Efficient Configuration and Control of SANETs using FACTS

Kirsten Terfloth and Jochen Schiller
Institute for Mathematics and Computer Science
Freie Universität Berlin
14195 Berlin, Germany
terfloth | schiller@inf.fu-berlin.de

ABSTRACT

The utilization of domain-specific programming abstractions for wireless sensor actor networks can greatly ease application development. A high level of abstraction from underlying system complexity fosters fast prototyping and less erroneous source code. However, especially in this domain access to low-level parameters can be mandatory to allow for sophisticated application fine-tuning.
In this paper, we present a layered system access interface we developed for our middleware platform FACTS to address this challenge. Context-based configuration and control of system-level resources is facilitated by relying on lean additions to the FACTS framework. These will be discussed with the help of a common use case to illustrate the simple, yet powerful abstraction provided.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design

Keywords

Wireless Sensor Actor Networks, Middleware, Layered System Access

1. INTRODUCTION

Domain-specific languages and dedicated middleware platforms for wireless sensor actor networks target to provide suitable programming abstractions to overcome intrinsic challenges of this domain. These networks are a composition of embedded, distributed nodes that communicate over a wireless link and often times operate in an event-driven manner, a combination that makes application development a complex task. Adaptation of nodes to internal and external events as well as global coordination patterns have to be implemented in an energy-efficient and robust manner to allow for long-lived applications. A variety of middleware platforms has been introduced so far to leverage some of the system-inherent complexity by means of e.g. support for dealing with asynchronous tasks, transparent networking or managed memory access.
Exporting a reasonable interface to system-related settings to applications is a non-trivial task in middleware design: Clearly, many applications developed for SANETs demand for flexible adaptations of underlying hardware at runtime, which prohibits static compile-time configurations. Parameters such as sampling frequencies, actuation timing or data filters have to be able to be individually altered based on the state of a running application. Furthermore, as soon as the application development process goes from a state of prototyping to application fine-tuning, a more elaborate way of hardware configuration may be needed than in an earlier state. While during prototyping e.g. energy-efficiency is negligible it will well be of interest in a final deployment. Thus, the granularity of system access may vary over the development time of an application.
This trade-off of supporting a high level of abstraction from system-related parameters to enable fast prototyping while at the same time allowing access to low-level hardware settings for application fine-tuning is addressed in this paper. Therefore, we developed a layered system abstraction interface for our event-driven middleware platform FACTS that utilizes building blocks already inherent to the system.
The contribution of this paper is two-fold: First of all, we analyzed system access patterns, unified common utilization mechanisms into one elaborate abstraction and added it as a tool for hardware configuration and control to the FACTS API. These so called `context_facts` can be used local to a node as well as in a global network setting for context-aware parameter adaptation. Furthermore, we developed a layer for system-level event processing which serves as a complement for application development. Composed of modular pieces of software, these `system rulesets` encapsulate common means for processing and filtering of low-level data.
The remainder of this paper is organized as follows: The next section briefly introduces the FACTS middleware framework and its domain-specific rule-based language RDL. Section 3 presents a use case typical for the wireless sensor network domain that will serve as a reference throughout this paper. The main part of the paper, Section 4 will provide an overview of the differentiated system access schemes we developed for FACTS. Finally, Section 5 reviews work related to ours with Section 6 concluding the paper.

HeterSANET'08, May 26, 2008, Hong Kong SAR, China.

2. FACTS MIDDLEWARE FRAMEWORK

FACTS is a lightweight middleware framework developed to run on embedded networked sensors. The key to our approach is to strictly stick to a data-centric paradigm, optimize local node behavior and hide concurrency issues and manual stack management from the programmer. Combining the advantages of a runtime environment with its sandboxed execution and the possibility to obtain very dense bytecode, FACTS offers a suitable programming abstraction for wireless sensor networks [7].

2.1 FACTS Architecture

The core idea for designing FACTS has been to provide a simple but versatile model for dealing with both an event-based execution environment and wireless communication using a unified data model. Therefore, we rely on so called `facts`, which are named tuples of typed values. Facts can be used to define application specific data, to trigger the execution of actions and to transparently exchange data over the network with other nodes. Facts thus serve as the main data abstraction for storing, transmitting and accessing data within the framework available to an application developer. The basic programming primitive in FACTS are `rules`, sets of conditional actions. This choice contributes to the predominantly asynchronous execution of tasks, with sensor nodes usually responding to external events or to internal changes of the current state of the node itself. Rules are a very concise programming primitive, see Listing 1. Three lines of code suffice to specify a rule that forwards new data samples to a sink node - regardless of whether the facts have been created locally or received from neighboring nodes and only in case a fact with routing information is available. The syntactical specification for rules will be briefly reviewed in Section 2.2. Sets of rules that semantically belong together can be composed into so called `rulesets` to achieve modularity within the system.

Conceptually, FACTS provides a runtime environment to process rules. Hence, each node exhibits a central entity for scheduling rule evaluation, the `rule engine`, and global storage for data, the `fact repository`, see Figure 2.1. The rule evaluation process is solely triggered when a new fact is added to or an existing fact is altered within the fact repository. This may either be attributed to a fact being produced by the underlying system, as output of or modification by a former rule execution or to the reception of a fact over the radio interface. Interaction with the underlying firmware via functions is transparently implemented by wrapping system information into facts and will be exhaustively discussed in Section 4. Rules have to be provided as bytecode of compiled rulesets to allow the interpretation of rule conditions and actions by the rule engine accordingly. Note that the distinction between middleware and application-level rulesets is rather a semantically motivated than an architectual differentiation.

Listing 1: Rule for transparent data forwarding.

```
1 rule forwarding 100
2 <- exists {data_sample}
3 <- exists {sink_entry}
4 -> send {sink_entry next_hop} 100
5    {data_sample}
```

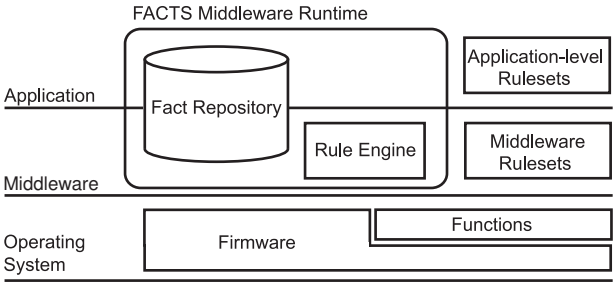

Figure 1: Components of the FACTS middleware framework

2.2 RDL - The Ruleset Definition Language

The event-centric nature of most algorithms and applications developed in the domain of wireless sensor networks is captured in FACTS at the language-level. Following a rule-based programming paradigm, the Ruleset Definition Language (RDL) is a very concise domain-specific language especially designed for node-level tasking. Programming primitives are rules, which are atomic units of conditional execution, and facts, a unified data abstraction of RDL. While formally RDL can be used to specify an unlimited number of recursions for conditional filtering, their practical implementation on sensor nodes can literally crash the nodes, a fact that led to bounding the allowed recursion depth. A brief formalization of the RDL language is presented below to serve as an overview of available language features for a latter use case discussion.

2.2.1 Facts

Let $F = \{f_1, f_2, ..., f_n\}, F \subset F^*$ denote the set of facts stored in the fact repository, where F^* corresponds to the set of all well-formed facts. Each fact has at least a name, thus a defined identifier, and may have a set of associated properties
$P = \{f_1.p_1,\ f_1.p_2, ...,\ f_1.p_a, ...,\ f_n.p_b\}$ within the domain of all well defined properties P^*, $P \subset P^*$.
A property is a key-value tuple, with available types for values being `bool, int, string, name`.
Besides application-specific properties, each fact is additionally tagged with system information including a timestamp, the id of the sensor node that altered or created the fact, a unique fact_id, a tag reflecting its current processing status, thus whether it was newly created, modified or not. The processing status is used to decide whether the corresponding fact is able to trigger the execution of a rule.

2.2.2 Rules

Let $R = \{r_1, r_2, ..., r_m\}$ be an arbitrary set of rules with a priority ordering $Prio = \{r_i > r_j | \text{rule } r_i \text{ has precedence over } r_j\}$. The definition of priorities is part of the rule syntax and therefore obligatory to the programmer. This is necessary since multiple rules may react to the same fact, so that the order of execution has to be explicitly specified. Hence, no guarantees will be given on the order of execution of rules with the same priority assigned.
To state when a rule fires, a rule r_i is composed of a set of conditions $C_i = \{\, r_i.c_1, \ldots, r_i.c_d\}$, with a set of statements $S_i = \{r_i.s_1, \ldots, r_i.s_e\}$ specifying what action will be taken. A condition can be of two possible types:

- $C_{ex} = \{\langle Ex, f\rangle \,|\, f \epsilon F^*\}$, the `exists` condition, allows to check whether a certain type of fact is currently stored within the fact repository.

- $C_{ev} = \{\langle Ev, f.p\rangle | f.p \epsilon P^*\}$, the `eval` condition, can be used to match the properties of facts against thresholds or in relation to other facts in the repository.

RDL allows a programmer to write nested conditions, thus to check whether a fact exists whose properties have to meet possibly a multitude of conditions. The statements of a rule r_i will only be triggered if $\forall_{j=1}^{d}\ r_i.c_j = true$, with d being the number of associated conditions. Furthermore, at least one of the facts involved in condition satisfaction has to be newly added or updated during the last run of the rule engine for rule execution. After a complete run of the rule engine all modified facts will be marked unmodified to prevent subsequent triggering. The sensor node can switch to low-power mode in case no more modified facts have to be matched against rules.
Currently, a static set of rules is assumed to be deployed on the nodes, but facts are dynamic and may change at runtime. Let S denote the set of available fact modification statements:

- $S_d = \{\langle D, f\rangle \,|\, f \epsilon F^*\}$ is a `define` statement, stating that a new fact with or without properties is being added to the fact repository.

- $S_r = \{\langle R, f\rangle \,|\, f \epsilon F\}$ is a `retract` statement to be invoked if fact(s) shall be deleted from the fact repository.

- $S_u = \{\langle U, f.p\rangle \,|\, f.p \epsilon P^*\}$ denotes the group of operations that update specified facts. This can either be a `set`, that operates on properties of facts, or a `flush` statement which will disable the specified fact from triggering a rule in the next run of the rule engine.

- $S_s = \{\langle S, f\rangle \,|\, f \epsilon F\}$ `send`s fact(s) over the radio interface. Both, unicast and broadcast messages are possible.

- $S_c = \{\langle C_i, f.p\rangle \,|\, i \epsilon I,\ f.p \epsilon P\}$ with I being the set of identifiers of functions defined in the firmware interface. This `Call` statement provides supervised access to the underlying software stack, possibly enabling resource-intense computations to be implemented in native code. These will discussed in the next section.

In rule-based programming, one tends to address a subset of facts for processing or rule triggering more than once. In order to make the code less verbose, we introduced the notion of *slots*. Slots are named filters on facts or on properties of facts. At runtime, slots are evaluated by matching the facts in the repository against the filtering rules. Naturally, this matching process may result in one, none or a set of facts meeting the slot, which in turn can trigger the execution of statements on one, none or a set of facts.
Another crucial part of RDL are *rulesets*, specifying a set of rules interacting with each other, analogous to a compilation unit in traditional programming languages. Rulesets may therefore be used to encapsulate services like routing, node self-inspection or application semantics.

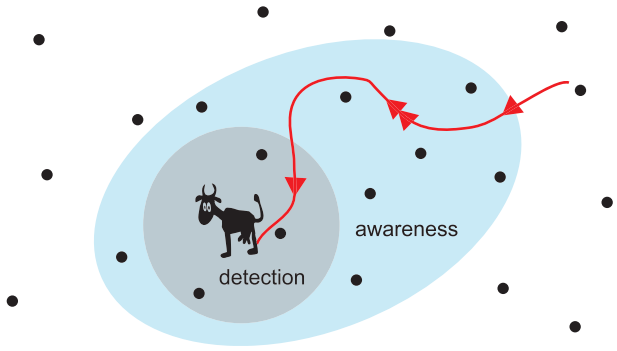

Figure 2: Tracking an object with wireless sensor networks

3. USE CASE

To illustrate the benefits of a layered access to system parameters we rely on an object tracking application which will serve as a use case throughout this paper. Note however that we do not intend to build a ready-to-deploy application but rather use this as a simple example to point out the strength of presented system features later on.

Consider a sensor network that is distributed within an area of interest and tasked to track e.g. animals of a certain species moving through the surveyed region, see Figure 3. With the help of appropriate sensors, a node is able to detect an animal approaching its monitored area and its presence in close vicinity to the node. For reasons of energy efficiency, it is favorable for nodes to remain in a low-power mode, thus stick to low duty-cycling, unless an event of interest is likely to be observed. As soon as an animal is approaching them, nodes change to a state of awareness and switch from passive monitoring to active sampling. Upon actual detection, sampling will be done with a greater resolution to get fine-grained event data. Since behavioral wild-life monitoring is usually not subject to any real-time requirements for user interaction, our tracking example does not include reporting event data to a central entity, but instead questions nodes to log this data on flash for later offline analysis.
For an implementation of this tracking scenario it is important to decide which sensors to utilize for presence recognition. Since this highly depends on the nature of the tracked object, the range of deployed hardware spans from acoustic, magnetic and motion sensors for e.g. battlefield surveillance to seismic or vibration sensors [2]. Our reference platform is the ScatterWeb MSB430 [6] that features a 16-bit microcontroller MSP430F1612 from Texas Instruments equipped with 55 KB of flash memory and 5 KB RAM, as well as a Chipcon CC1020 transceiver using the ISM band at 869 MHz. Available sensors include a Freescale Semiconductor MMA7260Q accelerometer, and a temperature and humidity sensor from Sensirion. For a prototype implementation of the tracking scenario, we rely on sampling the accelerometer and detecting the motion of an animal by means of measuring the intensity of its emitted physical activity.
Figure 3 illustrates the three states a node participating in our use case can adopt. Generally, a node will be in *idle* state with sensing disabled but regularly listening into the medium. The reception of a message containing an `acceleration_context` fact, see Section 4.2.4, will trigger sampling the accelerometer. In case a predefined threshold for the

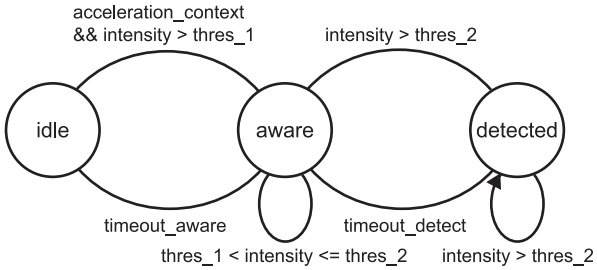

Figure 3: State machine of usecase.

measured intensity is crossed, node changes into the *aware* state, rebroadcast the received fact and resume sampling with a medium sampling frequency. Nodes in this state are aware of an animal possibly approaching themselves. A high intensity of the detected motion triggers nodes to change their current state to *detected*. Then, data will be gathered with a higher sampling frequency and automatically logged to secondary storage. ScatterWeb nodes are equipped with an SD slot mounted at the back of the node, thus may swap event data directly onto an SD card. In case the intensity of the detected motion drops and within a certain time frame no high intensity data samples are recorded, it is assumed that the animal leaves the area monitored by the corresponding node. A timeout triggers state adoption, finally allowing a node to go back to idle state when even no medium intensity can be detected any more.

4. SYSTEM ABSTRACTION IN FACTS

Application development for embedded networked sensors demands for a suitable interface to underlying hardware and system parameters. As can be derived from the use case, a programmer needs e.g. access to available sensors (and possibly actors), to peripheral hardware such as secondary storage and to a timer interface for manual flow control. Naturally, a high level of abstraction rather fosters fast prototyping while a lower level allows for better application-specific parameter tuning. In order to meet both requirements, FACTS provides a layered approach of dealing with calls to and return values from the underlying system and therefore exports several options or access points how to utilize them to a programmer.

The lowest level of abstraction is inherent to the language: RDL offers access to hard- and software via `Call` statements, and exports the native API where required. This way, full control of hardware settings is guaranteed when necessary.

To provide a better handle for hardware configuration and control, so called `context_facts` have been added to the API of FACTS. Dependent on the hardware resource, a corresponding `context_fact` aggregates configuration parameters and/or encapsulates its operational mode. As a consequence, a modification to a `context_fact` will result in the adaptation of its represented resource, giving an application programmer an easy-to-use abstraction to underlying hardware.

Finally, the observation that many applications share basic approaches e.g. in how to deal with system generated data led to the development of `system rulesets`. Akin to a system library, system rulesets are provided as a supplement that encapsulate tasks common to multiple applications.

4.1 Language-inherent System Access

As mentioned above, `Call` statements are the basic access mechanism to hardware and system-related parameters that FACTS provides. Whenever a `Call` statement is invoked, the corresponding action will be executed once. The signature of a Call Statement depends upon the called function, where the return values will be supplied as facts. Listing 3 provides an overview of fact names and property keys for these system facts, whereas the interfaces for calling ScatterWeb functions are depicted in Listing 2.

When examining an application programmers view upon given hardware resources, we observed that these can be classified into two distinct groups of devices: On the one hand, some devices, namely sensors and actors, need to be able to be configured in full extend by an application. On the other hand, those devices that we subsume under the label 'peripherials' are not subject to application-level tuning, thus configuration issues can be shielded from a developer.

Listing 2: FACTS system call API.

```
1  //access to sensors
2  Call getTemperature([int res], [int mode])
3  Call getHumidity([int res], [int mode])
4  Call get3DSample([int sensitivity])
5
6  //access to actors
7  Call setLED([int mode], [int blink_times])
8
9  //access to peripheral hardware
10 Call printFact(name fact_name)
11 Call logFact(name fact_name)
12
13 //access to timer
14 Call setTimer(name t_name, int interval)
15 Call removeTimer(name t_name)
16
17 //access to functions
18 Call getRandom()
```

Listing 3: System facts that are returned when invoking system calls.

```
1  //sensor values
2  fact temperature [int value]
3  fact humidity [int value]
4  fact acceleration [int x, int y, int z]
5
6  //timer and random output
7  fact t_name
8  fact random [int value]
```

4.1.1 Access to sensors and actors

Whenever a call for sampling a sensor is issued, a fresh reading is taken. The return value will be asynchronously pushed into the fact repository as soon as the called firmware function returns. All sensors can be called without arguments. In case no explicit information is supplied else wise via context_facts, FACTS will simply default to lowest resolution and accuracy.

The ScatterWeb platform currently offers three different sensors, a temperature, humidity and an acceleration sensor,

and one LED as an actor with additional actors and actuators attachable on demand. Temperature and humidity readings can be sampled in a high or a low resolution, with a high resolution taking about four times as long. Furthermore, the humidity reading may be corrected by a temperature coefficient to gain a better quality. The accelerometer allows for setting its sensitivity according to its sampling domain, thus dependent on the application specific range of values that are expected to be measured.
The more sophisticated an application has to be about its input values to enable decisions on application-relevant states and its output values to precisely control its actors, the more important low-level configuration of the corresponding measurement hardware becomes. Therefore, it is essential to grant access to such settings all the way up to the application and not hide configuration within the middleware.

Listing 4: Context_facts for configuration.

```
1  //context_facts to configure the sensors
2  fact temperature_context [int samplefreq,
       int res, int mode]
3  fact humidity_context [int samplefreq, int
       res, int mode]
4  fact acceleration_context [int samplefreq,
       int sensitivity]
5
6  //context_fact to specify timers
7  fact timer_context [int interval, name
       fact_name]
8
9  //context_facts for peripherals
10 fact log_context [name fact_name]
11 fact print_context [name fact_name]
```

4.1.2 Access to peripheral hardware

Peripheral devices, i.e. devices which offer a kind of background service such as logging or printing debug information, should be accessible in the easiest manner possible. In contrast to the core of SANETS, the sensors and actors, explicit configuration is neither needed nor desired. For instance, there is not only no need to let applications decide on the size of swap memory for data logging to flash, it may even be harmful due to resulting memory shortage at runtime in case of overly optimistic settings.
FACTS implements mechanisms to log facts to an SD-card and to print facts for debugging purposes using the serial interface. The interface to these peripheral devices reflects the above mentioned design rationale: logging or printing using the `Call` statement is possible by specifying a name of a fact or a slot as an argument without any additional configuration options.

4.1.3 Access to system functions

On the software side, `Call` statements can be used to access a subset of the interfaces supplied by the ScatterWeb API as well as e.g. mathematical functions that are not part of RDL. In the current implementation of FACTS, calls to set and remove timers and a function call to generate a random number are implemented. This interface can be extended in case further operations are needed.
Due to the event-driven nature of FACTS, timers as a means for manual flow control take a special role during application development. Since any action has to be triggered explicitly by an event, or more precisely by the occurrence of a new or altered fact, the timer interface is the only possible source for delayed event generation. Consequently, the expiration of a timer will generate a fact with the name specified in the argument list.

4.2 Context-aware Configuration and Control

The development of the use case described in Section 3 in RDL revealed several characteristic properties of event-driven programming that can be improved to achieve a leaner implementation: For instance, the object tracking example requires a lot of adjustments to timers, sampling intervals and sensor configuration as soon as a node is going from one state to another. This resulted in separate facts describing configuration parameters and many individual statements to modify each setting accordingly, thus lengthy source code with a lot of repetitions.

Furthermore, the way a device is utilized by a programmer differs among hardware categories. Sensors on the one hand usually exhibit a periodic timing. The accelerometer has to be repetitively called to be able to measure a possible emitted activity of a tracked animal and therefore recognize its presence. Event-driven programming, as opposed to iterative programming, relies on emerging events to trigger actions. The implementation of periodic sampling hence requires manually created timer events, which leads to extra control loops in the application source code.
On the other hand, peripherals are operated based on the application semantics. Logging of acceleration facts is for instance only requested when a node is in 'detected' state, so the timing for these events is inherent to the application. In this case, not the control of application flow, but the filtering mechanisms lead to bloated source code: RDL offers no straight-forward mechanism to address a single fact, but filters facts via matching. As a consequence, all facts that currently reside in the fact repository and match the requested slot for logging will be written to flash when invoking a `Call` statement, even if they have been logged beforehand.
To overcome all these difficulties, provide an easy-to-use abstraction and enable quick, context-aware hardware reconfiguration, we added the concept of `context_facts` to the FACTS API. All configuration parameters relevant to operate sensors and actors, to define user timers or to use peripherals are assembled in particular `context_facts`. Added to the fact repository just as a regular fact, the rule engine will configure the requested resource transparent to a programmer and according to its operational mode. The advantage of using specialized facts to integrate configuration and control is that it is a simple augmentation of the framework while at the same time preserving the overall unified data abstraction. A nice side-effect is that the transmission of `context_facts` opens up a new means for local interaction between nodes, as dynamic remote tasking can be easily implemented.
Listing 5 is an excerpt of a rule-based implementation of the object tracking use case and will clarify the different usage patterns for `context_facts`. The three depicted rules implement a node's change from a state of 'aware' to 'detected', its remainder in the state 'detected' and its transition back to 'aware'. It is noteworthy to mention that with the help of `context_facts` one rule is sufficient to describe each state as well as another one to describe each state transition.

Listing 5: Excerpt from the object tracking ruleset.

```
1  //state: 0 = idle, 1 = aware, 2 = detected
2  fact fsm [state = 0]
3  fact threshold [aware = 10, detected = 20]
4  slot cur_state = {fsm state}
5
6  [...]
7
8  rule awareToDetect 95
9  <- exists {acceleration
10       <- eval ({this intensity} > {
             threshold detected})}
11 <- eval (cur_state == 1)
12 -> set cur_state = 2
13 -> retract {timer_context}
14 -> retract {acceleration_context}
15 -> retract {acceleration}
16 -> define timer_context [interval = 5000,
     fact_name = {timeout}]
17 -> define acceleration_context [samplefreq
      = 100, sensitivity = 2, operation =
     1, threshold = {threshold detected}]
18 -> define log_context [value = {
     acceleration <- eval ({this intensity}
      > {threshold detected}}]
19
20 rule detectedCorrect 94
21 <- exists {acceleration}
22 <- eval (cur_state == 2)
23 -> retract {timer_context}
24 -> retract {acceleration}
25 -> define timer_context [interval = 5000,
     fact_name = {timeout}]
26
27 rule detectedToAware 93
28 <- exists {timeout}
29 <- eval (cur_state == 2)
30 -> set cur_state = 1
31 -> retract {timeout}
32 -> retract {timer_context}
33 -> retract {log_context}
34 -> retract {acceleration_context}
35 -> define timer_context [interval = 10000,
      fact_name = {timeout}]
36 -> define acceleration_context [samplefreq
      = 1000, sensitivity = 1, operation =
     1, threshold = {threshold aware}]
```

4.2.1 Usage of context_facts to aggregate configuration parameters

The ability to reconfigure sensors and actors at runtime is mandatory to achieve acceptable application behavior. `Context_facts` for this category of devices, see listing 4, can be used to aggregate all necessary information in one fact that is well-known to the rule engine as well as to specify access times via the property "samplefreq". In case this property is omitted or set to zero, the fact serves purely as a configuration fact for subsequent `Call` statements. Whenever no explicit configuration information is passed to the rule engine when invoking a call, it will inspect the fact repository for a `context_fact` associated with the resource in question to obtain the right data resolution.
In the object tracking example, the `acceleration_context` facts specified in line 17 and 36 of Listing 5 are e.g. used to set the sensitivity of the accelerometer. The resolution of samples is increased upon the recognition of a high intensity in acceleration measurements and decreased as soon as a timeout triggers the transition back to the 'aware' state. Note that instead of retracting the old setting (line 14) and inserting the new `context_fact`, modifications to the properties via set statements will have the same effect.

4.2.2 Usage of context_facts to encapsulate control loops

When utilizing a rule-based programming language such as RDL, the execution of a rule demands for a prior event to trigger it. Implementing a periodic task therefore requests a programmer to manually craft the triggering event which is usually done with the help of a timer, thus relying on a mechanism external to the FACTS framework.
The burden of managing this manual flow control has been taken from an application developer for those resources that are subject to periodic calls. The control loop is encapsulated in the corresponding `context_fact` and supervised by the rule engine, thus hidden from a programmer. For example, the `context_fact` associated with the accelerometer allows to specify its sampling frequency. Whenever such a fact is added to the fact repository (e.g. line 17), the rule engine will internally create a timer, sample the sensor upon its expiration and renew it, and push the reading into the fact repository. Likewise, the removal of the `acceleration_context` (line 34) will automatically delete the corresponding timer and prevent further sampling.
A general abstraction to periodically generate facts is the `timer_context`. Whenever a `timer_context` fact is added to the fact repository, the rule engine will create a fact with the name fact_name after the amount of time denoted in the interval property has passed. The interplay of the second and third rule of Listing 5 nicely illustrates its usage. As long as a node is in the state 'detected', the lack of an acceleration fact satisfying the intensity condition is tolerated for 5 seconds (lines 16 and 25). With every acceleration event coming in, this timer is reseted (lines 20-25). Its expiration will generate a fact with name timeout, thus trigger the third rule that defines a longer interval of 10 seconds as a threshold for incoming acceleration facts for keeping the node in the 'aware' state.

4.2.3 Usage of context_facts for semantic timing

Logging facts to flash or printing them for debugging purposes with the `Call` statement turned out to be of a peculiar semantic. In order to prevent matching a multitude of facts, the filtering had to be very precise, an effort that felt orthogonal to its purpose. The usage of these peripherals had more a flavor of a background functionality concurrent to the application than being an active part of it. Therefore, the introduced `context_facts` for these resources overcome the challenge of what we refer to as semantic timing because they basically realize a publish and subscribe mechanism for specified facts.
The definition of the `log_context` fact in line 18 is a good example to illustrate the implementation. Its addition to the fact repository instructs the rule engine to monitor all acceleration facts that are being added or altered and log them if the specified condition is satisfied. A copy of the fact will then be stored in a buffer page, and eventually evicted to flash. This way, a time series of modifications to a fact will also result in a time series of logged facts on the SD-card, a circumstance that is valuable to interpret system behavior

when debugging or data evolution when recording values. It is noteworthy to mention that this way of subscribing for facts can be really interpreted as an action concurrent to application execution. Imagine the `log_context` fact expressing its interest for acceleration facts without any filtering. Although the rule that issues the `log_context` fact will only trigger on acceleration facts of a high intensity, subsequent logging will not. This can be realized since conditions for initialization and execution of logging are simply decoupled, thus application flow and background activity can better be separated.

Listing 6: Excerpt II from the object tracking ruleset.

```
1  [...]
2
3  rule idleToAwarePending 100
4  <- exists {acceleration_context}
5  <- eval (cur_state == 0)
6  -> call setTimer({timeout}, 10000)
7
8  rule timedOutAware 99
9  <- exists {timeout}
10 <- eval (cur_state == 0)
11 -> retract {timeout}
12 -> retract {acceleration_context}
13
14 rule idleToAware 98
15 <- exists {acceleration}
16 <- eval (cur_state == 0)
17 -> call removeTimer({timeout})
18 -> set cur_state = 1
19 -> send 0 100 {acceleration_context}
20 -> retract {acceleration}
21
22 [...]
```

Listing 7: Excerpt from the system ruleset for the accelerometer.

```
1
2  rule removeAccSensorContext
3  <- exists {acceleration_context}
4  <- eval ((count {acceleration_context}) >
       1)
5  -> retract {acceleration_context
6         <- eval ({this modified} == true)}
7
8  [...]
9
10 rule checkConditionTrue
11 <- exists {acceleration}
12 <- eval ({acceleration_context operation}
       == 1)
13 <- eval ({tmp_acc intensity} > {
       acceleration_context threshold})
14 -> retract {acceleration}
15 -> define acceleration [intensity = {
       tmp_acc intensity}]
16 -> retract {tmp_acc}
```

4.2.4 Usage of context_facts for remote tasking

Syntactically, a `context_fact` is a regular fact, tagged with system information upon creation and altering and subject to any manipulation process that RDL offers. When turning from a node-local point of view on fact processing to a global viewpoint on the network, the transmission of a `context_fact` can be used for remote tasking. Upon reception, the rule engine of the node will add the fact to the fact repository and automatically behave as if the fact had been created locally.
Listing 6 depicts three rules of the object tracking use case that make use of exactly this behavior. In idle state, a node is set to low-power mode performing no sensing at all. The reception of an `acceleration_context` fact (line 4) triggers sampling the accelerometer at a medium data rate and sets a timer which will retract the `context_fact` in case of a false alarm. If an acceleration intensity is measured that satisfies the transition into the aware state, the `context_fact` is simply brodcasted to neighboring nodes (line 13) to trigger their observation of the vicinity as well.
On the downside of dynamically tasking neighboring nodes, the sudden appearance of multiple `context_facts` for one resource can lead to unexpected behavior. While having e.g. more than one `log_context` fact is not problematic and may well be intended, a variety of configurations for sensors lead to race conditions. We address this problem in the next section with the help of system rulesets.

4.3 System Rulesets

System rulesets are a powerful concept to outsource low-level filtering and fact processing shared by a variety of tasks from the actual implementation of an application. This way, we make use of the modularity of rules and ruleset to achieve a better encapsulation of concerns. Tagged with the highest execution priority, the rules of a system ruleset will automatically be evaluated first. Inspired by the way system libraries supply services to support application development, system ruleset can as well be linked to applications on demand to avoid exhaustive memory usage and superfluous rule evaluation.
In the current implementation of FACTS, we provide one system ruleset per sensor since our primary concern had been to prevent the fact repository from thrashing from redundant, system-generated facts during sampling. In the future, system ruleset to e.g. support more elaborate neighborhood interaction or data aggregation patterns will be added.
Listing 7 depicts two rules that are part of the system ruleset for managing accleration sensor related filtering and control. The first rule tackles the problems that multiple `acceleration_context` facts on a single node may impose, a situation that e.g. occurs when a node receives `context_facts` from more than one neighboring node. The policy is simply to evict the newest `acceleration_context` to preserve the current application context.
We observed that when working with the acceleration sensor, the raw acceleration readings are often condensed into a value reflecting the amount of change between two consecutive samples [8]. Due to space limitation we omitted its rule-based implementation. The second rule however controls whether such a processed acceleration fact is handed up to the application. To transparently realize a comparison between this fact and the current `acceleration_context`

with multiple operators, we tagged the `context_fact` with two additional properties not part of the FACTS API, an operation and a threshold property, see Listing 5 (line 17 and 36). The operation property encodes the type of comparison operation that is used to compare acceleration intensity and predefined threshold, with opcodes being specified in the corresponding ruleset. This additional tagging of a `context_fact` is perfectly feasible since it makes simply use of matching mechanisms that FACTS relies on. Again, the addition of `system rulesets` is no technique outside of the prior FACTS framework but rather uses its inherent building blocks to provide further improvements. Routines to retract unused system-generated facts thus low-level garbage collection as well as rules for managing cached data samples are currently part of the imlemented `system rulesets`.

5. RELATED WORK

A multitude of domain-specific programming abstractions and middleware platforms have been proposed to provide abstractions for sensor network tasking. A popular model to shield a developer from underlying network interaction is to offer a macroprogramming approach for handling distributed data streams. Representatives such as Regiment [5] or TinyDB [4] lack mechanisms for node level configuration thus explicit configuration and control is not supported.
Milan [3] is a platform that has been designed to support the implementation of applications with QoS requirements and provides means for meeting them based on sensor fusion. Instead of requesting a programmer to individually configure a sensor dependent on the application state as done in FACTS, the information on application QoS demands has to be passed to MILAN via *state-based variable requirements graphs* and *sensor QoS graphs*. MILAN will then determine which resources within a network will satisfy the given requirements and configure the network accordingly. Configuration and control of additional resources are not subject to MILAN management.
Applications built on top of the TeenyLime [1] middleware can benefit from transparent access to data of one-hop neighbors by relying on a shared tuple space. To make resources such as sensors available, a node has to put a *capability tuples* into the tuple space which can then be accessed via matching. Since data will be automatically transmitted to a node in case a remote sample has been requested, elaborate mechanisms have been added to ensure a global, user-defined data freshness value as well as to prevent a.g. excessive sampling of sensors. In contrast, communication in FACTS has always to be explicitly programmed by an application developer, but sampling frequencies may be set in a device-dependent manner.

6. CONCLUSIONS

In this paper we presented a lean augmentation of the FACTS middleware framework to efficiently realize low-level system access. Based on the current context and development state of an application, a developer may choose from a set of powerful tools to ensure the correct granularity of configuration and control to task SANETs. The provided abstractions of `context_facts` and `system rulesets` nicely integrate into the framework by encapsulation of common concerns in framework-inherent building blocks and offer a straightforward way to write concise application-level code.

7. REFERENCES

[1] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming wireless sensor networks with the teenylime middleware. In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007)*, Newport Beach (CA, USA), November 2007.

[2] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, G. Zhou, J. Hui, and B. Krogh. VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance. *ACM Transactions on Sensor Networks (TOSN)*, 2(1):1–38, Feb. 2006.

[3] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *Network Magazine Special Issue, IEEE*, 18(1):6–14, 2004.

[4] Madden, Franklin, Hellerstein, and Hong. Tinydb: An acquisitional query processing system for sensor networks. volume 30, pages 122–173, 2005.

[5] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In T. F. Abdelzaher, L. J. Guibas, and M. Welsh, editors, *IPSN*, pages 489–498. ACM, 2007.

[6] J. Schiller, A. Liers, and H. Ritter. ScatterWeb: A Wireless Sensornet Platform for Research and Teaching. *Computer Communications*, 28:1545–1551, Apr. 2005.

[7] K. Terfloth, G. Wittenburg, and J. Schiller. FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks. In *Proceedings of the First International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE'06)*, New Delhi, India, Jan. 2006.

[8] G. Wittenburg, K. Terfloth, F. L. Villafuerte, T. Naumowicz, H. Ritter, and J. H. Schiller. Fence monitoring - experimental evaluation of a use case for wireless sensor networks. In K. Langendoen and T. Voigt, editors, *Proceedings of the Fourth European Conference on Wireless Sensor Networks (EWSN '07)*, pages 163–178, Delft, The Netherlands, Jan. 2007.