Elsts, A., Hassani Bijarbooneh, F., Jacobsson, M., & Sagonas, K. (2015). Enabling Design of Performance-Controlled Sensor Network Applications through Task Allocation and Reallocation. In *2015 International Conference on Distributed Computing in Sensor Systems (DCOSS 2015): Proceedings of a meeting held 10-12 June 2015, Fortaleza, Brazil* (pp. 248-253). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/DCOSS.2015.44

Peer reviewed version

Link to published version (if available):
10.1109/DCOSS.2015.44

Link to publication record in Explore Bristol Research
PDF-document

This is the author accepted manuscript (AAM). The final published version (version of record) is available online via IEEE at http://ieeexplore.ieee.org/document/7165052/. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/red/research-policy/pure/user-guides/ebr-terms/

Enabling Design of Performance-Controlled Sensor Network Applications Through Task Allocation and Reallocation

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

***Abstract*—Abstract Task Graph (ATaG) is a sensor network application development paradigm where the application is visually described by a graph where the nodes correspond to application-level tasks and edges correspond to dataflows. We extend ATaG with the option to add nonfunctional requirements: constraints on end-to-end delay and packet delivery rate. Setting up these constraints at the design phase naturally leads to enabling run-time assurance at the deployment phase, when the conditions of the constraints are used as network's performance goals. We provide both run-time middleware that checks the conditions of these constraints and a central management unit that dynamically adapts the system by doing task reallocation and putting task copies on redundant nodes. Through extensive simulations we show that the system is efficient enough to enable adaptations within tens of seconds even in large networks.**

I. Introduction

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

- 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 doing what it is intended to, and meeting its QoS requirements.
- Make use of redundant hardware nodes in order to improve data quality and at the same time increase network's lifetime and maintenance intervals.

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

Help is offered by high-level WSN macroprogramming methodologies such as the Abstract Task Graph (ATaG) [1]. We have implemented ATaG in ProFuN TG [1] [2], a high-level sensor network development toolkit. ProFuN TG allows users to describe the functionality of an application with a task graph, to macrocompile its code, and to deploy it in real and simulated networks.

In this paper, we describe how ATaG is extended in our implementation by allowing to incorporate end-to-end reliability requirements in descriptions of applications. The requirements are expressed in form of *constraints* on packet delivery rate (PDR) and delay, and are set on dataflows between tasks.

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

At the design stage, the tool takes these requirements in conjunction with a model of the network as the input, and outputs an optimized, constraint-satisfying task mapping.

The supporting run-time middleware (implemented for *msp430* MCU based sensor nodes) efficiently sets up the task mapping in the network, manages task-to-task communication, gathers application performance statistics and determines whether the conditions of the constraints hold, enabling run-time assurance. On failures, alert notifications are issued, and automated maintenance through task remapping is performed.

This work provides a bridge between an existing high-level WSN programming paradigm, ATaG, and several other research ideas; specifically, the idea of end-to-end constraints on dataflows as a useful abstraction for the WSN application programmer [3] and the idea of run-time assurance [4] as an important non-functional aspect of WSN applications. We show how allowing the user to specify high-level constraints in the design phase naturally leads to enabling run-time assurance in the deployment phase.

II. Conceptual foundations

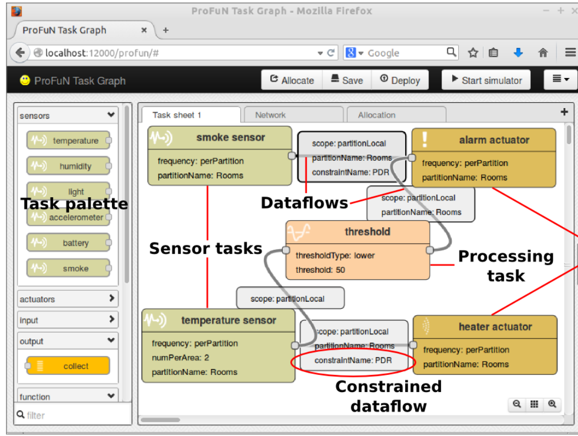
A. Programming model

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.

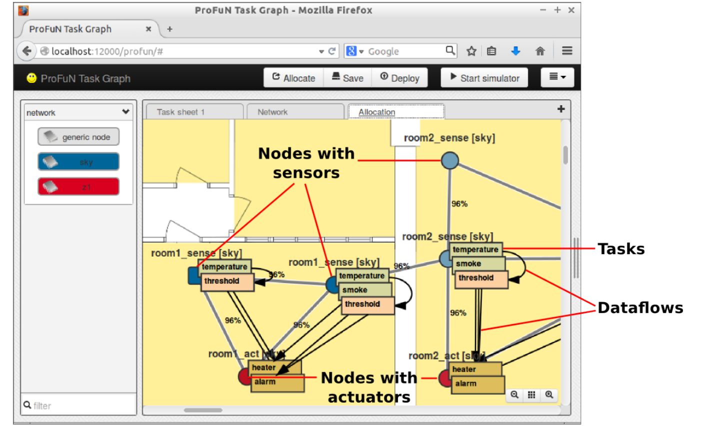
An *abstract task* is a clearly defined chunk of application-level functionality. Tasks are annotated with properties, such as their firing rule (periodic or event-based), firing period, and the number of task copies to instantiate. Each abstract task is instantiated on one or more sensor nodes.

An *abstract dataflow* is a link that connects a pair of abstract tasks. All dataflows have *scope*: a property that restricts the maximal distance (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 maximal 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 is textual and imperative. ProFuN TG task code is written in C.



(a) Task graph view showing tasks and their relationships



(b) Network view showing (a part of) the allocation of tasks to nodes

Fig. 1: **The visual interface of ProFuN TG**, showing a heating control application with fire detection

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.

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 can be set using *name:value* syntax. 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.

C. Constraints

One of the key features of ProFuN TG is its 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 avoids producing mappings that violate them.
• Diagnostic: the runtime system continuously tests whether constraint conditions are met. In case the test fails on a node, it 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 $P = 0.98$:

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

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 number of values has been gathered. This minimal-number-of-values is another user-configurable parameter.

III. Design-time 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 the 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. We require that each heater task should receive input from at least two temperature tasks located in the same room, and each

alarm task should receive data from a smoke sensor in the same room, with delay smaller than 30 seconds with at least 99.5 % probability.

ProFuN TG allows to configure high-level relations between tasks 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 operates as a filter on the set of nodes eligible to host the task.

The user may require certain task-to-task reliability guarantees. In the general case, it is not possible to reduce these guarantees to a simple metric such as the 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 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:

```
Normal(0.5,0.0): 0.6, 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. 2).

As an example consider a constraint on maximal delay. 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:

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

By definition,

$$\mathbf{P}(\text{Delay}_{path(s,d)} < C) = \text{CDF}_{Path(s,d)}(C)$$

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

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

$$\text{PDF}_{path(s,d)} \approx \sum_{link \in path(s,d)} \text{PDF}_{link}$$

where the sum is calculated as convolution of the link PDFs.

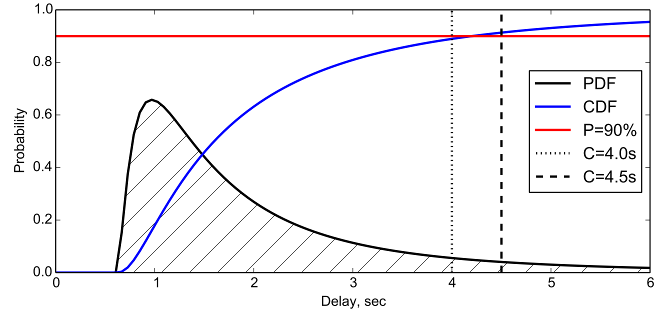


Fig. 2: **Probabilistic constraints on a log-normal delay distribution.** Given minimal probability $P = 0.9$, constraints with delay value of $C = 4\,s$ are not satisfiable, while constraints with delay value $C = 4.5\,s$ *are* satisfiable (in the model).

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

$$\text{CDF}_{path(s,d)}(C) \approx \int_{-\infty}^{C} \sum_{link \in path(s,d)} \text{PDF}_{link}(t)\text{d}t$$

Since in the general case the resulting multiple 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 maximal retranmissions is finite, there is a non-zero probability that the packet is never delivered. To handle this case, the user should include an additional term as a part of the mixture description of the delay distribution of a link. The term can be a single point at a very large t coordinate. For example, if the “infinity” floating point value defined by IEEE 754 standard is used for this purpose, the path CDF is guaranteed to be above any finite constant C if a least one of the links drops the packet.

What happens when there are *no* satisfying mappings for a particular pair of communicating tasks? In this case, ProFuN TG is capable of automatically creating *duplicate copies* of the source task of the pair. Continuing with the delay constraint example, it is easy to see that this increases the probability that the destination task d will receive data with acceptable delay from at least one source task s_i (assuming the probabilities are independently and identically distributed):

$$\mathbf{P}_{\text{fail}}(C, s, d) = 1 - \text{CDF}_{path(s,d)}(C)$$

$$\mathbf{P}_{\text{fail}}(C, \{s_1, s_2, \ldots, s_n\}, d) = \prod_{i=1}^{n} \mathbf{P}_{\text{fail}}(C, s_i, d)$$

where $\mathbf{P}_{\text{fail}}(C, s, d)$ is the probability that the path from s to d does not satisfy the delay bound C.

The tool estimates k — the required number of source task's copies to satisfy C with probability P — by using the equation:

$$(\mathbf{P}_{\text{fail}}(C, s_1, d))^k \geq 1 - P$$

As k must be an integer, the estimate is given by:

$$k = \left\lceil \left(\frac{\log(1-P)}{\log(\mathbf{P}_{\text{fail}})} \right) \right\rceil$$

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

By allowing to re-run the mapping algorithm when updated perfomance statistics and alarms are received from the network, ProFuN TG enables automated maintenance and adaptive optimization of the network.

IV. Run-time functionality

When all abstract tasks have been mapped on the network model in a way that satisfies the user, the next step is to issue the *"Deploy"* command. On this command, the frontend of the tool sends the current task mapping to active gateway servers. Each gateway server then sends out commands in the WSN to create runtime state for the mapped tasks on the network nodes. This is done both for nodes connected by a cable and wirelessly; it is automated by the ProFuN TG middleware.

The middleware is a C library we 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:

1) gateway to nodes (*mesh* protocol);
2) nodes to gateway (*collect* protocol);
3) 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 in the task-setup stage. Setting up a task on a remote node requires reliable end-to-end transport both for the task message and its middleware-level acknowledgement. Setting up another task on a neighboring node requires almost completely repeating the process. The default CSMA-based MAC protocol leads to severely reduced performance if many end-to-end messages have to be exchanged within 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 [5] 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 [6] based scheduler, which replaces Rime for the first two traffic patterns. The gateway-controlled scheduler has support for two phases: a *periodic* schedule phase, in which all nodes can originate messages to the gateway periodically, and a *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 in the task graph, and is significantly more energy efficient and faster: the node sends an ACK immediately, without waiting up to several seconds for its periodic schedule slot.

To assure that the application is working correctly, the middleware gathers application performance statistics and determines whether the conditions of the constraints hold, enabling maintenance alert notifications in case of link-level and node-level faults, as well as automated maintenance through task remapping. To do that, it keeps track of the performance history for each constraint on each active data flow at its destination node. The history is kept either as bit-buffer marking which packets have been received, or as a scalar EWMA value of past performance, depending on a configuration option.

There is also a compile-time option to store hop-by-hop performance (time-series average of PDR and delay) on all network links used by task-to-task traffic. If this is enabled, these statistics are collected in the network and periodically sent to the gateway, so that they can be used to update the network model.

V. Evaluation

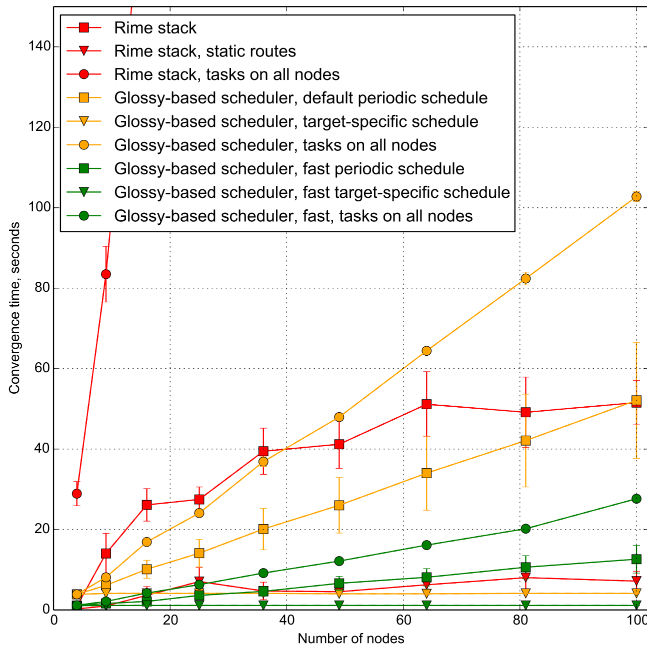
The runtime system is feasible on low-power *msp430* MCU based sensor nodes: the middleware with the default configuration settings uses 1.4 KB of RAM and 6.6 KB of flash memory. The runtime state of a single task uses 30 bytes of RAM, so up to approximately hundred tasks can be instantiated in a single Tmote Sky-like 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 Rime *mesh*-based and Glossy-based implementations of the task management protocol. We use the Cooja simulator and report the average performance of 10 runs.

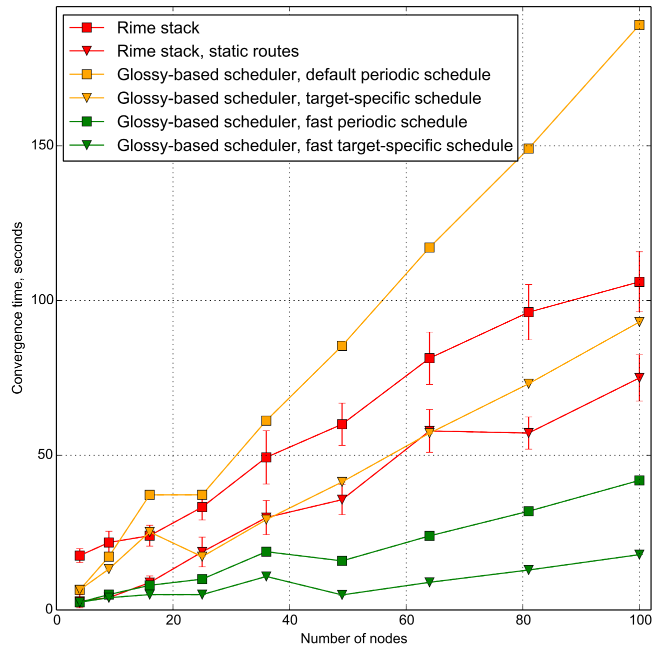
For Rime *mesh*, we compare the performance of two cases: (1) all nodes start with empty routing tables, and (2) static routes are pre-installed along the forwarding path. The second approach leads to higher performance, but it is not going to scale, as sensor nodes do not have enough RAM to hold the 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 D that is N hops away from the gateway (Fig. 4). Then we



(a) Task setup



(b) Task reallocation

Fig. 3: **Performance of the task management protocol.** Error bars show standard deviation divided by 2.

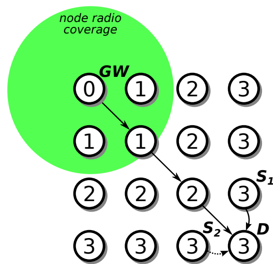


Fig. 4: **Example of a 16-node grid network with uniform radio coverage.** The numbers on nodes describe number of hops to gateway. *GW*: gateway node, *D*: destination node, S_1: initial source node of a dataflow, S_2: source node after task remapping. Straight arrows: task setup message path for node D; curved arrows: task-to-task dataflows.

also measure the time to set up a single, but unique task on *each* of the nodes. For a single task, the default periodic Glossy-based scheduler shows (Fig. 3a) 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 minutes on all networks, showing much better scalability. Rime, in constrast, is already not able to do this task within our 10 minute 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. Finally, reducing the period of the flooding protocol leads to proportional performance increase, while the performance of Rime is known *not* to scale linearly with increased duty cycle.

We also measure the time to reallocate a single task, moving it from S_1 to S_2 after a failure detection (Fig. 3b). This scenario is harder for the Glossy-based protocol, but nevertheless, target-specific schedule is competitive with Rime & static routing, and schedule with the fast period is significantly better.

To summarize the 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 $\frac{13.7\times2.47}{3.8} = 8.9$ times smaller for Glossy!

VI. Related work

There is a large body of work on high-level programming for sensor networks; however, tolerance to failures has been noted [7] as an open research issue. 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.

We took the general idea of user-defined probabilistic end-to-end constraints from Bijarbooneh *et al.* [3]. 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.

Furthermore, their automated reasoning about performance of the system is severely limited by the capabilities of symbolic integration: only normally distributed random variables are supported. This is insufficient to model sensors networks accurately, as a distribution that describes e.g. a delay on a link is likely to be: (1) skewed, (2) with a long-tail, and (3) multimodal. We use numerical integration, and therefore are able to support arbitrary mixtures of sums of distributions from the exponential family (i.e. normal, log-normal, Pareto), as well as the uniform distribution.

Similarly to our work, Srijan toolkit [8] is a graphical ATaG macroprogramming system. However, 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 must be implemented in Java programming language and require the presence of JVM at runtime.

There are other tools with functionality that overlaps with ours to some extent. make*Sense* [9]) is a high-level WSN programming toolkit that includes dynamic run-time adaptation to application goals. The adaptations are relevant to specific parts of the system, and 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 [9] through repeated simulations. Therefore, it is a black-box approach in which integration of expert knowledge is not easily possible. Furthermore, the make*Sense* runtime system requires more extensive information about the network state to enable adaptations, while our solution in the typical case of periodic task-to-task traffic adds overhead only for sending a single alarm message from the network to the gateway.

pTunes [10] is an approach for network-wide parameter adaptation. However, pTunes is designed specifically for collection tree based periodic sense-only applications, and adds non-negligible overhead due to frequent and continuos network-wide link-state information gathering: constant 0.07 % to 0.35 % radio duty cycle overhead [10].

Both in make*Sense* and pTunes, the result of the optimization procedure is a set of new parameters for the network; these tools are unable to use our methods to increase robustness, i.e. reallocate tasks on different nodes and to duplicate them for redundancy. Similarly, deployment and experimentation support systems such as DREAMS [11] and MakeSense [12], and runtime assurance systems such as the ones developed by Wu *et al.* [13] and Fairbairn *et al.* [4] all lack the capabilities of task allocation and reallocation.

Redundancy *is* heavily exploited by the two competing standards in the area of WSN for industrial monitoring and automation: WirelessHART [14] and ISA100.11a [15]. Both enable dynamic application-level adaptations, and in both, routing and scheduling information is dynamically calculated by the central network manager based on topology information it continuously collects from the network. However, this approach adds up to high implementation and management complexity, and significant operational overhead for the collection of the required information.

VII. CONCLUDING REMARKS

ProFuN TG enables design of task graph applications that are aware of performance 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 a 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 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.

ACKNOWLEDGMENTS

The authors acknowledge support from SSF, the Swedish Foundation for Strategic Research. Thanks to UU IT department students for the initial implementation of the middleware.

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 *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*. USENIX Association, 2005, pp. 19–24.

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

[3] 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.

[4] 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.

[5] P. A. Levis, N. Patel, D. Culler, and S. Shenker, *Trickle: A self regulating algorithm for code propagation and maintenance in wireless sensor networks*. Computer Science Division, University of California, 2003.

[6] 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.

[7] 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.

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

[9] F. Casati, F. Daniel, G. Dantchev *et al.*, “Towards business processes orchestrating the physical enterprise with wireless sensor networks,” in *Software Engineering (ICSE), 34th International Conference on*. IEEE, 2012, pp. 1357–1360.

[10] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele, “pTunes: Runtime parameter adaptation for low-power MAC protocols,” in *ACM/IEEE IPSN*, 2012, pp. 173–184.

[11] R. Figura, M. Ceriotti *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.

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

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

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

[15] “ISA-100 Wireless Compliance Institute: Official Site of ISA100 Wireless Standard,” http://www.isa100wci.org/.