Technical Report

ProFuN TG:
A Tool for Programming and Managing Dependable Sensor Network Applications

Atis Elsts and Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden

Abstract—Sensor network macroprogramming methodologies such as the Abstract Task Graph (ATaG) 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 development support for dependable application programming.

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 macro-compilation, application simulation, deployment, and runtime maintenance. It also allows users to incorporate quality-of-service requirements in the applications, expressed through constraints on task-to-task data flows. For the design stage, we show how ProFuN TG can be used to efficiently and optimally map tasks on networks while satisfying user-defined constraints. For runtime, we provide an efficient flooding-based protocol to set up tasks in the network, and a middleware that continuously checks whether the conditions of the constraints are satisfied and requests task remapping on failure.

I. INTRODUCTION

The currently dominating system-level approach to wireless sensor network (WSN) software engineering does not provide ready-to-use tools for implementing functionality commonly required by WSN users, such as:

• Create a tailored firmware image for each sensor node in the network depending on its hardware components and software configuration.

• Using node configuration properties, partition the model of the network in logical regions.

• Set up specific application-level tasks on sensor nodes; control and change them during the network’s lifetime.

• Given the network model, assumptions about its environment, and an application with specific quality-of-service (QoS) requirements, determine the nodes on which the application should be deployed so that the requirements hold.

• Continuously assure the user that the application is still meeting its QoS requirements.

Such functionality is instead implemented on application-specific basis; an approach that is both tedious and error prone.

We take an existing WSN programming methodology, the Abstract Task Graph (ATaG) [1], and implement it in ProFuN TG[2], 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 extends ATaG by allowing to incorporate end-to-end reliability requirements [2] in that description. The requirements are expressed in form of constraints on packet delivery rate (PDR) and delay, and are set on dataflows between tasks.

The tool includes support for mapping these task graphs on network nodes, for macrocompilation of their code, and for their deployment both in simulated and real networks.

We implement a task allocation algorithm that efficiently and optimally maps task graphs on networks while keeping the mapping in bounds of user-defined constraints.

For the runtime, we provide an efficient flooding-based protocol that sets up tasks in the network. The supporting run-time middleware (implemented for msp430 MCU based sensor nodes) manages task-to-task communication, gathers application performance statistics and determines whether the conditions of the constraints hold, enabling run-time assurance. On failures, maintenance alert notifications are issued, and automated maintenance through task remapping is performed.

ProFuN TG is customizable: both user-defined tasks and user-defined objective functions for task mapping are allowed.

The rest of this technical report includes a conceptual background (Section II); a high-level description of the tool (Section III); its architectural overview and a brief description of its components (Section IV); an evaluation of the components (Section V), and a comparison with related work (Section VI).

II. CONCEPTUAL FOUNDATIONS

A. Programming model

We adopt the Abstract Task Graph (ATaG) [1] macroprogramming model, which builds on the dataflow programming paradigm. The core concept of this programming model is the task graph (Fig. 1a), a user-defined graph where the vertices correspond to abstract tasks and edges denote dataflows between these tasks.

[http://parapluu.github.io/profun/](http://parapluu.github.io/profun/)
An abstract task is a chunk of functionality with a fixed interface, such as 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 the firing rule (periodic or event-based), firing period, and the number of task 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 constraint properties that bound the acceptable PDR and packet delay values on this flow (Section II-C), number of maximal retransmissions property, 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 is textual and imperative. ProFuN TG task code is written in C.

ProFuN TG provides a number of predefined task types in several categories: sensors, actuators, processing tasks, and other data I/O. Sensors typically produce data, actuators consume data, and processing tasks take one or more input data items and convert them to one or more output data items.

While users are free to introduce their own types of tasks of any category by extending the tool, the predefined function task type is unique in the sense that it can be used implement functions specific to a single task graph using just the visual interface. Users have to provide the name of the function, its properties (such as number of inputs and outputs), and its code. The code of a function task consists of several separate blocks, possibly empty, instantiated as separate C language functions:

- initialization;
- periodic actions;
- input data item received;
- cleanup.

The initialization function is executed when the task is created. The periodic function is executed only for tasks with a periodic firing rule; its frequency is determined by another property of the task. The input data item received function is executed when the task receives data from some other task, and cleanup is executed on termination. All types of tasks, including the predefined ones, share this division of runtime functionality.

The task functions must run to completion; in other words, they are not interruptible by other tasks, and are not allowed to voluntarily yield during execution. This limitation may seem to hinder implementation of asynchronous operations. However, there is a workaround that takes just a few lines of wrapper code. The idea is to start a thread (a protothread in Contiki OS) from a task’s initialization function and return immediately. The periodic function should either poll that thread to produce data items, or the thread can schedule itself to wake up when needed, and output data using ProFuN TG API. Finally, the task’s cleanup function should destroy that thread.

### B. Network model

ProFuN TG allows the user to interactively create and refine a model of the network and its environment (Fig. 1b).

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. 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. Examples
of such properties include transmission success probability and delay value. We do not restrict the descriptions of 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, recall that packet delivery failures on wireless links are caused by several different reasons \cite{3}. Some of these causes, such as interference, lead to bursty packet losses, while others, such as a signal fading on weak links, do not exhibit this property as strongly. Consider two imperfect network links, one of which is interfered, while the other one suffers from signal fading. Assuming they have the same average PDR, the first one is likely to have far higher variance. Now assume 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 such a constraint.

C. Constraints

One of the key features of ProFuN TG is the support for user defined, end-to-end constraints between source and destination tasks. These constraints serve two roles:

- Predictive: the task allocation algorithm takes the constraints into account and will not produce mappings that violates them.
- Diagnostic: the runtime system continuously tests whether constraint conditions are met. In case this test fails on a node, that node notifies the central system, which then re-allocates the tasks.

For each constraint, the user is allowed to specify the minimal acceptable probability $P$ with which it is predicted to hold in the task-mapping stage. For example, let us take $P = 0.98$:

$$P(\text{Delay} < 3000 \text{ ms}) \geq 0.98$$

$$P(\text{PDR} > 90 \%) \geq 0.98$$

What does the probability $P$ represent at runtime? There are at least two possible answers. The first is that the user is willing to tolerate some violations at runtime, as long as their proportional frequency is not higher than $1 - P$. The second is that $P$ represents just the subjective uncertainty about the model; for the runtime, the user wants guarantees that all communication will be within the bounds of the constraints, irrespective of the probabilities in the model.

The second interpretation leads to a simpler runtime check, but in some cases it is too restrictive. For example, the user might not want to remap the source task (and possibly other tasks) just because a single packet failed to arrive within the expected delay bounds. Therefore ProFuN TG offers to select one of the two interpretations as a configuration option. For the first interpretation, statistically significant run-time tests are only possible after certain amount of data has been gathered. This minimal-number-of-values is another user-configurable parameter.

III. ProFuN TG Functionality

Consider an example application: an indoor heating control system extended with fire detection functionality (Fig. 1). This application has two sensing tasks: temperature and smoke, two actuation tasks: heater and alarm, and a data processing task: threshold operation. Fire is detected when either a smoke sensor is activated, or when temperature in a room exceeds a predefined threshold. The action taken by the system on fire is abstracted by the alarm task.

We assume the application is deployed in a building with several 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 equipped with actuators;
- each room in the building should have at least one active heater task;
- each heater task should receive input from at least two temperature tasks located in the same room;
- each alarm task should receive data from a smoke sensor in the same room, with delay smaller than 30 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 programming, task graph design & mapping, and task setup & maintenance. In the node programming stage firmware images are created.
and programmed on network nodes. In the task graph design and mapping stage application-level functionality is first described by the user, and then mapped on a model of the network by the tool. In the task setup and maintenance stage application-level functionality is introduced in the network, and is reactively remapped in case the execution environment or the physical environment changes.

The tool is designed for and supports an iterative development process (Fig. 2). For example, observing the operation of a real network may 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 programming

The programmer of WSN applications is faced with the problem of configuration complexity. In a set of nodes that all run the same application-level code many different configurations may be desired, for example:

- System-level and network-level parameters (for example, whether to retransmit packets on behalf of other nodes).
- Node location, network address or ID number.
- Per-node software correction settings for biased sensors.
- Per-region application level settings, such as the by-default desired temperature in the room in which the sensor node is located.
- Hardware components present on the node. Manual configuration is typically required, as trying to initialize or use a non-existing hardware component may lead to suboptimal energy usage or even a node reboot, and autodiscovery of components is impossible without hardware support.

One solution is turn on or off functionality of a node depending on its static network address or serial ID number. The problem with this approach is that on a larger number of configurations the ID->configuration mapping becomes hard to track, and laborious to modify. Clearly this approach does not scale. Furthermore, coexistence of nodes with different hardware platforms in a single network is often desired; each platform typically requires a distinct compilation process to produce its firmware.

ProFuN TG helps by automating this process by allowing the user to describe the platform of each network node (from a pre-defined palette), and then to configure the nodes — to specify their hardware components, their constant properties (included in the C source files from which the firmware is built as C preprocessor directives), 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. The image is not required to be unique — if several nodes share the same configuration, they all have identical firmwares, so running a separate compilation process is not required for each.

Finally, the produced firmware images are automatically deployed on sensor nodes. At the moment only wired programming is supported; however, there are no conceptual obstacles against extending our approach with over-the-air reprogramming.

In the example application, the user starts by defining the templates for sensing nodes and actuating nodes. The description of these templates contain a list of hardware component drivers to be included in the firmware, and may also contain properties common to all or most of nodes, such as the identifier of the building. The user defines constant properties on each node, for example, room identifiers (a scalar property room_number), and also variables, such as the desired temperature in a room. (Managing the runtime state of these variables is not a part of ProFuN TG.)

B. Task graph design and mapping

The next difficulty of WSN application programming is to divide and setup the desired application-level functionality on all the nodes. At first, this task seems relatively straightforward to perform even manually: the user first designates a few different roles of nodes, for example, sensing nodes, data processing nodes, actuating nodes, and then maps the functionality of the application on each node depending on its role.

However, the situation is complicated when specific relations are desired. For instance, in our example application, the user may want at least two distinct temperature sensor nodes sending data to each heater node. Such a wish for redundant measurement sources is not atypical among WSN users; some common reasons are: (1) to provide the data consumers with a balanced view of the situation; (2) to allow to detect and ignore measurement errors; (3) to increase the probability that at least one of the sources successfully delivers the data; (4) to support applications that explicitly require redundant data, e.g. data collection for scientific studies.

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 the fact that these relations are met everywhere in the network.

Furthermore, it allows to map tasks only to nodes with a specific configuration. The user can write a binary predicate for a task (i.e., a logical expression on node properties) which is evaluated at design-time and functions as a filter on the set of nodes eligible to host the task.

The situation gets complicated even further if certain task-to-task reliability guarantees must be made between tasks, especially if they cannot be mapped on the same node. It is usually not possible to reduce these guarantees 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 the user-defined network model in order to automatically determine optimal mappings of task pairs that are within bounds of these constraints.

The network model used by ProFuN TG supports random variables, such as delay and PDR, defined on each link of the network. The variables are described by probability distribution functions. Both sums and mixtures of arbitrary
distributions are supported. For example, a network link may have a delay distribution that is sufficiently well approximated by a single sharp peak at 0.5 sec with 60\% probability and a log-normally distributed tail of larger delays with 40\% probability and parameters $\mu = 0.6, \sigma = 1$. ProFuN TG supports the following syntax for writing down this example mixture distribution:

\[
\text{Normal}(0.5, 0.0); 0.6, \text{LogNormal}(0.6, 1); 0.4
\]

Given the probability distribution functions (PDF) of individual links, the task allocator estimates the cumulative distribution functions (CDF) between each pairs of nodes in the network. Given a particular constraint, it uses the path CDF to check which mappings satisfy the condition of the constraint (Fig. 3).

![Fig. 3: Probabilistic constraints on a log-normal delay distribution. Given minimal probability $P = 0.9$, constraints with delay value of $C = 4s$ are not satisfiable, while constraints with delay value $C = 4.5s$ are satisfiable (in the model)](image)

As an example, consider a maximal delay constraint. The constraint has two user-configured parameters: delay bound $C$ and minimal probability $P$. A path from source node $s$ to destination node $d$ satisfies the constraint iff:

\[
P(\text{Delay}_{path(s,d)} < C) \geq P
\]

By definition,

\[
P(\text{Delay}_{path(s,d)} < C) = \text{CDF}_{path(s,d)}(C)
\]

\[
\text{CDF}_{path(s,d)}(C) = \int_{-\infty}^{C} \text{PDF}_{path(s,d)}(t)dt
\]

Delay of a packet in sensor networks is heavily dependent on the number and quality of the links it has to cross. Therefore the path delay distribution can be approximated as the sum of its link delay distributions:

\[
\text{PDF}_{path(s,d)} = \sum_{\text{link} \in \text{path}(s,d)} \text{PDF}_{\text{link}}
\]

where the sum is calculated as convolution of the link PDF functions.

Therefore to determine whether the path from $s$ to $d$ supports dataflows with the specific constraint, ProFuN TG evaluates:

\[
\text{CDF}_{path(s,d)}(C) = \int_{-\infty}^{C} \sum_{\text{link} \in \text{path}(s,d)} \text{PDF}_{\text{link}}(t)dt
\]

Since in the general case the integral cannot be solved analytically, ProFuN TG uses numerical integration by Monte Carlo sampling to approximate its value.

In the case when the number of retransmissions is not infinite, there is a non-zero probability that the packet is never delivered. To handle this case, the user should include an additional mixture term in each link delay distribution. The term can be a single point with a very large value. For example, if the “infinity” floating point value defined by IEEE 754 standard is used for this purpose, the path PDF is guaranteed to be above any finite constant $C$ if a least one of the links drops the packet.

Once the set of the 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 starts by creating a task graph for the application (Fig. 1). Then he creates a new constraint, describing the bound on the maximally acceptable delay, and puts it on the relevant dataflows, such as the dataflow between the smoke sensor and alarm actuator.

Then the user specifies task frequency properties. To fulfill the requirements, he first must partition the network in regions — in this case, rooms. Partitioning is done by writing an expression on node properties. These expressions must be in Python syntax, as they are evaluated by the Python interpreter.

In the example, the expression is simply `room_number`, but more complex expressions are supported; they are not even required to take scalar values; for example, the pair `(building_number, room_number)` is a valid expression — given that both of these properties are defined on each node. Each partitioning expression completely separates the network in non-overlapping regions, so that all nodes in each region have the same value of the evaluated expression.

Once the expression is written, it is named (for example, “rooms”) and further 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: two-per-room frequency for temperature tasks, one-per-room for all other tasks.

If a task requires specific hardware components or software configuration of a node, the user writes a binary predicate (in Python syntax) specifying these requirements. This predicate is evaluated on properties of each node, similarly to the partitioning expressions.

C. Task setup and maintenance

When all of the abstract tasks have been the mapped on the network model in a way that satisfies the user, he issues the “Deploy” command, which takes the mapped tasks one-by-one and instantiates them on network nodes that at this point are executing firmware images created in the first 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 executed for some time in a simulation environment. In this way, the user can see whether the constraints hold in the simulator. The rest of this section also applies to simulated networks.

Task setup commands are generated by the gateway server, which then sends them to the gateway WSN node, which then forwards them to the WSN. The following 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. Dataflows are instantiated before tasks are created in order to optimize performance in the case when a flooding protocol is used to exchange the commands. In that case, explicit task binding messages can be skipped, as the whole model is received by each node in the network. However, keeping the whole model in the memory typically is not feasible, so we allow a node to store only those bindings that are relevant, i.e. only for tasks to which there are some dataflows originated in the 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.

Perhaps the biggest difficulty is to keep the network working correctly. Sensor networks suffer from aggressive and changing environmental conditions, software bugs, and hardware faults. Functionality-wise these faults can be grouped in three classes:

1) Sensor-level faults: sensor-reading functions returning error values or invalid values.
2) Node-level faults: nodes rebooting, losing communication capabilities, or completely turning off, typically caused by insufficient remaining energy.
3) Link-level faults: links with no or severely degraded capability for communicating data, typically caused either by interference or by radio signal fading.

While WSN operating systems can handle link and node failures through dynamic rerouting, they are unable to provide application-level end-to-end QoS guarantees because they lack application-specific information. Moreover, sensor level faults are highly-application specific, and, even supposing the system can detect the fault in a node-level code, it is typically not able to do anything about it other than returning an error code when an application tries to read the sensor.

In contrast, the supporting run-time middleware of ProFuN TG gathers application performance statistics and determines whether the conditions of the constraints hold, enabling maintenance alert notifications, as well as automated maintenance through task remapping.

In this way, link-level faults are detected and handled. Since a node-level fault implies effects similar to a link-level faults (no network traffic on dataflows routed through this node), we do not handle them separately. The performance prediction algorithm at the task-mapping stage also does not take into account node-level faults for now; this is motivated by the observations that in WSN node-level faults are much more rare than link faults, and that the node-to-node variance in a WSN is much smaller than the link-to-link variance, at least in homogeneous networks under normal conditions.

Since sensor-level faults are application specific, we do not offer any functionality to detect them at the middleware level. Applications can detect them instead inside the application-specific task code. The middleware API then can be used to send an alert message to the base station requesting that the offending sensor on that specific node is marked as bad, and the task is remapped. However, not all faults can be detected at WSN node level. More computationally and data-intensive fault detection can be done on server-side hardware, by using either application-specific or third party tools. 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.

D. Discussion

Inferring the probability distributions from data is a well-known topic in statistics and machine learning still under active research. In our case, standard statistical methods can be used, such as the expectation-maximization algorithm \[3\] to find the maximum likelihood mixture of distributions for each link based on its past performance data. The inference algorithm is not included in the tool itself, as choosing the right parameters for inference algorithms and interpreting the results may require significant domain expertise and manual tweaking. It would be naïve to try to propose a general-purpose algorithm for inference that does not take into account the specifics of different applications, networks, environments, and different user requirements.

Instead, we offer a fairly general model format that can incorporate both network-level properties (such as Tx success rate) and higher-level properties (such as the delay distribution) for each link\[4\].

An important aspect specific to wireless communication is the time-varying nature of link QoS. Often the changes are periodic and, to some extent, predictable, especially for outdoor WSNs \[5\]. Since the user who sets QoS constraints is mostly interested in the worst-case performance, we propose that the worst-period data on a link should be used to construct

\[2\]The format is documented in [http://parapluu.github.io/profun/files/profun-schema.json](http://parapluu.github.io/profun/files/profun-schema.json)
the link’s delay and PDR distributions. Finding the optimal window size (period length from which to take the data for constructing the distributions) is in our list of immediate future work.

IV. ARCHITECTURE AND COMPONENTS

![Fig. 4: Architectural overview](http://www.gecode.org)

Under the hood, ProFuN TG uses a number of well-known software tools and libraries: Contiki for system-level functionality, Cooja for network simulation and as a generic interface to platform-specific firmware compilers, Gecode for constraint solving (used in the task allocation algorithm). For the visual frontend, an adapted version of Node-RED is employed.

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. The distributed nature of the tool means that each of the four main components (frontend web server, task allocator daemon, gateway server, and network simulator) can be run on a separate computer as long as they are in the same network, or, alternatively, are connected to the Internet.

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

The tool provides an HTTP interface for data export in JSON format. Through that interface, third-party tools can access the data, as well as impose dynamic constraints on the task mapping algorithm. This enables integration with the plethora of existing fault detection tools and software libraries.

To provide a working example, we demonstrate how to apply a Principal Component Analysis based fault detection algorithm that automatically infers acceptable covariance bounds from a historical data training set and is able to recognize faulty data by detecting abnormal variance using these bounds. More generally, platforms such as The Universal Translator (UT3) can be used an a adapter to commercial and custom Fault Detection and Diagnostic (FDD) tools.

A. Frontend

The frontend consists of a web interface implemented in HTML5 and JavaScript, and a nodejs web server written in JavaScript.

The web interface allows the user to design the application in the task graph view (Fig. 4a). The user starts the design process by dragging task templates from palette in the left side of the view on the main editor window. After that, he is free to edit the properties of each task — for example, specify its firing period —, and to link the tasks together by dataflows.

In a similar fashion the interface allows to create and edit the network model: nodes and communication links between them. Finally, it allows to visually observe how the task graph is mapped on the network by the task allocation algorithm (Fig. 4b), as well as to trigger the allocation manually by issuing the “Allocate” command.

The web server stores the persistent state of the complete model. The user loads this state when connecting to the server, and can overwrite it with an updated version that includes his edits by issuing the “Save” command. Collaborative editing is not possible, but collaborative viewing of the dynamic state of the network model is, in principle, supported by this architecture.

The frontend also functions as a communication hub for the rest of the non-WSN side of the system, as shown in (Fig. 4).

B. The task allocator

The task allocation takes place in two stages (Fig. 5).

First, the input model is validated and preprocessed by a Python daemon. Then, the daemon executes a C++ application using the preprocessed model as an 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 role of the preprocessor is to evaluate the predicates and partitioning expressions. If a task has a user-defined predicate specified in its properties, the predicate is evaluated on each node. If it fails on a node n, then n is added to the list of the nodes on which the task cannot be mapped.
The evaluation (Section V-A) shows, no algorithm or strategy consistently dominates all others on every problem instance. On the other hand, it also shows that the fastest approach is often at least an order of magnitude faster than others, even the second best.

The pseudocode of the subsequent process is given in Algorithm 1.

**Algorithm 1.**

```plaintext
The C++ application (for example, because of an error in user’s supplied objective function) do not destabilize the whole system, as the crashed C++ application is simply terminated, after which the Python daemon logs and reports a task allocation failure.

In future work, we also plan to extend the approach by starting multiple search algorithms with different strategies in parallel as multiple C++ application instances. As the evaluation (Section V-A) shows, no algorithm or strategy consistently dominates all others on every problem instance. On the other hand, it also shows that the fastest approach is often at least an order of magnitude faster than others, even the second best.

**Fig. 5: Process sequence for mapping the tasks to nodes**

**Fig. 6: Process sequence for programming the nodes**

**Fig. 7: Process sequence for setting up a new instance of a task in the network**
network. It is calculated as the sum of costs of all dataflows in the network. (This particular objective function assumes that task invocation costs are the same on all nodes, so they make a constant term that can be factored out.) 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 also supply their own objective functions by writing their code in the interface.

Using Gecode API, we prune the search tree as early as possible to avoid fully generating mappings that cannot be better than the best mapping already found. The consequence is that the search algorithm should be designed to start with mapping the part of the task graph that is least likely to change.

Therefore relatively good performance is expected if the algorithm starts the mapping process with the most central, highly connected and heavily constrained tasks. However, there are different notions of centrality, and different centrality measures of a node in a graph, such as: degree of the node, closeness, etc. A promising measure in our case is eigenvalue centrality; using it to order branch selection leads to much better results than random ordering (Section V-A).

For a vertex (task) $x$ eigenvalue centrality $x_v$ in graph $G$ is defined as:

$$x_v = \frac{1}{\lambda} \sum_{v \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t$$

where $A$ is the adjacency matrix of the task graph, $M(v)$ is the set of neighbors of $v$, and $\lambda$ is the greatest eigenvalue of $A$:

$$Ax = \lambda x$$

The values $x_v$ for all tasks are calculated during the preprocessing stage of the task allocation algorithm, using iterative approximation.

The mapping order of the allocator is determined by sorting tasks in decreasing order by merit value: merit$_v = x_v + a \cdot c_v$, where $c_v$ is the number of constraints on task $v$, and $a$ is a constant.

We implemented the BranchingOrder function that returns this mapping order; the result of this function is used internally by Gecode.

C. Interface to the network simulator

The interface is implemented as a plugin for Cooja [9]. The frontend periodically sends (by default, once per second) a network model and options to the plugin, which then constructs a network in Cooja corresponding to that model. The options include details such as transmission success probability, maximal transmission distance, and which one of the predefined Cooja radio models to use. The potential radio links and their quality indicators are estimated by Cooja in real time; this information is collected and sent back to the frontend.

The second function of this interface is to control the execution of the task graph application in the simulated network. Execution is started and stopped depending on user’s commands. The gateway server accesses the simulated network in the same way a physical network is accessed. We use the Serial2Pty plugin to construct a virtual serial port for the gateway node, to which the gateway server then connects.
D. Macrocompiler

We use two-stage compilation (Fig. 6). The first stage takes templates of C source files and C code build scripts (Makefiles) and evaluates them using Jinja2 templating library, replacing 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.

The second stage is the regular compilation process that generates 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, this stage compiles and links together the just-generated main.c file, the ProFuN TG middleware library, and Contiki system code to create a different firmware image for each type of node.

Hardware components are included and excluded from the firmware images by using build script commands. Each platform provides a sensible set of hardware components enabled by default. Node properties specified in the visual interface are included in the compilation environment and seen by all source files a C preprocessor directives.

The users of ProFuN TG are not limited by the software components available out of the box. For one, they can write their own components. For another, they can use one of the two function components on the palette. Through these components, flow-specific functionality can be implemented.

Functions can be composed directly in C, but they can also be programmed in SEAL [10], a WSN-specific node-level programming language. SEAL comes with its own middleware library whose functionality is partly orthogonal to that of ProFuN TG middleware. For example, it implements custom scheduling mechanisms on top of Contiki timers and protothreads. It also implements commonly required functions, such as logical and arithmetical operations, data filtering and aggregation functions, etc.

Programming tasks in SEAL has a number of benefits:

- a task is not required to be partitioned in three distinct functions;
- writing a task does not require knowledge of the middleware C API;
- advanced functionality (e.g. periodic self-scheduling of the task) is comparatively much simpler to implement;
- the code cannot cause the whole system to misbehave or even reboot because of an accidental infinite loop in the code: it is impossible to write non-interruptible loops in SEAL.

Thus the AtaG architecture allows to join tasks written in multiple programming languages in a single task graph. We are also evaluating other programming languages for programming the functions, such as Ceu [11], another node-level programming language that is compiled to C code.

E. Gateway server

The gateway server is a Python daemon that acts as interface between the sensor network and the rest of the system. The server is intended to run on border routers, for example, in our testbed deployment (Section V-D) we put them on Raspberry Pi nodes. Each active gateway server instance has 1:1 correspondence with a WSN node connected through the serial port.

When the user issues the “Deploy” command, the frontend sends the complete model to the gateway server instance. The model in this case includes the task mapping. The server, after successfully receiving this model, compares it with the last one received and determines the changes between them. For the first received model, an empty previous model is assumed. From these changes, an ordered list of commands is created (see Section III-C). Each command also includes the address of the node on which it must be processed. The commands are, one by one, formatted as binary messages and sent to the connected WSN node (Fig. 7). Each binary message includes a single-byte checksum to ensure its integrity. Each command may be retransmitted by the gateway server several times until an ACK is received.

Each command is processed by the connected node. Commands with bad CRC are rejected. Otherwise, in case the command is addressed to the node itself, a reply ACK is sent immediately. In case it is addressed to some other node in the network, the command is forwarded to target node, possibly multiple times, until a network ACK is received from the target. After that, the directly connected node responds with a serial ACK to the gateway server.

The gateway WSN node also receives data messages and status messages, such as constraint condition violation alerts, and forwards them to the server. Status messages describe the dynamic state of the network model; they are forwarded to the frontend server. Data are temporary kept in a RAM cache and also optionally logged in a file. Custom or third party tools can access the data in the RAM by issuing an HTTP JSON request.

F. Middleware

The middleware is a C library built on top of Contiki OS. It manages the runtime state of the task graph and also includes msp430-specific implementations of predefined tasks.

The middleware initially used Contiki Rime protocol stack. There are three distinct traffic patterns it handles:

- gateway to nodes (mesh protocol);
- nodes to gateway (collect protocol);
- task to task (mesh protocol, easily replaceable with application-specific protocols).

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 dataflows described by the task graph.

We extended the Rime mesh protocol by adding reliability support in form of retransmissions. Nevertheless, we discovered that the mesh protocol suffers from severe scalability problems for gateway-to-node traffic. Setting up a task on a remote node requires reliable end-to-end transport for the task.
message, and then also for the end-to-end acknowledgement. Setting up another task on a neighboring node requires almost completely repeating the process. The CSMA-based MAC protocol leads to severely reduced performance if many end-to-end messages have to be exchanged in a short time period. This scenario is very typical for the initial setup of the whole task graph on the network, as well as for complete remapping.

A network flooding protocol such as Trickle [12] would lead to more efficient communications. However, using it would require keeping a copy of the complete task graph on the flash memory of each node. It would lead to increased implementation complexity, require energy for accessing the flash, and could wear it out in short time in dynamic networks.

The solution implemented in the current version of ProFuN TG is a Glossy [13] based scheduler, which replaces Rime for the first two traffic patterns. 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 message 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 in mass. 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 benefits of a flooding protocol-based solution are also clear from a more theoretical perspective. The purpose of the task setup algorithm is to achieve consistency in the network, while not sacrificing availability properties. According to the CAP (Consistency, Availability, Partition tolerance) principle [14], the system is therefore vulnerable to partitioning of the network. The single-path routing used by mesh protocol is much more likely to introduce transient partitioning of the routing tree, compared with the probability to have multiple partitions in the multi-path flooding graph.

Since Glossy is not by default capable of coexisting with ContikiMAC, the latter was extended with inactivity period, during which it does not attempt to send packets or listen to the radio. The duration and the frequency of these periods are controlled by the Glossy scheduler at runtime.

An additional benefit from the Glossy-based scheduler is for-free time synchronization on all nodes. Since we need synchronized clocks in order to measure end-to-end delays in the runtime phase, by running Glossy we avoid the need to have a separate time-synchronization protocol.

The middleware uses Contiki timing events to schedule the execution of periodic tasks, and a custom Contiki process event to notify event-based tasks about incoming of new data. In both cases, there is just one handler process that receives all the events, and based on them and task state determines which task callbacks should be executed.

Last but not least, the middleware is responsible for detecting violations of constraint conditions. To do that, it keeps track of the performance history for each constraint on each active data flow at its destination node.

There are two different implementation approaches to runtime checks of a data flow:

- Windowed approach. A buffer containing the last \( n \) time series values is kept and dynamically updated. The algorithm does not take into account any values beyond this window.
- Moving average approach. A scalar value containing an aggregate of the whole observed time series data is kept and updated.

In our case, the window approach is only feasible for PDR history. Single-bit-per packet memory requirement means that keeping a window of delivery history of last \( n \) packets with 2-byte sequence numbers requires just \( \lceil n/8 \rceil + 2 \) bytes. In contrast, keeping a window of packet delay history (2-byte values, non-compressible in the general case) requires \( 2n \) bytes; keeping, for example, \( n = 100 \) values for each delay constraint would be prohibitively expensive on nodes with limited RAM.

As a consequence, delay history performance is estimated using the EWMA algorithm, while PDR performance is estimated using either EWMA or a windowed approach. Both the window size and the \( \alpha \) value of EWMA (the decay rate) are user configurable parameters.

A dataflow packet is detected as “not received” when a periodic timeout timer for that dataflow is fired. This creates a problem for tracking PDR and delay on event-based dataflows. Following the ideas of Wu et al. [13], our approach is to track PDR on such dataflows by simulating the event on the source node and sending the produced data to the destination task as usual, only different in the way that the data in that packet are marked as “test-only”. These test packets are received and processed by the middleware normally, with the exception of the last step: they are not fed as inputs to the destination task. Since sending these test packets takes significant energy, it is done only when the user has explicitly configured to do so in addition to placing the constraint.

V. EVALUATION

A. The task allocator

Optimal task allocation is a combinatorial problem with exponential-sized search space, therefore domain-specific heuristics are believed to be necessary to make the search feasible.

We compare the performance of our task allocation algorithm with that of Bijarbooneh et al. [16], as well as with the greedy algorithm by Pathak et al. [17].

We reuse the test instances from these two papers, and measure performance on HVAC and traffic application task graphs, as well as for the task graph of the example application (Section III) in \( N \times N \) grid networks. Node capabilities in
the networks are randomized: each node has 20% probability of supporting a specific hardware component (temperature or smoke sensor, heater or alarm actuator). No node supports more than one component. For ProFuN TG, we measure the time-to-solve both on a "plain" task graph and on a graph where constraints have been enabled and rule out task-to-task communication paths with more than two intermediate hops.

Performance is measured on a PC with Intel Core i7 3.4 GHz CPU (4 physical cores) running Ubuntu 14.04.1 LTS. We use 8 threads for Gecode-based algorithms and a single thread for the greedy algorithm, as it is fast and not trivially parallelizable.

We use a cutoff time of 10 minutes to bound the maximal runtime on each instance. Algorithms that haven’t finished after that time are considered unsuccessful on that instance and terminated.

The measurements do not include time to preprocess the model (i.e. to find routes in the network); this phase is skipped to make the comparison more fair, as the algorithm of Bijarbooneh uses an already preprocessed model as input. The preprocessing time can be quite significant, as discussed in Section V-C, but its complexity is polynomial, therefore the maximal execution time of this preprocessing is relatively well predictable and, on complex instances, negligible compared.
with time required to find the optimal mapping.

The results (Fig. 10) show that for the example application task graph, the ProFuN TG algorithm executes within ten seconds even on very large networks.

Overall, ProFuN TG algorithm has much better performance than Bijarbooneh et al.’s algorithm for the AVERAGE ENERGY objective function. One reason for this could be the fact that Bijarbooneh’s algorithm is not a parallelizable: during the search process, we typically observed only one CPU core having full load, while ProFuN TG algorithm typically used all 8 cores. We speculate that due to using a more intricate model structure, Bijarbooneh’s algorithm creates interdependencies between search threads that lead to blocking waits. However, the speed difference of our algorithms on many instances goes far beyond the constant factor of 8.

In contrast, for the MAX ENERGY objective function Bijarbooneh’s algorithm is typically better, and in gives feasible results on all instances, except a few of the largest grid network instances. According to private communication with the author, it is explained by the fact their model successfully exploits the properties of this objective function, while the AVERAGE ENERGY objective function does not permit similar gains at the level of the model.

Search on a graph with constraints on grid networks leads to even faster performance of ProFuN TG algorithm, as these constraints reduce the number of candidate mappings that need to be evaluated. However, for all the HVAC instances (Fig. 8), except the two simplest ones, this constraint is too tight and causes no satisfactory solutions to be found at all. Also, Fig. 9b shows that MAX ENERGY objective function on traffic instances is an exception: constrained search is slower there.

More complex task graph instances (Fig. 9) create problems for the branch-and-bound algorithm, making it prohibitively slow on large networks. The time-to-search until the optimal solution usually is much smaller than the time to prove the optimality of the solution found. However, in a few isolated instances (e.g. traffic task graph 45 tasks in a network with 32 nodes) it is consistently higher than the time of the full search. We speculate that this anomaly is a measurement artifact, and that the performance hit comes from evaluation of the extra if statement that compares the value of the objective function with the optimal value, which is passed as argument. The if statement is conditionally included in program code, and is compiled only for the time-to-best-solution tests.
The solutions that the greedy algorithm produces are often close to the optimal, and its performance is even better than that of ProFuN TG algorithm — and, more importantly, its time complexity is always guaranteed to be polynomial. For example, it gives perfect solutions for HVAC instances, and 40% median error for traffic instances. This is not a surprise, as the test instances were created by Pathak et al. to showcase the algorithm [17]. However, simple instances exist on which the greedy algorithm gives arbitrary worse results, or even fails to find a feasible solution. For example, in the network depicted in Fig. 11, the greedy algorithm will start by mapping tasks A and B on node 3. That node is disconnected from the rest of the network. Since there are no more arrows pointing in our out of B, this task is never going to be remapped to node 2. Therefore, wherever task A is placed, it is not going to be able to communicate with both B and C at the same time. The greedy algorithm is going to find a feasible mapping in Fig. 11, however, it is going to be many times worse than the optimal solution found by the tree search. We can conclude that the greedy algorithm is not a reliable replacement for full-scale search even in simple networks.

The recommended method in practice is therefore to run the full tree-search algorithm with a user-specified timeout as a configuration option. If the algorithm terminates before the timeout, it has found an optimal solution. If it does not, the solution found at that point may already be the best (the algorithm just has not proved its optimality yet), or fairly close to it, as evidenced by the “time to best solution” line in the figures.

B. Runtime system

The runtime system is feasible on low-power msp430 MCU based sensor nodes: excluding Contiki system code, the Glossy scheduler, and implementations of pre-defined tasks, the middleware with the default configuration settings uses 1.4 KB of RAM and 6.6 KB of flash memory. The runtime state of a single task uses 30 bytes of RAM, so up to approximately hundred tasks can be instantiated in a single Tmote Skylike node. Additionally, each outgoing connection to a local task uses 6 bytes of RAM, to a remote task: 16 bytes, each constraint: 26 bytes.

To evaluate dynamic performance we compare a Rime mesh-based and a Glossy-based implementations of the task management protocol. We use the Cooja simulator, and report the average performance of 10 runs. For Rime mesh, 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 sensor nodes do not have enough RAM to hold the complete network routing table in memory. For Glossy, we compare 4-second (“default”) and 1-second (“fast”) round time. The default Glossy round length is selected to give similar radio duty cycle to Rime for these tests. Each Glossy round has a maximum of 14 flooding slots: 6 for the gateway, 8 for maximum of 4 nodes.

To minimize the number of random variables for which to control, we use a simple and fixed network topology: $N \times N$ grids, where all radio links have 80% Rx success probability.

First, we measure the time to setup a single task on a node $N$ hops away from the gateway (Fig. 12). Then we also measure the time to set up a single, but unique task on each of the nodes. We start the timer when the gateway sends out the first “create task” message. We end the first type of test when the gateway receives ACK message from the single destination node, and the second when ACK is received from all nodes. The gateway tries to resend the “create task” message for each task once every 15 sec until it receives an ACK; if no queue space is available (maximal queue size is set to 6 packets), it waits for a second and retries. For Rime mesh, we limit each message to maximum of 15 hop-to-hop retransmissions, and otherwise use the default Contiki settings.

For a single task, the default periodic Glossy-based scheduler shows (Fig. 13a) better performance than Rime in almost all networks, as well as more predictable delivery times. However, Glossy is also capable of setting up a task on each of nodes in less than 2 min on all networks, showing much better scalability. Rime, in constrast, is already not able to do this task within our 10 min cutoff time on 49-node networks, so we do not evaluate it on larger networks. Furthermore, when Glossy with target-specific schedule is used, the time-to-setup becomes independent of the network size and diameter: the flooding protocol causes all nodes to receive all messages in any case.

Another nice property of the flooding protocol is that reducing its period leads to proportional performance increase. Its schedule could be made several times tighter than our “fast” scheduler without risking introduction of wireless medium contention, so the user is free to choose an application-specific tradeoff between performance and energy consumption. The performance of Rime, in contrast, is known not to scale linearly with increased duty cycle.

We also measure the time to reallocate a single task after...
a failure detection (Fig. 13b). The scenario involves moving a source task from node $S_1$ to node $S_2$, and is completely controlled by the gateway server. The timer in this scenario is started when the destination node $D$ sends out the alert message and stopped when the gateway successfully receives an ACK from $S_2$. This scenario is harder for the Glossy-based protocol, but nevertheless, target-specific schedule with the default period is competitive with static routing, and schedule with the fast period is significantly better.

To summarize radio duty cycle measurements, for the representative 25-node test case the 10-run averaged average and maximal radio-on proportion is 3.22%/6.82% for Rime for a single task, 2.47%/5.43% for Rime all tasks, 2.79%/5.43% for Glossy for a single task, 3.80%/4.61% for Glossy all tasks, and 8.4%/22.4% & 13.0%/15.0% for Glossy with fast schedule, respectively. However, since the all-task setup duration is 13.7 times smaller for Glossy (with default round period) compared to Rime, the actual average radio-on time is $\frac{13.7 \times 2.47}{3.8} = 8.9$ times smaller for Glossy!

Overall, the flooding protocol emerges as a clear winner for the task setup stage.

\section*{C. Frontend}

The system as such is capable of handling large networks, as long as the task graph is relatively simple. Figure 14 shows the performance of the frontend depending on the size of the network.

Time-to-redraw (Fig. 14a and 14c) is measured when the whole network is visible on screen simultaneously. As it can be seen, the performance is above the cinema gold standard of 24 frames per second even for a 1000-node network, and remains close to that for a 700-link network with 100 nodes.

Task allocation performance (Fig. 14b and 14d) is dependent mostly on the speed of the task allocation algorithm; the time for other processing and delay due to TCP interprocess communication overhead are relatively small even on large network instances. In the test, we use a simple task graph consisting of two connected tasks and require that a copy of each pair is setup on every network node. 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, where $N$ is the set of nodes. For a 1000-node network the whole process takes approximately one minute.

\section*{D. Constraint status prediction accuracy}

We focus on evaluating our model rather than our implementation. The core question is “given a specific constraint and network’s performance history data, with what accuracy can we predict that the constraint is going to hold in that network in near future?”

We use simulations based on packet traces on links in our 17-node testbed deployed in our office building in Uppsala University (Fig. 15). We started by running a link discovery test in the testbed. The test included packet broadcast bursts from each node in turn, scheduled in round-robin fashion. Each burst included sending 10 000 packets, 84 bytes each, and lasted approximately one minute. The PDR of each broadcast was logged in each receiver node, both the cumulative value of the whole burst and the PDR in increments of 100 packets. In
For the evaluation of the tool, initially 3 nodes were selected for the roles of heater actuators. Each had 3 temperature sensor nodes sending data to them with 15 second period. On the dataflows we put constraints on maximal delay \((C = 2000\, \text{ms}, P = 98.5\, \%)\). We measured raw packet delivery rate in the network in for 24 hours on 3 different radio channels. At least 490 000 packets were sent on each of the 9 links in each period. These traces were input to a simulator, in which the application was simulated 10 times for each link for the 24 hour period, and the condition of the constraint evaluated once every minute. It was satisfied in 72.3 % of test cases.

We evaluate a few simple prediction algorithms (Fig. 16), trying to see if the constraint is going to hold 30 min in future. We did not fix any distribution-from-data inference algorithm, but instead operated on raw delay data. Any aggregation would lose information and decrease the prediction accuracy; on the other hand, more advanced prediction algorithms could counteract that.
VI. RELATED WORK

A. Programming model

There is a large body of work on high-level programming for sensor networks; however, tolerance to failures has been noted [18] as an open research issue. We chose ATaG as the underlying formalism because it naturally allows to increase dependability of sensor network 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.

We depart from the canonical ATaG programming model [11] by simplifying the dataflow programming language. In particular, we do not include support for Abstract Data Items as first-class values. This simplifies the semantics of the application, but makes it impossible to specify more advanced types of interfaces between tasks (namely, the parent-child aggregation relationship). We find that this change is justified because having a language-level support for aggregation relationships, something not typically used in every application, imposes significant cost on almost all applications. Consider an application with just four tasks that are all directly connected. Our approach requires just 10 elements to fully specify this application (4 tasks and 6 dataflows); ATaG requires 4 tasks, 6 data items, and 12 dataflows: 22 elements in total. This complexity increase is a hefty cost for a feature that is not universally required.

For the overall accuracy (averaged across the three different radio channels), simple “output the current state of the constraint” algorithm is not beaten: 95.74% accuracy (5.47 percentage point standard deviation). However, we want the predictor to have conservative bias, so the proportion of false positives is an even more important metric. We generate a datastream consisting on the number of out-of-bounds delays for a tighter constraint with a bound $C = 1000$ ms. To that datastream we apply per-hour moving average and EWMA algorithms ($\alpha = 0.1$); the first gives 0.59% false positives with overall accuracy above 90%; the second 0.41% false positives while keeping it above 80%.

Fig. 16: Constraint status prediction accuracy for 30 min in future.
Another capability that was lost because of this simplification is the ability to exert finer control of the semantics of individual data flows: ATaG allows flows to be either pull-based or push-based; in contrast, from the point of ProFuN TG tasks, all communications are push-based. Nevertheless, this potential problem is mitigated by the fact that the runtime system of ProFuN TG may itself be implemented in a pull-based fashion, for example, using the publish-subscribe paradigm, where the receiver task is the one that initiates data collection from all connected sender tasks. We are working on such an implementation.

Our additions to ATaG have been already described (Section II-A). The include the support for reliability requirements, as well as support for regioning the network based on node properties.

B. Related tools

Srijan toolkit [19] is a graphical ATaG macroprogramming system. In contrast to our work, it is missing the features introduced by the constraints: both the predictive aspect at design time, and the diagnostic aspect at runtime. Furthermore, in Srijan, tasks in must be implemented in Java programming language and require JVM at runtime.

There are other tools with functionality which to some extent overlaps with ours: high-level programming environments [20] for WSN, deployment and experimentation support systems [21] [22], and runtime assurance systems [23], but they all lack automated algorithms for task allocation. As for sensor data storage, processing and visualization frameworks such as SicsthSense, they complement the functionality of our tool.

The user interface of ProFuN TG is adapted from NodeRED. However, not only we added the network model, the constraints, and the macrocompilation and deployment system, we also do not share their run-time system that is implemented in JavaScript, therefore requires interpreter support and HTTP communication plane at runtime.

C. Reliability and efficiency aspects

We took the general idea of user-defined probabilistic end-to-end constraints from Bjarbooneh et al. [2]. The main difference between these two models lies in the fact that their model included probability distributions on links between tasks, while our has them on network links. As a result, their design-time constraint satisfaction checker cannot differentiate between a dataflow on a single-hop path, and a dataflow between tasks separated by a multiple hops, while our task-mapping algorithm takes the number of hops into account.

Energy-efficient task mapping for WSNs has already been explored in detail [17] [16] [24]. Our main contribution here is an optimal mapping algorithm with considerably improved performance.

D. System design

WSN are typically treated as decentralized systems. Nevertheless, our work is far from unique in treating them in conjunction with a powerful centralized server-side functionality. The approach adapted here has roots in industrial control systems [25].

Specifically in WSN, the approach has been seen not only in task-allocation tools, but also in run-time assurance systems [15] [23], and tools for network-wide parameter adaptation [26] [20]. Another notable example is the WirelessHART protocol stack that uses centralized scheduling mechanisms and is an industry standard [27].

In particular, the seminal work by Ferrari et al. [13] on Glossy, which demonstrated how simultaneous transmissions can be used to transfer data in WSN using commercial off-the-shelf sensor node hardware, opened many new doors for centralized systems, ranging from more efficient data collection [13] and dissemination [26], to generic network-wide communication primitives [28], to network-wide synchronization guarantees [29].

E. Granularity and types of tasks

The task graphs we envision as typical for WSN applications do not contain tasks with small granularity. This is in contrast to visual dataflow programming languages such as LabVIEW [30], which use separate processing elements even for elementary operations such as Boolean functions AND, OR, and NOT. First, expressing every operation visually has been shown to reduce readability of the representation [31]. Second, and perhaps more to the point in WSN context, each task requires a significant amount of run-time state. Using several tens of bytes of RAM for each application-level logical operation is prohibitively expensive on a system that has just a few kilobytes of RAM in total.

We also eschewed the support for predefined tasks for generic data aggregation operations. Even though implementing such a generic task functionality for calculating e.g. arithmetical average of multiple inputs may seem trivial, in real-world applications this kind of aggregation has to deal with faulty and missing data, which makes the desired semantics far from clear. For example, should the faulty data be detected by applying a threshold? An absolute value or value relative to the readings of other sensors? Should the readings be clustered and the largest cluster assumed to be the correct one? Or something different still [32]? Should a missing data item be replaced with the latest non-faulty value from the same sensor? An average value of other sensors? A value predicted by some kind of model? These choices are application specific, therefore building a universal middleware level data aggregation block is not possible.

VII. Concluding remarks

ProFuN TG enables design of task graph applications that are aware of data quality requirements. It achieves that by allowing the user to write PDR and delay constraints on dataflows between tasks. The tool also enables deployment and maintenance of these applications in WSN by providing middleware that manages the runtime state of tasks and constraint conditions, and triggers reallocation in case a constraint violation is detected.
Our evaluation shows that the allocation algorithm can produce optimal task mappings in an acceptable time (up to a minute) even on very large networks (hundreds of nodes). In addition, the task setup protocol can instantiate runtime tasks on tens of nodes within a minute, making the reallocation approach feasible in hard-to-predict, constantly changing real-world environments.

In future work we will look at how the central system and the runtime middleware can work together to adapt the network for the QoS requirements without resorting to task reallocation. We will also look at how the task-to-task communication can be made more predictable by using technologies such as the Low Power Wireless Bus (LWB) [28].

ACKNOWLEDGMENTS

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

Thanks to Farshid Hassani Bijarbooneh and Martin Jacobsen for sharing their ideas and code, to UU IT department students (Andreas Moregård, Simon Young, and Per Ackermann-Bostrom) for the initial implementation of the middleware, and to Hongbin Liu for pointing out how to use principal component analysis for sensor fault detection.

REFERENCES


