FACTS – A Rule-based Middleware Architecture for Wireless Sensor Networks

Kirsten Terfloth
Freie Universität Berlin
Takustr. 9
14195 Berlin
+49 (0) 30 838 75211
terfloth@inf.fu-berlin.de

Georg Wittenburg
Freie Universität Berlin
Takustr. 9
14195 Berlin
+49 (0) 30 20256233
wittenbu@inf.fu-berlin.de

Jochen Schiller
Freie Universität Berlin
Takustr. 9
14195 Berlin
+49 (0) 30 838 75211
schiller@inf.fu-berlin.de

ABSTRACT
Introducing a middleware abstraction layer into wireless sensor networks is a widely accepted solution to facilitate application programming and allow network organization.
In this paper, we argue that although an event-based approach is the most obvious solution, it also provides the most natural way to address software development in wireless sensor networks. As a proof of concept, we introduce FACTS, a very flexible middleware framework able to provide support for a wide range of different applications. The objective is to combine advantages of event-centric processing and rule-based execution while preserving low resource usage.

Keywords
Wireless sensor networks, middleware, event-centric architecture, rule-based language, qualitative simulation

1. INTRODUCTION
Wireless sensor networks (WSN) comprise a variety of features that make software development in this domain rather challenging. Dealing with embedded devices in large numbers, each of which is prone to error and very restricted in terms of energy, memory and processing power introduces a burden upon the application development. Moreover, the distribution of nodes and their shared communication medium call for multi-hop routing algorithms and distributed coordination among them. The introduction of a middleware layer can alleviate problems arising from both complex communication and data sharing methods as well as those that originate at the underlying software layer.
Since the deployment of a middleware layer inevitably consumes memory, a decision whether to use this kind of abstraction has to be deliberated carefully. The more generic such a middleware layer, the more likely it is to be used in diverse application scenarios. Furthermore, a clear abstraction model able to combine benefits for supporting both distribution issues and embedded programming is needed. A number of solutions have already been suggested, ranging from tiny enhancements for programming [1] to full-fledged middleware architectures with rich sets of services [2, 6, 9].
In the latest discussions it has been argued, that event-driven programming is rather complicated from an application programmers' point of view. Asynchronous wake-ups related to the occurrence of state changes, the need for non-blocking calls to enable execution and coordination of multiple events, and the necessary means to maintain state seem backward to 'usual' imperative programming. On the other hand, an event-driven paradigm suites the application domain best: sensor nodes are supposed to detect certain changes in their environment, react to them appropriately and remain in low-power mode in case of absence of events or tasks. Hence, a triggered action, thus a push mechanism, is a more natural way to think of programming for sensor networks.
The middleware architecture we introduce in this paper provides a powerful mental model that emphasizes the use of trigger mechanisms and actions. Combining the advantages of virtual machines, grouping facilities and modularity-based approaches as described in [10], and inspired by expert-systems [12], our main abstraction are *rules*, *facts* and *functions*. In this context information – which is everything ranging from sensor readings to temporary variables – is represented as facts, which are stored in a *fact repository* and processed by rules. In other words, rules encode the information processing relevant to a specific application and are written in a high-level language consisting of the triggering conditions and the actions to be taken accordingly. In contrast to inference engines using backward chaining, our *rule engine*, which is the scheduling entity for rules, uses forward chaining in order to provide event-like semantics. Rules may also call functions that hook into the firmware or operating system of the sensor node and perform resource critical operations in a fast and memory-efficient manner.
Facts, rules and functions are local to each node of the sensor network and each node runs its own rule engine. However, facts are also used as the key abstraction for transmitting information from one node to another, to the entire sensor network, or to a distinct subgroup. Hence, one can think of it as a node sending one of the facts from its

own fact repository to another node, which in turn adds it to its fact repository. Due to the reception of the fact on its radio interface, the middleware is aware of not being the owner or originator of this fact. However, there is no further distinction made between locally created facts and those received from other nodes over the network.
The rest of this paper is organized as follows: First, we estimate FACTS concepts in the context of current research activities. Section 3 describes the main architecture of FACTS in detail and discusses design considerations, before introducing the prototype implementation in Section 4. A coverage algorithm is given in our rule language to present the expressiveness of our programming abstraction in Section 5, and give a first impression of the overall system in practical use. A conclusion and outlook on future work concludes this paper and once again summarizes the contributions of our work.

2. RELATED WORK

A number of approaches resemble FACTS in certain aspects. The SWARMS project [13] makes use of a distributed virtual shared information space (dvsIS), an adaptation of the Linda tuplespace, to share global state among all nodes. The idea of SWARMS is to provide global coordination, having the sensor nodes of the network behave like members of a swarm. Elements of this space are qualified by XML tags, and are distributed among swarm members. Our fact repository is implemented to enable distributed shared memory, so applications may use this in case they can benefit from it. Otherwise, it functions as local memory for state, data and variables, and is thus the central coordination entity for nodes. This concept of variable usage is a lot more flexible, since it allows for application specific tuning instead of predefined functionality.
The Generic Role Assignment (GRA) project [4] makes use of rules to encode node behavior. An algorithm is implemented by specifying a set of roles a node can assign to itself depending on its local state and that of neighboring nodes. Therefore, a change in state may inherently result in a re-evaluation process of roles of a potentially large set of nodes, and hence lead to massive communication between nodes for coordination. Our approach also relies on stateful information, but decisions on event occurrence or state changes have to be implemented by the programmer explicitly, leading to a more controllable communication pattern. Although GRA allows a higher abstraction from communication issues, we believe that the domain of WSN does not allow making such subsumptions for developers.
The introduction of new languages to implement applications for WSN can be widely observed [3, 7, 11]. Design goals combine offering a high-level language construction tailored to domain specific needs and allowing interpreted, dense byte-code to be deployed on the nodes. Furthermore, interpretation of code may serve in acquiring modularity of software components.

3. FACTS ARCHITECTURE

The architecture of FACTS will be introduced in the following pages. To emphasize the issues that were considered during the construction process, they will be briefly presented in a motivating section.

3.1 Motivation

The main criteria for the design of our middleware architecture are summarized in the following key points:

- Event-centric architecture
- Rule-based language capturing trigger/action semantics
- Minimalistic architecture, able to be enhanced at runtime according to application-specific services
- Support for distributed shared memory to enable grouping algorithms
- Support for cross-layer optimization

To reflect the event-centric domain of networked sensors and benefit from the inherent data compression of an event concept, the middleware design uses the primary abstraction of events which trigger actions. The overall system can therefore be implemented in a resource-aware manner, allowing nodes to switch to low-power mode in case of absence of events. The formalization of rules is a natural way to express actions triggered by changes either in a node's environment or internal state, or by any combination of both. Therefore we chose to make explicit use of these semantics and defined a suitable language that is concise and powerful, but also small in terms of memory consumption.
As the goal is to allow sets of basic services, depending on the envisioned application domain, to be implemented and added to the middleware on demand. Hence, a modular design to obtain a highly flexible framework is also a key design aspect. Furthermore, the ability to easily share data among nodes for coordination algorithms has been proven useful in WSNs [5]. Our fact repository with its ability to act as a distributed shared memory supports software relying on any kind of grouping or clustering mechanism. These features are not explicitly provided by FACTS to prevent the pollution of memory with unnecessary services, but can be added in a set of a few rules.
Not only data sharing among different nodes of a network can be realized through the use of the fact repository as central data entity on a node. Algorithms operating on different layers according to the ISO/OSI layering model can also exchange information to use it as means to

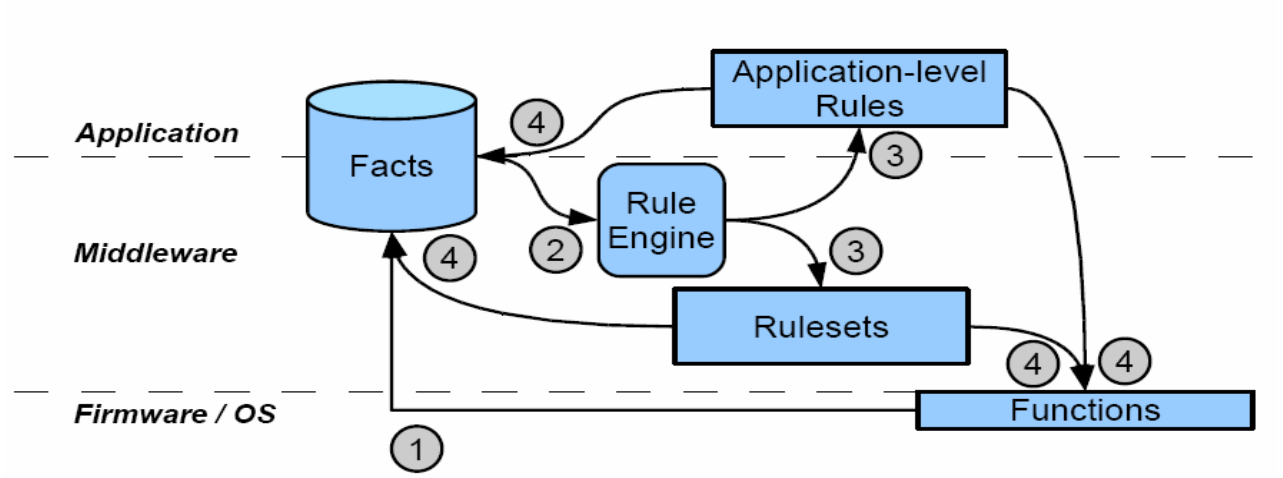

Figure 1: Component diagram of FACTS architecture: Low-level events start execution by creation of new facts (1), which trigger the rule engine (2) to match rules against fact repository (3) and eventually fires rules (4).

coordinate and tune themselves, enabling network optimization based on cross-layering.

The middleware architecture of FACTS as shown in Figure 1 has been designed to provide a highly flexible framework to cope with the limited resources inherent to the application domain of wireless sensor networks, but nevertheless equip a developer with a clear mental model. The fundamental concepts are introduced in following sections.

3.2 Rules

Rules are used to express algorithms and reactions to external events in the system. A rule is a named structure containing both a set of conditions and an ordered list of statements. A rule fires, i.e. the rule engine executes the statements belonging to the rule, if all conditions evaluate as true according to the facts in the fact repository. More precisely, a rule fires if all the conditions are true and at least one of the facts mentioned in these conditions is tagged as modified. This ensures that the system processes changes in the fact repository in an event-like manner. Furthermore, a rule also has a priority which defines exactly when during one complete run of the rule engine the conditions of the rule are checked against the fact repository, and based on this it is decided whether the rule should fire or not. Multiple rules can have the same priority, but for these rules the system does not provide a guarantee on the order in which they are executed.

Conditions have one of two forms:

Exists: This condition checks whether one particular fact exists in the fact repository.

Eval: An eval condition checks, whether a boolean expression is true given the data found in the fact repository. Statements modify the fact repository or generally interact with the rest of the system.

Available statements include:

Define: Adds a new fact to the local fact repository and initializes its properties.

Retract: Removes one or more facts from the local fact repository.

Set: Changes the value an existing property of one or more facts in the local fact repository or adds a new property.

Send: Transmits one or more facts in the local fact repository to other nodes in the sensor network.

Call: Calls a function implemented in native code and made available by the underlying software layer.

An example on how to use these operators to implement a typical algorithm for WSNs is given in Section 5.

3.3 Facts

Facts are the central means of representing any kind of data in the system. They are structured as a named set of key-value tuples. Multiple facts with the same name may be present in each local fact repository without disturbing the system. Keys are unique for each facts, values typed. A specific key-value tuple is called a *property* of a fact, with available types of values being *bool*, *int*, *float*, and *string*. Every fact has a few predefined properties which are updated by the system and available to the programmer as read-only values. More precisely, the keys and types of these hard encoded properties are:

(int) owner: The network wide unique ID of the sensor node that was the last to modify this fact by either creating it, adding a new property, or modifying an existing property.

(int) time: The time at which the fact was last modified. Note that this merely is the perception of the current time

of the modifying node, which may well be out of sync with the rest of the sensor network or otherwise inaccurate.
(string) id: The network wide unique ID of this fact. It is implemented as the dot-separated concatenation of the ID of the owning sensor node (which is unique in the sensor network) and the time of last modification (which is unique on the local node, that sets itself as the owner upon modifying the fact in question).
(bool) modified: This boolean flag indicates whether a fact has been modified by either another rule or some external system-generated event during or since the last run of the rule engine. As rules only fire when at least one of the facts referenced in their conditions has been modified, this property is useful for working on only those facts out of a potentially larger set that have caused the rule to fire and hence might require processing. The modified flag of all facts is cleared every time an entire run of the rule engine has been completed successfully.

As facts are dynamically added to the fact repository at runtime and as only facts marked as modified trigger the execution of rules, modified facts appear as events in the system to the application-level programmer, thus allowing for event-centric programming. Furthermore, facts are also the central means of transmitting information between nodes of the sensor network: The `Send` statement in a rule allows for one or multiple facts to be transmitted to another node or to be broadcasted to the entire network, possibly utilizing multi-hop forwarding by other nodes.

3.4 Functions

A function is a chunk of machine code that interacts with the firmware or operating system of the sensor node, and provides an interface that can be invoked by the rule engine. As such, its main purpose is to provide hooks for the rules to interact with the software layer below the middleware and to allow for efficient implementation of algorithms featuring critical performance considerations.

3.5 Derived Concepts

Complementing the basic components of the distributed expert system, the following concepts are helpful to fully understand and communicate about the system:

3.5.1 Slots

A slot is the primary means for addressing facts and their property values in the fact repository. It is a tuple consisting of two patterns, one identifying the fact and one identifying the property key, and a list of conditions that further specify which subset of the fact repository is to be addressed. The property key pattern may be omitted if only the fact itself, rather than one of its properties, is of interest.

3.5.2 Rulesets

A ruleset is the construct equivalent to a component in other middleware architectures. It is a set of rules and related facts that together provide a certain services in the system. Rulesets have a well-defined interface in terms of a set of facts that trigger the execution of their rules. They encapsulate locally used rules and facts in their own namespace which is implemented as dot-separated prefix to the name of the fact it contains.
Rulesets may explicitly provide services as identified by a well-known descriptor, or require them to be present on a local system when being installed. This allows for the construction of a dependency graph at compile time and possibly for automatic loading of rulesets at runtime. On the downside, one has to be aware of possible excessive resource usage in case of automated functions, a drawback that should be investigated extensively at design time.
Experiences gained by implementing a few common algorithms typical for wireless sensor networks show that a ruleset is typically in the magnitude of ten to twenty rules and a similar amount of facts. Of course, these numbers are merely rough estimates and vary depending on the complexity of the functionality to be implemented. The possibility to add such ruleset-like library functions to the system according to application needs once again emphasizes the powerful and adaptable design of FACTS.

3.5.3 Globally Shared Information Space

Facts have an unique ID that is the concatenation of the ID of the owning node and the time when the fact was last modified. Only one fact can be modified on a node at any given time. The fact repository may thus contain various facts from different nodes, being the knowledge base of the node owning it. This knowledge is composed of information about its current state, events it has spotted in a certain time interval or communicated details of neighboring or potentially reachable nodes in the network.

3.6 Design Details and Considerations

The concepts as described above were only rough sketches when the work of formalization was begun. Several decisions were made during the design phase of the project, which eventually led to the final semantics of our system. This section lists important details of our design and the considerations details that led to these decisions.

3.6.1 Sets of Facts

As facts are addressed by their name – which is not required to be unique – evaluating a slot against the fact repository may result in a set of multiple matching facts. Hence, a condition or statement that takes a slot as a parameter may either process a single fact or a set of facts depending on the content of the fact repository at runtime.

For a condition to return true when evaluated against a given fact repository, it is sufficient if at least one of the facts matched by the slot satisfies the constraints stated in the condition. A statement however will be executed separately for every single fact matched by the slot, which will lead to multiple executions of the same rule. In case of a statement containing multiple slots, the statement is applied sequentially to all possible combinations of the matching facts, i.e. the cross product of the respective sets of facts. If only one specific fact is to be processed, this can either be achieved by carefully naming the facts, or by providing more specific constraints in the form of conditions as additional parameter of the slot as explained in the following section.

3.6.2 Filtering Facts for Processing by a Statement

Apart from addressing a fact by its name, a slot is frequently required to be more specific about exactly which fact from a potentially large set of matching facts to work with. In order to do so, a slot can filter the set of all facts with matching names by providing further constraints on the required values of properties of selected facts. This is done in the form of a list of conditions that can be specified as additional parameter for slot refinement. Obviously, the result includes only that subset of facts where all conditions on slot parameterization are met. It is up to the application-level programmer to ensure that the properties of facts differ enough for selection to isolate a specific single fact in case this is required by the application. The system supports this by providing an unique ID for each fact in the fact repository and therefore making any fact directly accessible via the read-only id property.

An alternative design would have been to implement *implicit* filtering of the facts based on the conditions of the rule in question. While this would allow for a simpler syntax, there are several disadvantages:

- Statements may access facts that are not referenced in the conditions. Adding conditions for filtering purposes alone would bloat the application-level code.
- Questions would arise on the proper order concerning statement processing: All statements could be executed sequentially for all matching facts (horizontal execution), but with the same logical implication each single statement could also be applied to each matching fact before executing the following (vertical execution). None of these two options seems intuitive enough to be acceptable by an application-level programmer or preferable due to its inherent properties.
- Statements would have different semantics depending on which rule they appear in. This would not only be confusing but also violate the idea of strict decoupling of conditions and statements within a rule, as detailed in the next section.

3.6.3 Separation of Conditions and Statements

Even when associated with the same rule, conditions and statements are clearly separate entities. Conditions are specified to only take care of firing a rule and should not result in unintentional side effects. On the other hand, statements only alter the fact repository or interact with the system, thus can be totally ignored by the rule engine when determining whether to fire rules. Clearly, statements do have exactly the same semantics independent of the calling rule. However, this also implies that the filtering of exactly which facts to process needs to be done for each statement separately, and hence is independent of the conditions of the rule.

3.6.4 Adjusting Ownership of Modified Facts

Whenever a node changes the properties of a local fact, the owner property of the fact is set to the ID of modifying node. The goal is to keep the global namespace of facts intact and to ensure that sensor readings processed within the sensor network are clearly marked as such. If the original fact is to be preserved while processing, a copy needs to be made beforehand. The copy is then owned by the local node and can therefore safely be modified. In case only changes to facts owned by the local node are intended, one has to add a filtering condition to slots which state that only facts whose owner property matches the ID of the local node are to be processed.

An alternative solution proposes facts to be owned by the local node exclusively, i.e. independent of any filters. Updates would then result in changes just in case the current executing entity is also the owner of the fact and otherwise leave the fact untouched as read-only. It turned out that while allowing for the same functionality to be implemented, these semantics resulted in bloated code because of bad interaction with sending facts across the network: In this case, facts can be seen as packets, and packet properties, thus facts properties, needed to be updated slightly for each hop they travel on the network. Having to make a copy before being able to process a packet would not only waste memory but also result in unreadable code.

3.6.5 Usage of Local Variables

Unlike JESS [12], a reference expert system we examined tor language formalization, our system does not support the notion of local variables to which a specific fact can be bound within a rule. We consider binding facts to variables at runtime to be too expensive in terms of memory usage for an embedded system. As proven above the syntax of

filtering on slots is able to provide the same functionilty, while lowering system overhead for memory management. Due to this optimization it will be possible to give bounds on memory consumption of a future implementation even at compile time.

4. PROTOTYPE IMPLEMENTATION

FACTS as introduced in the previous sections was first implemented as purely functional prototype in the Haskell programming language. The following section explain the reasons why this somewhat unusual approach was taken, gives an overview of the implementation and presents relevant code fragments.

4.1 Rationale

As [14] points out, a functional design or a prototype can be most useful even if the ultimate goal is an imperative solution. In our case the advantages were as follows:

- Initially, the basic concepts were not well understood beyond traditional event-centric architectures and their exact semantics changed rapidly while new requirements and interdependencies were discovered. During this phase of rapid iterative prototyping, the emphasis on concise functional definitions helped the project to stay coherent.
- The definitions of functions obtained from the prototype can serve as formal specification of the system. Also, the definitions of the data types can be used as basis for a grammar. Eventually this approach can lead to the construction of a compiler for this grammar able to translate human readable rule definitions into an intermediate format or byte-code, suitable for deployment in the sensor network.
- Higher order function and their capability of using functions as parameters makes the code base very compact and easy to maintain, while at the same time preserving type safety.

The availability of our functional prototype allows to run test cases and check the semantics of the system very early in the development process. The result is a smooth codebase without known inconsistencies or dead code. Furthermore, the codebase is perfect in the respect that nothing can we removed without losing functionality.

4.2 Overview

The core of our system is implemented as a Haskell module. Its public interface contains constructors for the condition and statement primitives and functions to create rule, fact and function entities as well as slots and rulesets. For evaluation purposes we also implemented functions to construct a sensor node and set up a sensor network. and based on these run a simulation. Using these mechanisms a core set of rulesets has been implemented as additional Haskell modules and is available to be used in test runs of the system.

Following the functional paradigm, the simulation runs by iteratively transforming the current state of the sensor network – including all nodes and their respective rule and fact repositories – into the subsequent state. For all nodes the conditions of their local rules are checked against the fact repositories and the statements are executed if the given facts suffice for the rule to fire. In order to implement unique IDs of facts the simulation environment provides a timer counter that is increment whenever a fact is modified and set to an steadily increasing well known value after a complete run of the simulated rule engines on all nodes has been completed. The current notion of time of the sensor nodes is thus known for each simulation step which allows for the simulation to supports the injection of facts to the fact repositories of one specific are all sensor nodes. As external events appear to the rule engines as new facts in their repositories, the injection method can be used to simulate sensor readings at certain points in time during the deployment of the sensor network.

4.3 Relevant Code Fragments

Listing 1 is a shortened version of the main “loop” of the functional simulation.

```
1 processState :: [Event] -> State -> State
2 processState events state =
3   (State (step + 1) nextStepTime
4   (Network (map cleanNode newNodes)))
5   where
6     nextStepTime =
7       time + simulationStepTime
8     (State step time network) =
9       state (State _ newTime
10    (Network newNodes)) =
11      processNetwork (processEvents
12      state currentEvents) network
13    currentEvents =
14      filter (\(Event eventStep _ _) ->
15      (eventStep 15 == step)) events
```

Listing 1: Main loop of functional simulation

A simulation step is broken down into several operations: In lines 13 to 15 the events for the current simulation step are filtered out of the global event list. Lines 10 to 12 first process these events by updating the state of the network accordingly, The new state is then used as input for the central processing of the network in the same line. It results in a new list of nodes the facts of which have their `modified` property cleaned up in line 4 as a final

operation. Together with the calculation of the time for the next simulation step in lines 6 and 7 this concludes the transformation of the current state.
Listing 2 shows the complete logic that decides whether to fire a rule or not and updates the simulated state of the sensor network accordingly.

```
1 applyRule :: State -> MAC -> Rule -> State
2 applyRule state mac
3   (Rule name _ conditions statements)
4   | oneFactIsModified &&
5     allConditionsAreTrue =
6     foldl (\state -> applyStatement state
7     mac) state statements
8   | otherwise = state
9   where
10    oneFactIsModified = or (map
11      (checkFacts facts) conditions)
12    allConditionsAreTrue = and (map
13      (checkCondition facts) conditions)
14    facts = getFacts state mac
```

Listing 2: Determination of rule execution

Taking the MAC address as ID of the current sensor node and the rule to be applied to its fact repository as additional parameters, this function can be broken down into the following operations: As can be seen in lines 4 and 5, a rule fires only if at least one fact is tagged as modified and all conditions of the rule evaluate as true. If this is the case, a rule is applied by folding its statements into the current state of the simulation in lines 6 and 7, otherwise the state is returned unchanged in line 8 as the rule did not fire. The calculations whether a fact referenced in the conditions has been modified and whether all conditions are true state lines 10 to 13 respectively.

5. EXAMPLE: COVERAGE

The goal of the coverage algorithm in a wireless sensor network scenario is to determine which areas of a geographic region are covered by the sensor network, and where redundant information can be gained. The collected information may be used to selectively power down sensor nodes in order to extend the total lifetime of the sensor network, a situation applying usually to densely deployed networks.
A partial listing of the rule-based implementation of the coverage algorithm is given in listing 3. Following name of the rule, conditions are listed prefixed with "`<-`" and statements with "`->`".

```
1  sendRange
2  <- Exists Timer.expiredSlot
3  -> Retract Timer.expiredSlot
4  -> Define "rangeFact"
5  -> Set ("rangeFact" "xMin")
6       (posXSlot - System.txRadiusSlot)
7  -> Set ("rangeFact" "xMax")
8       (posXSlot + System.txRadiusSlot)
9  -> Set ("rangeFact" "yMin")
10      (posYSlot -  System.txRadiusSlot)
11 -> Set ("rangeFact" "yMax")
12      (posYSlot + System.txRadiusSlot)
13 -> Send 0 System.txPowerSlot
14      ("rangeFact" [(("rangeFact" "owner")
15      == nodeIDSlot)])
16 -> Define "rangeSendFact"
17
18 xyMinCovered
19 <- Exists "rangeSendFact"
20 <- Eval ((posXSlot - System.txRadiusSlot)
21      < ("rangeFact" "xMin"))
22 <- Eval ((posYSlot - System.txRadiusSlot)
23      < ("rangeFact" "yMin"))
24 -> Define "xyMinCoveredFact"
25
26 determineCoverage
27 <- Exists xyMinCoveredFact"
28 <- Exists "xMaxYMinCoveredFact"
29 <- Exists "xyMaxCoveredFact"
30 <- Exists "xMinYMmaxCoveredFact"
31 -> Define "coveredFact"
```

Listing 3: Coverage algorithm in rules

As a first step and firing when a timer has expired, the `sendRange`-rule removes the timer fact that caused it to fire from the fact repository in line 3. It then proceeds to calculate the range it expects to cover and stores this information in a fact in lines 4 to 12. Note that establishing the position of the node is not in the scope of the coverage algorithm. In lines 13 to 15 the newly created `rangeFact` is broadcasted to the neighboring nodes. Each node has to take care however, to only send the fact that it created locally as otherwise it would re-broadcast the equally named facts that it received from its neighbors. Finally, the node changes its state by defining a fact which indicates that range information has been sent in line 16.

In the next stage, the node then waits for the `rangeFacts` of its neighbors. Upon reception of a matching fact, the `xyMinCovered`-rule inspects the data in lines 20 to 23 and fires if the covered area as reported by the fact overlaps with its own. The result is stored in the `xyMinCoveredFact`. Similar rules for the `xMaxYMinCoveredFact`, `xyMaxCoveredFact` and `xMinYMmaxCoveredFact` have been omitted for brevity.
Finally as a last step, the `determineCoverage`-rule checks whether all four sides of a nodes area are covered by other nodes in lines 27 to 30, and if this is the case, stores this information in the `coveredFact`. After this process has been completed for all nodes, each node knows whether it is the only node of sensor network to cover one particular geographic region or not and can act accordingly in the future.

The example of the coverage algorithm illustrates how our middleware provides intuitive event-like semantics and abstraction from low-level communication details. Furthermore, it shows how remote data transparently becomes available for local processing while still preserving the semantics of global adressability.

6. CONCLUSION

In this paper we presented the fundamental concepts and the prototype implementation of our middleware approach FACTS, which is able to provide a highly flexible framework for applications in wireless sensor networks. The goal is to alleviate challenges in programming arising from the underlying embedded hardware, asynchronous event-handling, and distribution issues of sensor nodes by introducing a rule-based programming environment.
To determine the state of each single node, as well as to coordinate groups of dedicated nodes in a simple way, FACTS uses a single data management facility, the fact repository. This may serve as distributed shared memory, as well as to maintain state and temporal data. Rules can be specified to implement algorithms and take care of processing of sensor data. With its modular design the overall system is aimed to be especially suitable for the restricted resources of wireless sensor networks and allows for application specific adaptations, envisioned to be performed even once deployed.

7. FUTURE WORK

After establishing the semantics of the FACTS system and small-scale qualitative simulations, there are two major directions in which to proceed:

On the implementation side, we will replace our functional prototype and custom made simulation environment with an imperative implementation that can run on both more widely used network simulators and eventually on a real-world sensor network platform. We plan to enrich our current qualitative data with quantitative measurements and finally verify our findings in real-world scenarios.

On the application-level side, we plan to extend to functionality of our middleware by implementing more algorithms and services commonly used in wireless sensor networks in the form of rulesets. This will lead to questions about the dependencies between rulesets and their interactions at runtime. We envision that rulesets will eventually be used as drop-in components for sensor networks that extend the capabilities of a deployed wireless sensor network while making these advantages transparently available to the application that runs on the middleware.

8. REFERENCES

[1] A. Dunkels, O. Schmidt and T. Voigt. Using Protothreads for Sensor Node Programming. In *REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[2] B. M. Blum, P. Nagaraddi, A. Wood, T. F. Abdelzaher, S. Son and J. A. Stankovic. An Entity Maintenance and Connection Service for Sensor Networks. The First International Conference on Mobile Systems, Applications, and Services (MOBISYS `03), California, May 2003.

[3] P. Lewis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS-X*, San Jose, USA, October 2002.

[4] K. Römer, C. Frank P. M. Marron and C. Becker. Generic Role Assignment for Wireless Sensor Networks. In *ACM SIGOPS European Workshop 2004*, Leuven, Belgium, 2004.

[5] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *ACM MobiSys 2004*, Boston, USA, June 2004.

[6] S. R. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *OSDI 2002*, Boston, USA, December 2002.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler. The nesC language: A holistic approach to networked embedded systems. In:*Programming Language Design and Implementation (PLDI)*, June 2003.

[8] K. Römer and F. Mattern: Event-Based Systems for Detecting Real-World States with Sensor Networks: A Critical Analysis. In *DEST Workshop on Signal Processing in Wireless Sensor Networks at ISSNIP*, Melbourne, Australia, December 2004.

[9] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic Parallel Sensor Systems. In *ACM SIGPLAN*, San Diego, USA, June 2003.

[10] K. Terfloth and J. Schiller. Driving Forces behind Middleware Concepts for Wireless Sensor Networks, In *The REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, June 2005.

[11] O. Kasten, K. Römer: Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines, In I*EEE/ACM International Conference on Information Processing in Sensor Networks (IPSN)*, Los Angeles, USA, April 2005.

[12] Jess, the Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/jess/

[13] Carsten Buschmann, Stefan Fischer, Norbert Luttenberger and Florian Reuter: Middleware for Swarm-Like Collections of Devices, In *IEEE Pervasive Computing Magazine*, Vol. 2, No. 4, 2003.

[14] S. Thompson: Haskell – The Craft of Functional Programming. Addison-Wesley, Second Edition, 1999.