantiated tasks on the network nodes, which at this point are executing firmware images created in the node programming stage. This step is done both on nodes connected by a cable and wirelessly; it is automated by the tool. If desired, the complete model can also be tested in a simulation environment. In this way, the user can see whether the constraints hold in the simulator.

The following management command types are used: {*add*, *remove*} a task; *set parameters* for a task; {*add*, *remove*} a dataflow; {*add*, *remove*} a constraint; {*add*, *remove*} binding of a task on a node. The commands are sorted in classes to guarantee that, for example, a constraint condition on a dataflow is not evaluated before the dataflow itself has been set up. All commands from a single class must be acknowledged by destination nodes before commands from the next class may be sent. The order of the classes are: (1) commands that remove constraints; (2) commands that instantiate dataflows; (3) all other commands, except (4) commands that add constraints. Additionally, if an existing task is reallocated on a different node, an attempt to remove it from the old node is only made after an ACK has been received from the new node. As message loss is possible and probable in a WSN, all commands of the task setup protocol are designed to be idempotent: i.e., receiving and processing the same command more than once has no effects on the application functionality.

To assure that the application is working correctly, the ProFuN TG runtime middleware running on the nodes gathers application performance statistics and determines whether constraint conditions hold, enabling maintenance alert notifications, as well as automated maintenance through task remapping.

Alternatively, more computationally and data-intensive fault detection can be done on server-side hardware, by using either application-specific or third party tools, such as The Universal Translator (UT3) [9]. ProFuN TG provides an HTTP API through which these tools can: (1) continuously poll the gateway server instance for sensor data, and (2) send alert notifications directly to the front-end, requesting sensor blacklisting and task remapping in case a fault is detected.

IV. Architecture and components

The main software components of ProFuN TG are: (1) frontend web interface; (2) task allocator daemon; (3) gateway server; (4) runtime middleware; (5) WSN simulator. Under the hood, ProFuN TG uses a number of well-known software tools and libraries: Contiki for system-level functionality, Cooja [10] for network simulation and as a generic interface to platform-specific firmware compilers, Gecode [2] for constraint solving

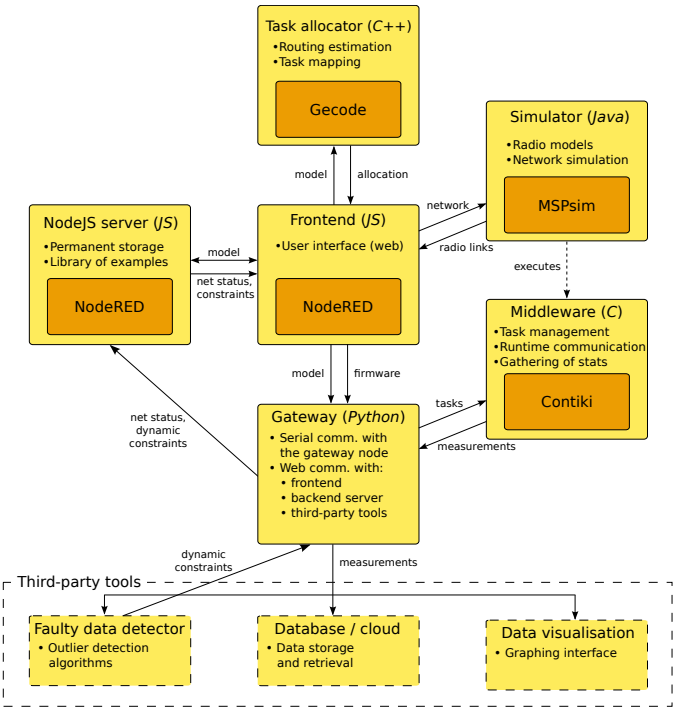

Fig. 4: Architectural overview

(used in the task allocation algorithm), and an adapted version of Node-RED [3] for the visual frontend.

ProFuN TG joins these components in a distributed microservice architecture (Fig. 4). The components communicate by passing asynchronous JSON requests through HTTP, with the exception of the WSN middleware, which uses an efficient binary data format. Each of the main components can be run on a separate computer as long as they can reach one another.

Since an HTTP message may time out at any point, the system must remain reliable under presence of lost messages. Furthermore, we want the system to be able to transparently handle component restarts. These properties is achieved by periodically (by default, every second) exchanging the whole model [4] between the components. It is feasible due to the compact JSON data format: for example, a 100 node network with 20 tasks is described by just a 12 KB large JSON file.

A. Automated task allocation

The task allocation takes place in two stages. First, the input model is validated and preprocessed by a Python daemon that evaluates the user-defined predicates and partitioning expressions. Then, the daemon executes a C++ application that takes the preprocessed model as its input. The role of this C++ application is to utilize the Gecode search API to find one of the globally optimal solutions to the task mapping problem; the details of the allocation algorithm are described in the accompanying technical report [5].

By default, the objective function that the allocation algorithm attempts to minimize is the total energy consumption for

[2] http://www.gecode.org

[3] http://nodered.org

[4] Documented in http://parapluu.github.io/profun/files/profun-schema.json

communication in the network. It is calculated as the sum of costs of all active dataflows. The cost of a dataflow is defined as the sum of the costs of all links the dataflow has to cross (according to pre-computed shortest path information) to get from the node on which the source task is mapped to the node on which the destination task is mapped. The shortest path algorithm uses an abstract metric supplied in the network model for each link; it may be simple hop count or ETX.

A few other predefined objective functions are available, such as: "minimize the maximal energy consumption for nodes in the network", and "maximize the least remaining energy after a user-defined time period". The users can supply their own objective functions by writing C++ code in the web interface.

B. Macrocompilation of node firmware images

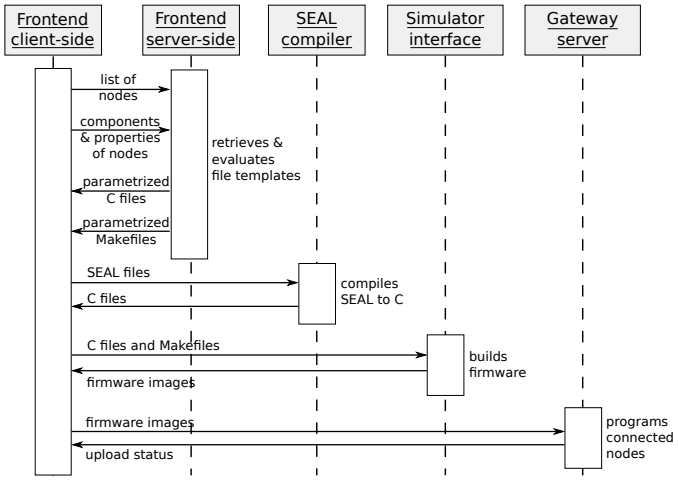

Fig. 5: The macroprogramming process in ProFuN TG

We use two-stage compilation (Fig. 5) to build the configuration-dependent firmware images of the WSN nodes. The first stage takes templatized C source files and C code build scripts (`Makefiles`) and evaluates them using Jinja2 templating library, replacing the template patterns in the files with variables supplied by the frontend. The result of this first stage is a `main.c` file and a `Makefile` for each type of firmware image. If SEAL functions are present in the task graph, their code is also compiled to C files in this stage.

The second stage is a regular compilation process that generates the hardware-specific executable images. This process is outsourced to the Cooja simulator, which is capable of building arbitrary applications given a build script and source files. Briefly put, it compiles and links together the generated C files, the ProFuN TG middleware library, and Contiki system code to create a distinct firmware image for each node type.

C. ProFuN TG middleware

The runtime middleware (see also [6]) is a C library built on top of Contiki OS. It manages the runtime state of the task graph and also includes hardware-specific implementations of predefined tasks. The initial implementation is tailored towards *msp430* MCU based sensor nodes.

There are three distinct traffic patterns in our system: data dissemination, data collection, and node-to-node traffic. The first pattern is used to set up tasks and other dynamic state on network nodes. The second pattern is used to send data and status messages from the network to the gateway node. The third pattern is used by application-specific task-to-task dataflows described by the task graph.

ProFuN TG uses a Glossy [11] based scheduler for the data dissemination and data collection traffic patterns, and Contiki Rime stack for the task-to-task traffic pattern. The scheduler is completely gateway-controlled. It has support for two phases: periodic schedule phase, in which all nodes can originate messages to gateway periodically, and target-specific traffic phase, in which only the gateway originates messages periodically, while nodes originate messages only if they have data to send and the gateway has explicitly scheduled them to do so. The periodic schedule phase is suitable for the initial setup of the task graph and for collection of alert and data messages coming from nodes *en masse*. The target-specific phase is suitable for making minor adjustments to the task graph, and is significantly more energy efficient. Setting up a task on a single node in this phase is much faster, as the node sends an ACK immediately, without waiting up to several seconds for its periodic schedule slot. However, when in this phase, nodes may have to use other protocols (such as *collect*) to send data to the gateway on their own initiative.

The middleware uses Contiki timing events to schedule execution of periodic tasks, and a custom Contiki process event to notify event-based tasks about incoming new data. The functions of tasks must run to completion; in other words, they are not interruptible by other tasks and are not allowed to voluntarily yield during execution. However, this limitation can be overcomed at the level of task implementations: task code is allowed to make use of WSN operating system services, such as threads or protothreads, to provide asynchronous interface to access long-running operations.

To detect violations of constraint conditions, the middleware keeps track of performance history for each constraint on each active dataflow at its destination node. The history is kept either as bit-buffer marking which packets have been received, or as a scalar EWMA (exponentially weighted moving average) value of past performance, depending on a configuration option. There is also a compile-time option to collect hop-by-hop performance statistics in the central system.

V. Evaluation of the runtime system

We attempt to answer two questions: (1) what is the overhead and timing characteristics of the centrally managed task setup mechanism, and (2) what are the gains from task reallocation in a network with unreliable and dynamically changing links?

We use trace based simulations of the test network (Fig. 2): with the help of the RealSim plugin [12] we replay packet traces from IEEE 802.15.4 channel 20 we recorded [5] in a typical working day (27th January 2015) when heavy WiFi activity was present in some regions during daytime and caused significant WSN link quality deterioration because of interference.

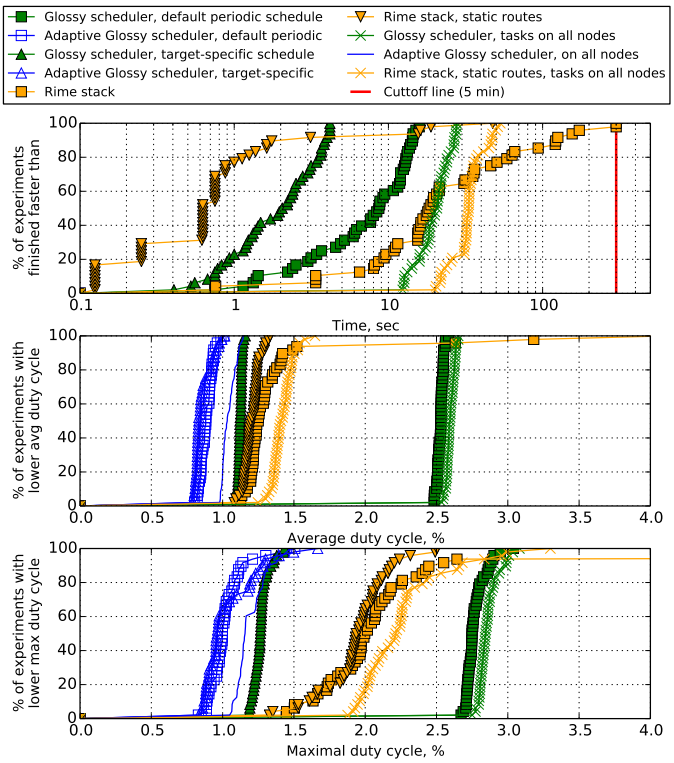

Fig. 6: Performance of the task management protocol

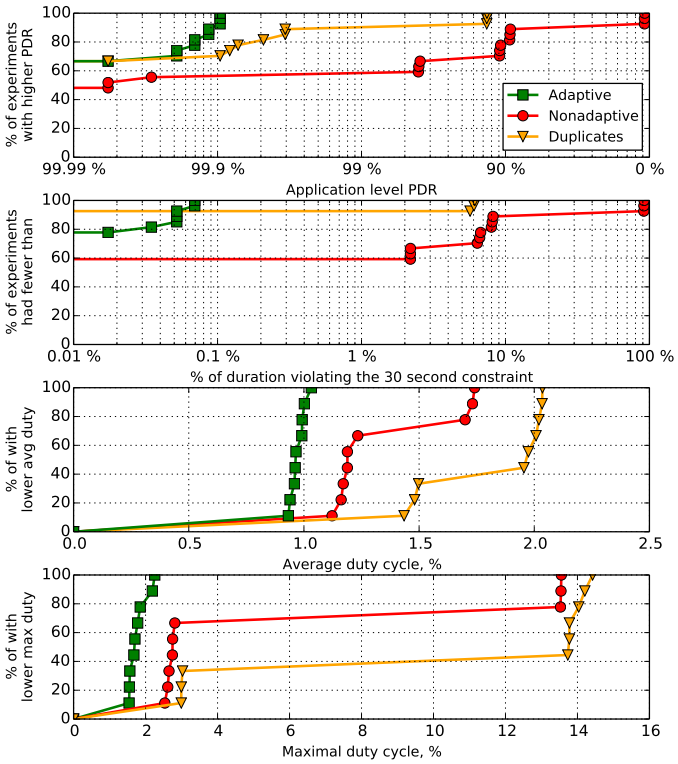

Fig. 7: The effect from adaptations via task reallocation

A. Task setup

We compare the time required to set up tasks in the test network (Fig. 2) when using the Rime *mesh*-based and a Glossy-based implementations of the management protocol. For the former, we compare the performance of two cases: (1) all nodes start with empty routing tables, and (2) static routes are pre-installed along the forwarding path. The second approach leads to higher performance, but it is not going to scale, as nodes do not have enough RAM to keep routing tables for large networks in memory. We use Glossy with 4-second rounds, each of which has 14 flooding slots: 6 for the gateway and 8 for a maximum of 4 nodes.

We perform two types of experiments: (1) set up a single task on a specific node, (2) set up a single, but unique task on *each* of nodes. We run each test for 5 minutes (using traces starting at 12:00, 27th January), and repeat each test 3 times for each node. All "create task" messages are continuously retransmitted by the gateway until they are ACKed by the endpoint. The end-to-end retransmission timer is set to 15 seconds; each of these end-to-end messages additionally are allowed to use up to 15 hop-to-hop retransmissions. Otherwise we use the Contiki default settings, and report the duty cycles of the whole 5-minute period.

Figure 6 shows that the performance of Glossy is more predictable: the maximum time required to set up a single task (4.2 sec and 16.0 sec) is smaller than that for Rime with static routes (47.5 sec), even though the average time is longer. Unlike Glossy, static routing would neither scale nor be able to handle changes in network topology. Furthermore, Glossy is also capable on setting up a task on *each* of the nodes in less than 30 seconds, something that Rime without static routes is not capable of doing within the 5 minute cutoff time (therefore not included in Fig. 6).

The benefits of Glossy come with a larger energy cost (average, not maximal!) unless the one-direction-only target-specific schedule is used. To counter this, we have also implemented an adaptive version in which the gateway turns off Glossy on nodes after the tasks have been set up; it demonstrates the best energy efficiency of all (Fig. 6); however, the current evaluation does not take into account the energy consumption required to adaptively *start* Glossy on all the nodes. It remains to be seen whether the cost of starting up Glossy when-needed in the whole network is small enough for this approach to be feasible in realistic applications.

B. Task reallocation for constraint satisfaction

We use the example application (Section III) mapped on the network similarly as shown in Figure 2. However, in contrast to the figure, we assume that temperature data is highly correlated within each region, so that receiving data just from one sensor node is sufficient for the actuator node. This models a scenario with redundant hardware nodes. We assume that to ensure the comfort of inhabitants the heater node requires T° data with delay no larger than 30 sec. To adaptively keep the application within this constraint, we instantiate a more strict (2000 ms) delay constraint in order to predict the performance of the 30-second constraint, as this approach gives good results [5].

We select three actuator nodes (105, 107, and 110 in Fig. 2), each having three potential sensor nodes as data sources. On

violation of the 2000 ms constraint, the actuator node sends an alert to the central system and resets the state of the constraint (to avoid excessive alert flooding). The central system then reallocates the source task to another candidate node; less frequently and less recently failing nodes are preferred.

We repeat the test 3 times for each of the nine candidate nodes, producing temperature data messages with 15 s interval, and present simulation results (Fig. 7) using packet traces from a 24-hour period (January 27th, 2015) for each test.

Without either task reallocation or duplication, only 60 % of the experiments show "good" results (do not violate the constraint for >1 % of time). With reallocation, the *maximal* violation is below 0.1 %. Non-adaptive task duplication on two nodes increases the probability that the constraint is satisfied; however, that technique is not sufficient in the 3 out of 27 experiments when *both* active sensor nodes happen to have bad links. In contrast, the adaptive approach achieves very high PDR (99.89 %) even in the worst case. Furthermore, it also demonstrates much better energy efficiency compared to the others, as the overhead for control traffic is negligible: 0.30 application-level end-to-end messages (not counting retransmissions) per hour per actuator on average, and 0.38 during the most erratic experiment.

We note that ProFuN TG is capable of making use of dynamically constructed link metrics to implement better and more sophisticated reallocation policies; however, these experiments show that the reallocation approach leads to application performance boost even when a very small amount of network state is transferred to the central system. Here it is just *a single bit* of information: the constraint failure status.

Arguably, the results are dependent on the existence of at least *one* good path from a sensor node and an actuator node in each region of the network. The redundant sensor nodes should be placed in a way that provides spatial diversity while still keeping the environmental variables of interest sufficiently correlated on all of them. If this setup is not possible, or the redundant nodes have not been installed, then ProFuN TG is not capable of delivering good performance; however, it still makes itself useful by delivering alert messages to the central system, where they can be observed by a human operator and used to make control decisions.

C. The performance of the frontend

The user interface as such is capable of handling large networks, as long as the task graph remains relatively simple. For example, if a single task is set up on each node, the time for full task allocation from sending a JSON data to the reception of the reply is 249 ms for a 100-node 1000-link network. (This and other results in this subsection are obtained on a desktop system with Intel Core i7 3.4 GHz CPU.)

For larger networks the time to calculate the shortest-path routing tables is the main bottleneck. We use the Floyd-Warshall algorithm, which has $O(|N|^3)$ complexity for calculating the routing tables (where N is the set of nodes). For a 1000-node network the whole process takes approximately one minute.

Specifically the visual interface is also capable of handling large networks. It redraws a 1000-node network (without any links) in 6.4 ms, and a 100-node 1000-link network in 60.4 ms.

VI. Related work

There is a large body of work on high-level programming for sensor networks; however, tolerance to failures has been noted as an open research issue [13].

We chose ATaG as the underlying formalism because it naturally allows to increase dependability of sensornet applications: at runtime, by remapping tasks to other nodes in case of failure, and at design time, by allowing the programmer to use redundant hardware nodes for additional copies of tasks.

`make`*Sense* [14] is a high-level WSN programming toolkit that includes dynamic runtime adaptation to application goals. The adaptations are relevant to specific parts of the system, and are based on performance annotations expressed by users in the application code. Dynamic information about the state of the network is collected in a central system, which then attempts to maximize an objective function defined on the network. However, `make`*Sense* does not use constraint solving, but instead relies on Monte Carlo reinforcement learning [14] through repeated simulations, therefore is a black-box approach in which integration of expert knowledge is not easily possible. Furthermore, it requires more extensive information about the network state to enable adaptations, while our solution in the typical case of periodic task-to-task traffic requires sending of only a single alert message from the network.

Srijan toolkit [4] is a graphical ATaG macroprogramming system. In contrast to ProFuN TG, it is missing the features introduced by the constraints: both the predictive aspect at design time and the diagnostic aspect at runtime. Unlike ProFuN TG, in Srijan it is not possible to express even simple dataflow instantiation rules, e.g. require that the endpoints are directly connected. Furthermore, in Srijan, tasks must be implemented in Java and require the presence of JVM at runtime.

Both Srijan and `make`*Sense* support the LogicalNeighborhoods (LN) [15] programming abstraction that includes support for data-dependent routing. In contrast, the dataflows and regions in ProFuN TG are determined by the central system, and routing is managed by the Contiki OS. LN both requires a more heavyweight runtime system and complicates the reasoning about the network state, therefore conflicts with our goals to support performance prediction and monitoring.

We took the general idea of user-defined probabilistic end-to-end constraints from Bijarbooneh *et al.* [16]. However, their model does not include probabilistic properties on network links, and their design-time constraint satisfaction checker cannot differentiate between single-hop and multihop dataflows.

As opposed to the original ATaG, we do not include Abstract Data Items in our task graphs, so our model loses the ability to visually specify more advanced relationship between tasks, such as the parent-child aggregation relationship. The gain from this simplification is a less cluttered visual layout: for example, the example application (Section III) is described by

just 9 visual elements (5 tasks and 4 dataflows) in ProFuN TG, but would require additional 4 data items and 4 dataflows if the original ATaG was used. In contrast to the lost functionality, this gain is relevant to all task graphs with any connected tasks.

The user interface of the task graph design view is adapted from Node-RED. However, Node-RED does not include predictive or adaptive aspects. Also, its runtime requires a JavaScript interpreter, therefore is not feasible on sensor nodes.

Redundancy in conjunction with a centralized control plane is heavily exploited by the two competing standards in the area of WSN for industrial monitoring and automation: WirelessHART [17] and ISA100.11a [18]. However, these systems do not have the concept of a *task*, therefore they are not able to offer automated application-level task reallocation and duplication. Furthermore, they are much more heavyweight, while our runtime system uses a reactively-adaptive approach and therefore does not require collection of extensive network state in a central controller.

There are also deployment and experimentation support systems such as DREAMS [19] and MakeSense [20], and runtime assurance systems such as the ones developed by Wu *et al.* [21] and Fairbairn *et al.* [3], but they all lack the abstraction of a *task*, therefore do not enable performance adaptations through task reallocation.

VII. Concluding remarks

ProFuN TG joins together support for high-level programming using ATaG with performance prediction and runtime assurance capabilities. It achieves that by allowing the user to write PDR and delay constraints on dataflows between tasks. The tool enables deployment and maintenance of WSN applications by providing a middleware that manages the runtime state of tasks and constraint conditions, and triggers reallocation in case a constraint violation is detected.

Our trace-based evaluation using data from a 17-node interference-exposed test network shows that the task setup protocol instantiates runtime tasks on all nodes within 30 seconds, making the reallocation approach feasible in hard-to-predict, erratic real-world environments, and that our adaptive task reallocation approach improves application-level performance in presence of unreliable links by exploiting node redundancy.

Acknowledgments

The authors acknowledge support from SSF, the Swedish Foundation for Strategic Research.

References

[1] A. Bakshi, V. Prasanna, J. Reich, and D. Larner, "The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems," in *USENIX EESR*, 2005, pp. 19–24.

[2] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco, "A compilation framework for macroprogramming networked sensors," in *IEEE DCOSS*. Springer, 2007, pp. 189–204.

[3] Y. Wu, K. Kapitanova, J. Li, J. A. Stankovic, S. H. Son, and K. Whitehouse, "Run time assurance of application-level requirements in wireless sensor networks," in *ACM/IEEE IPSN*, 2010, pp. 197–208.

[4] A. Pathak, Q. Zhou, and V. Prasanna, "Srijan: A graphical toolkit for wsn application development," in *IEEE DCOSS*, 2008, pp. 34–39.

[5] A. Elsts and K. Sagonas, "ProFuN TG: A Tool for Programming and Managing Dependable Sensor Network Applications," Technical Report. [Online]. Available: http://www.it.uu.se/research/profun/tools/tg-2015.pdf

[6] A. Elsts, F. H. Bijarbooneh, M. Jacobsson, and K. Sagonas, "Enabling design of performance-controlled sensor network applications through task allocation and reallocation," in *7th International Workshop on Performance Control in Wireless Sensor Networks*, 2015.

[7] A. Elsts, J. Judvaitis, and L. Selavo, "SEAL: a Domain-Specific Language for Novice Wireless Sensor Network Programmers," in *EUROMICRO SEAA*, 2013, pp. 220–227.

[8] K. J. Aström and T. Hägglund, *PID controllers: theory, design and tuning*, 2nd ed. International Society for Measurement and Control, 1995, ISBN:1556175167.

[9] P. Haves, "The universal translator - a freeware platform for third party diagnostic algorithms." [Online]. Available: http://utonline.org

[10] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with Cooja," in *IEEE LCN*, 2006.

[11] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in *ACM/IEEE IPSN*, 2011, pp. 73–84.

[12] M. Strübe, F. Lukas, B. Li, and R. Kapitza, "DrySim: simulation-aided deployment-specific tailoring of mote-class WSN software," in *ACM MSWiM*, 2014, pp. 3–11.

[13] L. Mottola and G. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011.

[14] F. Casati, F. Daniel, G. Dantchev *et al.*, "Towards business processes orchestrating the physical enterprise with wireless sensor networks," in *IEEE ICSE*, 2012, pp. 1357–1360.

[15] L. Mottola and G. Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *IEEE DCOSS*, 2006.

[16] F. H. Bijarbooneh, A. Pathak, J. Pearson, V. Issarny, and B. Jonsson, "A constraint programming approach for managing end-to-end requirements in sensor network macroprogramming," in *SENSORNETS*, 2014.

[17] D. Chen, M. Nixon, and A. Mok, *WirelessHART(TM): Real-Time Mesh Network for Industrial Automation*. Springer, 2010, ISBN: 1441960465.

[18] "ISA-100 Wireless Compliance Institute," http://goo.gl/5nySj.

[19] R. Figura, M. Ceriotti, C.-Y. Shih, M. Mulero-Pâzmâny *et al.*, "Iris: Efficient visualization, data analysis and experiment management for wireless sensor networks," *EAI Endorsed Transactions on Ubiquitous Environments*, vol. 14, no. 3, 11 2014.

[20] R. Leone, J. Leguay, P. Medagliani, C. Chaudet *et al.*, "MakeSense: Managing Reproducible WSNs Experiments," *RealWSN*, 2013.

[21] M. L. Fairbairn, I. Bate, and J. A. Stankovic, "Improving the dependability of sensornets," in *IEEE DCOSS*, 2013, pp. 274–282.