REED: Flexible rule based programming of wireless sensor networks at runtime.

Fei, X. and Magill, E.

Author post-print (accepted) deposited in CURVE April 2013

Original citation & hyperlink:
Fei, X. and Magill, E. (2012) REED: Flexible rule based programming of wireless sensor networks at runtime. *Computer Networks*, volume 56 (14): 3287–3299.
http://dx.doi.org/10.1016/j.comnet.2012.06.004

REED: Flexible Rule Based Programming of Wireless Sensor Networks at Runtime

Xiang Fei
Department of Computing
Coventry University
Coventry, UK
x.fei@coventry.ac.uk

Evan Magill
Department of Computing Science and Mathematics,
University of Stirling, UK.
ehm@cs.stir.ac.uk

Abstract

Wireless Sensor Networks (WSN) have emerged as an enabling technology for a variety of distributed applications. WSN middleware eases the development of these applications by providing a uniform programming environment. In this paper we present a rule based approach called REED (Rule Execution and Event Distribution) and describe how it supports flexible programming of WSNs at runtime. Indeed REED is required by the nature of its project setting to allow runtime programming. We demonstrate that by combining this runtime programmability with rules in an event, condition, action format we can support a range of paradigms, including Publish-subscribe and data aggregation algorithms. Current WSN middleware solutions have limited on-line programmability support so the applications cannot re-configure their WSNs while operational. Yet the runtime nature of the prototype requires both the distribution of rules and the events that trigger them so we also describe the rule management approach used to support the rule distribution; in particular a novel rule merging and filtering algorithm is described. The paper reports on the results gained from a REED prototype system constructed in our laboratory using Gumstix.

Keywords: wireless sensor networks, rules, programmability, middleware, pub-sub, aggregation

1 Introduction

Progress in sensor technology, wireless communications, and micro-processors, has provided a strong research interest in Wireless Sensor Networks (WSNs). Typically such networks consist of distributed sensor nodes interconnected via wireless links. WSNs feature a number of embedded sensor devices, each of which has constrained processing power, memory, and energy. The error-prone wireless links, over which devices communicate, often lead to message loss. In addition, the number of sensors can be very large and they can be heterogeneous in nature. This is a challenging environment for WSN software applications development [5]-[7]. To support the collection, delivery and querying of data; WSN middleware is often introduced to shield the application (developer) from the complexities arising from a WSN. The PROSEN[1] (PROactive SENsing) research project employed a wind farm setting to develop a proactive wind farm condition monitoring system; the research contribution being the proactive nature of the approach. A major aspect in PROSEN, was the combination of high-quality filtered data from the sensor nodes and AI based data analysis, to provide proactive goal-driven configuration management. This proactive approach adds new challenges to the WSN middleware: it requires the WSN middleware solution to support programmability, especially when the system was operational. This paper focuses on the run-time programmability aspect of the WSN middleware, and describes the Rule Execution and Event Distribution (REED) middleware used in the PROSEN project. In particular the paper highlights the

programming flexibility provide by the rule-based approach. While the paper describes the broader project to provide a context, in particular where it influences the implementation of the REED prototype, the paper aims to focus on the REED middleware proper rather than this wider context. This middleware supports both the distribution of rules and the events that trigger them. REED employs a rule-based paradigm to allow sensor networks to be programmed at run time. This provides a flexible environment where applications and users can program the sensor nodes to allow their behaviour to adapt to the applications' goals and the changing environment [8]. Also later in section 3.5 and 4.1 we describe how the REED middleware is also lightweight and energy-conservative. In Section 2, the REED middleware architecture is described, followed by the definition of the formal language for REED. The REED middleware is evaluated in section 3. The rule management is also discussed in this section. A REED prototype for PROSEN-WSN has been implemented and two WSN services provided via REED rules are described in section 4 to highlight the breadth of programming permitted by REED. Related work is discussed in Section 5, followed by the conclusions.

2 REED middleware architecture

To clearly describe the REED middleware architecture, the system architecture for the PROSEN is introduced first in section 2.1. In addition, the core definitions of the REED language are provided in section 2.3.

2.1 PROSEN architecture

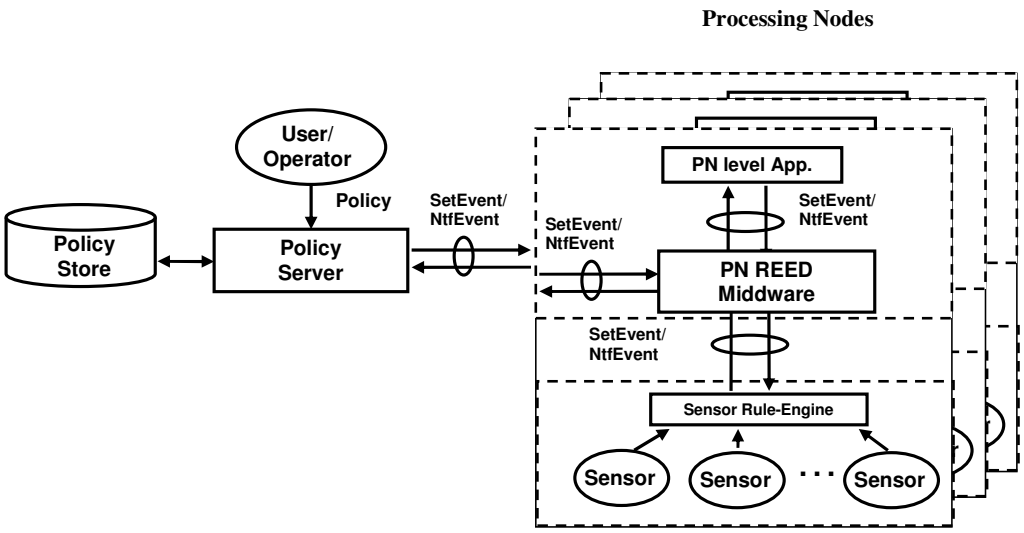

Figure 1: PROSEN system architecture

Figure 1 shows the system architecture for PROSEN, which consists of a *Policy Server* (PS)[29], a *Processing Node* (PN) for each wind-turbine, and sensors to measure parameters such as temperature, wind-speed, wind-direction, battery-level, and gearbox temperature. The PS wirelessly interconnects with the PNs via GSM; the PNs are wirelessly interconnected via VHF (174 MHz wireless links). Two communication primitives are both event based: *SetEvent* describes an event of setting a rule or a configuration parameter, etc.; *NtfEvent* describes an event notifying data or a timeout, etc. The PS interacts with users and operators to obtain the goals for the system. Such goals might describe a desirable power output or response to poor

weather conditions. The PS converts the goals to a set of policies. These policies are stored in the *Policy Store*, and then converted to low-level rules and distributed to the PNs via *SetEvent*. These rules describe the behaviour of individual PNs. The PROSEN REED manages and executes these low-level rules within each PN. It is also possible to transfer these rules between PNs.

In addition to distributing rules, the REED middleware also transfers *NtfEvents* between the system components. It is these events that trigger the rules.

Conceptually, a *rule* takes the form of <*event*, *condition*, *action*> where:

- an *event* is received from any other system component. This event often contains data values, but events such as a timeout, a sleep or wake-up can also occur.
- a *condition* is a Boolean expression that will be evaluated when the *event* occurs.
- an *action* is executed if the above *condition* is true when the *event* is received. The action may manipulate or store data. It may also generate another *event* to other components in the system, such as an *event* to trigger other *rules*.

2.2 REED architecture

To implement REED, a *rule-engine* has been designed and implemented. The functionality of the *rule-engine* includes:

- managing a *rule-base* to add, remove, and override REED *rules*
- verifying *rule* consistency, and
- executing the *rules* in response to received *events*.

Figure 2 shows the general architecture of the REED middleware. This echoes typical structures given in the literature [16]. The middleware must record certain aspects of the state of the node and the events that have occurred. These are recorded in the *Fact-Base*. Here we borrow the term *Fact* from a separate rule-based WSN approach [17]. In our architecture the *Event-Manager* is responsible for receiving events, passing them to the *Rule-Engine* where the engine executes any matching rules, and distributes any resulting events. The *Rule-Base* stores all the rules used by the engine.

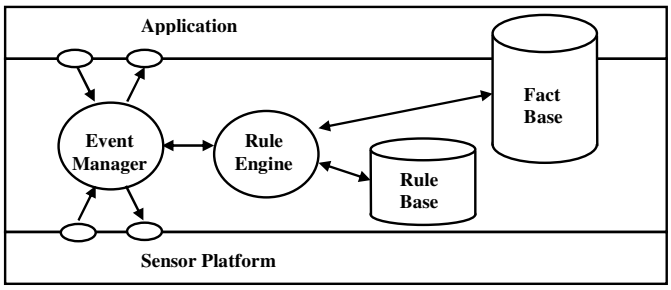

Figure 2: REED architecture

The REED middleware actually has two levels of rule-engine within a PN. Figure 1 illustrates the two-level REED architecture for PROSEN, where the *sensor rule-engine* is responsible for local sensor data collecting and processing[9], while the *PN REED middleware* is employed for wider event processing; such as data event correlation between processing nodes. The former is always on while the latter is only on when required, so it can offer more computational power when necessary while limiting its overall power demands. While the latter provides more powerful rule processing, it does require a limited Java Virtual Machine to operate; so the full two-

level approach does require the computational power of say a Gumstix. In summary, PROSEN rule based paradigm consists of three-level hierarchy: policy server, processing node REED, and sensor rule engine. In this paper we shall focus on the PN level REED middleware and this is discussed further in Section 3.

2.3 Language definition

To provide a clear description of the REED middleware, we use a formal notation using a variance of BNF described in Ref. [15] to define the REED language. To save space, only the core definitions of REED are given in Table 1. The full definitions can be viewed on [20]. As a result some minor items in Table 1 are not defined but are self-explanatory; for example *sc* for semi-colon, *cls_sqr_brckt* for close square bracket, *gte* for greater than or equal to, and *prprt* for property.

Table 1: Core language definition for REED

<reedRuleSet>	::=	<rule>+
<rule>	::=	<rule_id> <equals> <event_id> event_handler>
<event_handler>	::=	(<opn_sqr_brckt> <cond_set> <sc>
		<actions> <sc>
		<priority> <cls_sqr_brckt>)+
<cond_set>	::=	<cond> (<logic_op> <cond>)*
<cond>	::=	<true>
		| <exist_op> <fact_id> <dot> <prprt_name>
		| <fact_id> <dot> <prprt _name> <comp_op> <value>
		| <fact_id> <dot> <prprt_name>
		<comp_op>
		<fact_id> <dot> <prprt_name>
<logic_op>	::=	<and_op> | <or_op> | <not_op>
<comp_op>	::=	<equals> | <gt> | <lt> | <gte> | <lte> | <ne>
<fact>	::=	<state> | <event>
<fact_id>	::=	<state_id> | <event_id>
<event>	::=	<event_id>
		<opn_crl _brckt>
		<prprt> (<sc> <prprt>)*
		<cls_crl_brckt>
<state>	::=	<state_id>
		<opn_crl _brckt> <prprt> (<sc> <prprt>)* <cls_crl_brckt>
<prprt>	::=	< prprt _name > <equals> <value>
<actions>	::=	<action> (<comma> <action>)*

3 Properties of REED

3.1 Programming at run-time

Rule-based middleware, such as to FACTS [16], enable individual WSN nodes to be programmed. The stored rules define a node's behaviour in response to a series of events. However the rules are not changeable once they have been deployed and stored. (Although rule parameters can be changed at run time.) In contrast, PROSEN requires that the PS be able to alter the low-level rules after its deployment. In other

words, it is required that the system can be programmed at run time. REED enables the rules stored within the PN to be updated at any point in time.

Another advantage of providing a dynamic rule-set is the ability to easily apply REED to other applications without the need to re-flash the static rule-set within a PN. However to support dynamic updates of the *rule-base,* rule management is required and Section 3.4 will discuss this in more detail.

To demonstrate the flexibility of this approach we describe two distinct programming regimes that we have implemented using REED. Firstly we show how our rules can implement the publish-subscribe (pub-sub) paradigm (and a reliable variant), and secondly we implement a WSN aggregation algorithm selected from the literature.

3.2 Support for pub-sub service

Programming a PN with REED rules allows a broad and flexible approach. For example, an application may require event publish-subscribe services from REED. Here we show that this can be constructed using REED rules. Typically the events originate from sources such as sensors, internal timers, and peer PNs. The event subscription is equivalent to the subscriber sending REED a *rule* in which the *event* and the *condition* describe the subscribed event, and the *action* is set to send the received event back to the subscriber[1]. Crucially, this is possible because the rules can be updated dynamically by peer nodes. Thus, the event notification is published as the result of executing the *action* part of such a *rule.* Consider an example where a subscriber, say a PS, wishes to receive wind speed values when the wind speed is greater than 60 kph; the PS subscribes to a PN REED middleware by sending a *rule* to the PN in the form:

```
Rule 1    =  wind_speed [2]
             [  wind_speed.Value >> 60;        Send(PS, WindSpeed)        ]
```

(For simplicity, the *Priority* field is not shown in the EventHandler.)

Assume that later on the REED receives a WindSpeed event from a sensor via a *sensor rule-engine* in the form of:

Event 1 = wind_speed {Value = 67; ID = 2; Time = 23:14:12; Date = 26-08-10}

This event will trigger the execution of the Rule 1, and as a consequence, this event will be notified to the PS. There can of course be more than one subscriber; and so a single event can result in a number of notifications. Also the threshold value can be set low to allow all events of a particular type to be published.

3.3 Extended functionality

Section 3.2 describes a basic sub-pub functionality. This is given to emphasise the flexibility that the approach gives a programmer. However the programmer may, for example, wish to extend the pub-sub functionality to ensure it is more reliable. One approach that could be employed is when either a *subscriber* plans to reliably send a rule or a *publisher* to send a notification, the node can simply execute the following Rule 1-0, as shown in Table 2 to its *rule base*, and send itself a ReliableMsgEvent to trigger this rule.

---

[1] To do so is the choice of the source of the rule; our approach does not require that any generated event returns to the rule source.

[2] WindSpeed represents a wind speed *Event* while wind_speed represents the *EventID* of the WindSpeed. These formats of representing *Event* and *EventID* are used throughout this paper.

Table 2: rule for reliable transmission

```
Rule 1-0  =  reliable_msg_event
             [  TRUE;
                ReliableMsgEvent.Id = ReliableMsgEvent.CRC32( )
                Add_rule (Rule 1-1),                Add_rule (Rule 1-2),
                Send (Receiver, ReliableMsgEvent),
                Start (send_timer)                                          ]
```

where

```
Rule 1-1  =  ack_event
             [  AckEvent.Id == ReliableMsgEvent.Id;
                Clear (send_timer)                  Delete (Rule 1-1, Rule 1-2)  ]
```

```
Rule 1-2  =  send_timeout
             [  TRUE;
                Send (Receiver, ReliableMsgEvent),
                Start (send_timer)                                          ]
```

On the receiver side, the following rule will be executed.

```
Rule 1-3  =  reliable_msg_event
             [  ReliableMsgEvent.CRC32( ) == ReliableMsgEvent.Id;
                AckEvent.Id = ReliableMsgEvent.Id,
                Send (Sender, AckEvent)                                     ]
```

where ReliableMsgEvent.CRC32( ) is the CRC32 value of the received message. By comparing the calculated CRC32 value with the received ReliableMsgEvent.Id which is the CRC32 value of the message sent, the transmission errors can be detected.

For the message sender, when the local *rule engine* executes Rule 1-0 triggered by the local ReliableMsgEvent, it will carry out the four actions shown in Table 2. Firstly it will add two rules to its own *rule base*: Rule 1-1 is to respond to the acknowledgement from the receiver showing the message has been successfully received; Rule 1-2 is to respond to the timeout event, which indicates either the message or the acknowledgement doesn't reach the other end successfully, by sending the ReliableMsgEvent again. In action 3, the sender sends the message to its receiver with its identifier (CRC32 value); and in the meanwhile starts the timer during which the sender waits for the acknowledgement from its receiver. Note that in the actions the node will substitute the concrete value obtained from the trigger to Rule 1-0, Rule 1-1 and Rule 1-2.

By using Rule 1-0 and Rule 1-3, the messages can be reliably transmitted, and no duplicated messages will be received.

In the implementation of the REED, some general rules, such as those for the reliable transmission (Rule 1-0 and Rule 1-3), can be preset as default rules and thus are loaded to the *rule set* at boot time. At run time, new rules, when needed, are added one by one via those preset rules for the reliable transmission.

3.4 Rule base management

In REED middleware, the *rule base* is used to store the rules for the rule-engine, and as a consequence rule base *management* is required. In addition to handling the adding, removing, and updating of the rules, the *rule engine* must be able to:

- Maintain the consistency of the *rule base*.
- Merge the rules from various sources.
- Filter the rules to its *sensors rule engine*.

The rules are updated dynamically at runtime; indeed the pub-sub example described above would not be possible without this capability. Therefore it is important that the rule management too is dynamic and can manage rule changes at runtime. To the authors' knowledge, this work is the first to employ a rule merging and filtering mechanism supporting rule changes at runtime.

3.4.1 Rule base consistency. Consistency is required because the *rule engine* can receive rules from various sources. In PROSEN, the PN level REED middleware may receive the rules from the policy server, its own application entities, or from its peers. To maintain the consistency of the *rule base*, the *rule engine* should detect and resolve any conflicting rules. These conflicts arise as an event may trigger more than two rules and generate conflicting actions, e.g. one rule setting a sensor on and another rule setting the same sensor off. Normally, the way to resolve this is to set different priorities so only the rule with the highest priority will be triggered. In PROSEN, the core responsibility of the rules quality is taken by the top-level of the hierarchy: the *Policy Server*. This is because firstly, the *Policy Server* is the main source of the rules. It provides the interfaces to the operators for domain knowledge based policies, maintains the consistency of the policies using meta-policies [21], and then transforms those policies to the rules for the REED. Secondly, the *Policy Server* runs on a resource rich device powered by the mains electricity. To make sure no conflicts among the rules running on the REED, the REED accepts control or configuration related rules from the *Policy Server* only. The rules from its own application entities or from its peers are all data acquisition and data processing related rules such as publish-subscribe service and data aggregation service that will not cause conflicts in control or configuration of the system. However it has been noticed that a stronger mechanism is required on the REED to ensure the quality of the rules from other sources, such as rules authentication etc. This is a subject for further work and indeed while aspects of rule conflict can be addressed by middleware alone, the broader issue of rule quality requires domain guards.

3.4.2 Rules merging. Rules for the same *event* but from various sources may be merged. For example, should the REED on a PN receive a rule from another PN (denoted as PNx) in the form of:

```
Rule 2   =  wind_speed
            [  wind_speed.Value >> 50;        Send(PNx, WindSpeed)        ]
```

and also receives a rule from the PS saying:

```
Rule 3   =  wind_speed
            [  wind_speed.Value >> 70;        Send(PS, WindSpeed)         ]
```

instead of storing two separate rules in the *rule engine*, they can be merged into a single rule as:

Rule 4	=	wind_speed		
		[wind_speed.Value >> 70;	Send(PS, WindSpeed)	]
		→ [wind_speed.Value >> 50;	Send(PNx, WindSpeed)	]

where the symbol "→" means a coverage link which will be explained in Table 3. Rule merging not only reduces the number of rules in the *rule base*, but also supports the rule filtering described in section 3.4.3.

3.4.3 Rules filtering. In PROSEN, a hierarchical rule-engine structure is adopted such that the PN-level *rule engine* accepts rules, and forwards any sensor-level rules to the *sensor rule engine*. As the *sensor rule engine* runs on a more resource-limited processor, its rule set should be concise and free of any redundant rules. Hence REED filters out any redundant rules before forwarding them to the *sensor rule engine*. For example, should the REED on a PN receive Rule 2, as shown in section 3.4.2, from a peer node (say PNx), and then receives Rule 3, as shown in section 3.4.2, from the PS, instead of sending two corresponding rules to the *sensor rule-engine*, REED sends only one rule:

Rule 5	=	wind_speed		
		[wind_speed.Value >> 50;	Send (REED, WindSpeed)	]

to its *sensor rule engine* with another one being filtered out. When a *WindSpeed* event is sent from the *sensor rule engine* to the REED, the rules for this event are executed as follows: first check whether the wind speed (*wind_speed. Value*) is greater than 70 kph, and then check whether the wind speed is greater than 50 mph, to determine where to send the notification: to both the PS and PNx if the reading is over 70, or to PNx only if the reading is between 50 and 70.

For space considerations we give a brief description of the rule merging and filtering algorithm in Table 3, and suppose the *ConditionSet* contains one *Condition*.

Table 3: Algorithm for rule merging and filtering

Definition 1: Given a *Condition1* and a *Condition2*,

$\forall$ *event,* if in meeting *Condition1* means it also meets *Condition2*, then we say *Condition1* is **covered** by *Condition2*, denoted as

Condition1 $\subseteq$ *Condition2*.

Definition 2: Given

rule1 = *eventID* [*Condition1*, *Action1*] and

rule2 = *eventID* [*Condition2*, *Action2*] are triggered by the same event,

if *Condition1* $\subseteq$ *Conditiont2*, then we say *rule1* is covered by *rule2*, denoted by *rule1* $\subseteq$ *rule2*, which means that if the *rule1* is triggered by the *Event*, the *rule2* must be triggered too.

Algorithm for rule merging and filtering:

When the PN REED *rule engine* receives a rule in the form:

R1 = *event_ID* [*Condition1*, *Action1*]

IF there is no other rule in the current *rule base* that has coverage relationship with *R1*, **THEN**:

Save this rule to the *rule base*;

Initialize the *counter* for node [*Condition1*, *Action1*] as 1;

Construct a rule:

r1 = *event_ID* [*Condition1*, *send(REED*, *event*)];

Forward *r1* to the *sensor rule engine*;

Later on, when the *rule engine* receives another rule in the form:

R2 =event_ID [*Conditions2*, *Action2*]

IF *R2* is covered by *R1*, **THEN**:

Change the *R1* originally saved in the *rule base* to

R1 = event_ID [*Condition1*, *Action1*] → [*Condition2*, *Action2*];

/* where the symbol "→" means a coverage link with [*Condition1*, *ActionSet1*] being the head and [*Condition2*, *ActionSet2*] being the tail of the link **(rule filtering)**. */

Initialize the *counter* for node [*Condition2*, *Action2*] as 1;

IF *R2* covers *R1,* **THEN**:

Change the *R1* originally saved in the *rule-base* to

R1 = event_ID, [*Condition2*, *Action2*] → [*Condition1*, *Action1*];

Initialize the *counter* for node [*Condition2*, *Action2*] as 1;

Construct a new rule:

r2 = event_ID [*Condition2*, *send*(*REED*, *event*)];

Forward *r2* to the *sensor rule engine* to replace the original one **(rule merging)**;

IF [*Conditions2*, *Action2*] already exists, **THEN**::

Increases the counter for [*Conditions2*, *Action2*] node by 1 **(rule merging)**;

When the *rule engine* later receives a rule in the form:

R3 = event_ID [*Condition3*, *Action3*]

IF the current coverage link for *Event* is

[*Condition1*, *Action1*] → [*Condition2*, *Action2*], **THEN**:

Insert [*Condition3*, *Action3*] into this coverage link (rule merging);

Initialize the *counter* for node [*Condition3*, *Action3*] as 1;

IF [*Condition3*, *Action3*] becomes the new head of this coverage link, **THEN**:

Updates the rule to the *sensor rule-engine*;

When a *Remove*(*R1*) is received:

Decrements the counter for [*Condition1*, *Action1*] by 1;

IF the result reaches 0, **THEN**:

IF [*Condition1*, *ActionSet1*] is NOT at the head of the covering link, **THEN**:

Remove [*Condition1*, *Action1*] from the link;

ELSE IF [*Condition1*, *ActionSet1*] is at the head of the covering link, **THEN**:

Remove [*Condition1*, *Action1*] from the link;

IF [*Condition1*, *Action1*] has NO child node, **THEN**:

Send a command to the *sensor rule-engine* to delete the rule:

r3 = event_ID [*Condition1*, *send*(*REED*, *event*)];

ELSE IF [*Condition1*, *Action1*] has a child node && [*ConditionSet2*, *Action2*] is the child node, **THEN**:

Make [*ConditionSet2*, *Action2*] the head of the coverage link;

Construct a new rule:

r4= event_ID [*Condition2*, *send*(*REED*, *event*)];

Forward *r4* to the *sensor rule engine* to replace the original one.

3.5 Performance considerations

REED is lightweight in terms of the energy and memory consumption. This is because first of all, it is event triggered instead of continuously polling and this saves wireless bandwidth and energy. Secondly, unlike JESS[26] where all the facts are stored in its working memory before the execution of their rules, REED filters the received data events using its rules and only those needing further processing will be saved to the *fact-base*. This makes the overhead for memory consumption much lower. Thirdly, the pub-sub service (or indeed any rules only generating a data anomaly) ensures that data events are only handled by those components that require them. This is in contrast to LIME[10] and TinyLIME[11] middleware which are Tuple Space-based where the data sharing and synchronization across the network is both bandwidth and CPU consuming.

For real applications, some rules can be set as default rules and are embedded within the REED *rule base* locally during the initiation. The rules are updated at run time only when necessary. This will further save the power for rule distribution and rule management.

As REED is event based, it can go into a *sleep* state in order to save the battery energy when there is no event for processing. It returns to the *work* state either by a scheduled timeout or a triggered event. In PROSEN, the signals for *sleep* and *wake-up* are triggered by the *sensor rule engine* which is always in a *working* state. When the *sleep* event is received, the REED writes the unsaved *rules* and necessary *facts* to the flash memory before it exits. When the REED is initiated as the result of a *wake-up* event, it will, before processing any event, restore those *rules* and *facts* back from the flash memory.

Although the two rule engines could be replaced by one that simply had an effective sleep mode, the project adopted this dual processor approach as it allows continuous monitoring of sensors. Certain quantities such as wind speeds (for gusting) and vibration require frequent monitoring and so the dual approach allows continuous monitoring without the higher power consumption associated with processors capable of supporting Java. This is pertinent as the rule execution within the MSP430 allows the Gumstix to remain off over much longer periods of time than the MSP430.

When REED is executing rules, laboratory measurements show a power consumption of 1.4W. Clearly the ratio between the wakeup and sleep states is important and will be determined by particular applications (rule-sets). The ratio of activity states is a function of event traffic and the number of rules per event. The goal of such in-node processing is to drastically reduce such traffic. When the REED rule engine is off the remaining sensor engine uses the order of 750 μW.

Section 4.1 discusses REED resource requirements in more detail.

4 Prototype implementation

4.1 Prototype implementation architecture

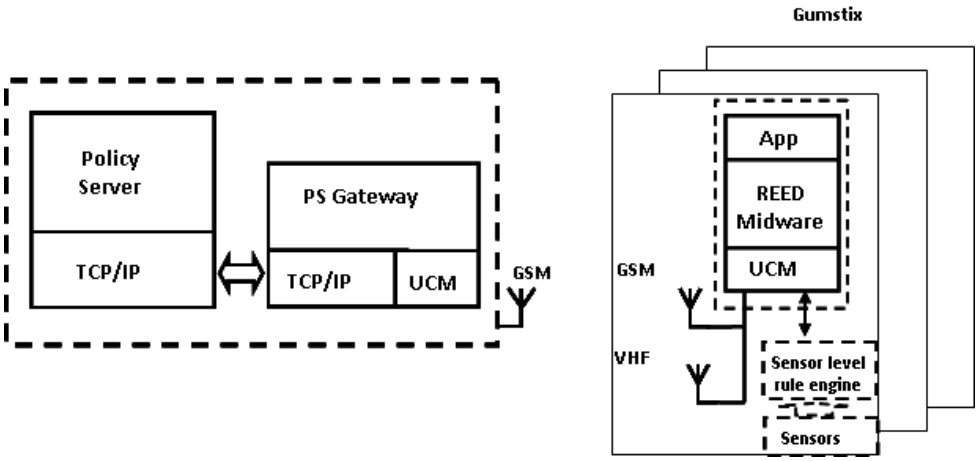

Figure 3: Prototype implementation architecture

Figure 3 shows the prototype implementation architecture. The software structure for REED middleware is illustrated in Figure 4. REED sends and receives external messages via the interfaces provided by the UCM. The UCM (Unified Communications Manager), developed by other partners in the PROSEN project, provides a platform to communicate to the PS via GSM, to peer PNs via VHF, or to its *sensor rule engine* via a UART. The *Event Constructor* constructs events with the received messages. It classifies them either as *SetEvents* containing rules, or as *NtfEvents* (e.g. data events), and then puts them onto their corresponding queues. These two queues may have different priorities. In our implementation, the queue for *NtfEvents* has higher priority in order to respond to the data events as quickly as possible. When any event is to be distributed, the *Msg Constructor* will transform it to the corresponding message format before delivering it to the UCM.

REED is running on a Gumstix™[24] GS400K-XM, which is a miniature full function Linux motherboard based on low power Intel XScale® technology. GS400K-XM has 16MB flash memory which can accommodate JamVM[25], which is a compact JVM (Java Virtual Machine), and so REED is developed using Java. Hence REED can easily be ported to other Java based platforms. Indeed we have used Gumstix as a realistic testing environment, rather than advocating Gumstix as an ideal setting for REED.

At the time of writing, the core functionality of REED has been implemented: that is, functions for adding and updating rules; executing rules triggered by events; merging and filtering rules based on the same event; and *rule base* and *fact base* store/recovery in response to *sleep* and *wake-up* events. Although a *sensor rule engine* has been built, for experimentation purposes we employed a simple event generator to emulate the *sensor rule engine*.

Before running REED, 11760 KB memory (RAM) on the Gumstix is used, including the memory for running UCM. When REED is running, the memory used is 16624 KB. So the memory footprint for REED is 4864 KB. As the GS400K-XM used in our system has 128MB RAM, the REED footprint takes around 3.8% of the total available RAM. Other Linux measurements show that when waiting for events REED uses less than 1% of CPU processor time.

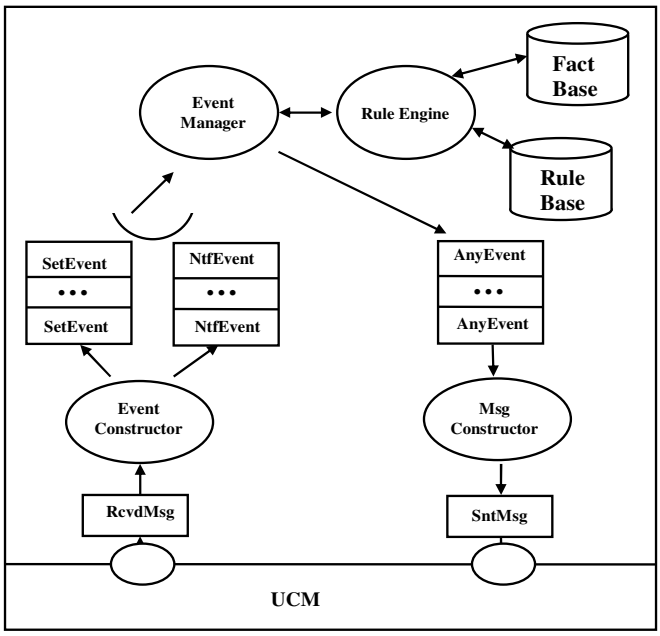

Figure 4: REED Software Structure

4.2 Sensor data pub-sub service

The sensor data pub-sub service has been implemented in REED. The system (see Figure 5) consists of one PS, one Gateway and one PN. As this service does not include any cooperation between peer nodes, one PN is sufficient for experimental purposes. The PS and the Gateway are connected via the Internet, and the Gateway and the PN are connected via a 174MHz wireless link. Each 174MHz wireless board shown in Figure 5 consists of a Radiometrix BiM1-173.250-10 10mW NBFM Transceiver and an in-house extension board with RS-232 Universal Asynchronous Receiver/Transmitter (UART) interface to the Gumstix. For testing purposes the sensor reading is simulated via a random number generator with uniform distribution between 0 and 100. Taking the example given in section 3.2, where the *event manager* on a PN receives Rule 1 from the PS and stores this rule in its rule base, converting it to a rule understandable to its *sensor rule engine*, before forwarding the converted rule to the *sensor rule engine* via the UCM. The converted Rule 1, expressed in REED notation, is:

```
Rule 6    =  wind_speed
             [  wind_speed.Value >> 60;        Send(REED, WindSpeed)        ]
```

Compared to Rule 1, Rule 6 asks the *sensor rule engine* to send the *WindSpeed* event to the REED instead to the PS. This is because the REED may do some further processing on the *WindSpeed*, such as composite condition checking, or merged condition checking.

In this scenario, the sensor data filtering is actually carried out by the *sensor rule engine* instead of the REED. This is to save the energy of both the raw data transfer from the *sensor rule engine* to the REED, and the power consumption of the REED rule engine.

A snapshot of the PS is shown in Figure 6 where Event 1 is received in response to Rule 1.

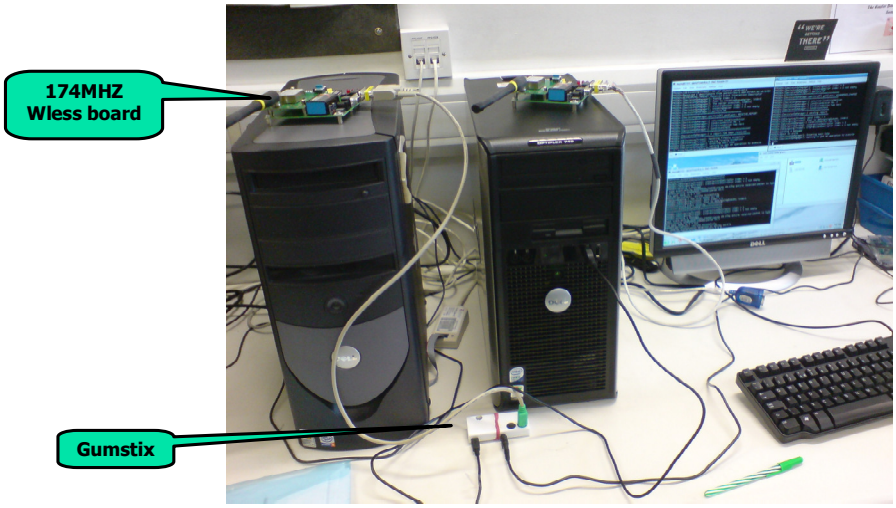

Figure 5: PROSEN prototype system

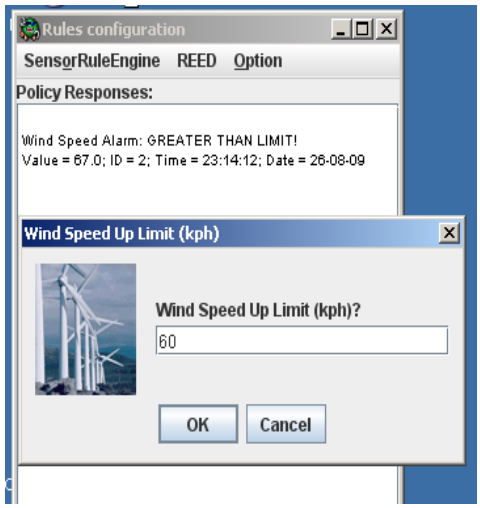

Figure 6: a snapshot on the PS

4.3 Sensor data aggregation over wireless links with message loss

Increasing computational power within the wireless network components offers the opportunity to aggregate data within the network; so rather than each sensor simply report values to a central point, the network can calculate an aggregate value. Indeed data aggregation reduces communications cost and increases the reliability of data transfer. This is most pronounced for WSN applications which have large amount of data to send across the network[5]. In PROSEN, for example, the administrator may want to know the average wind speed experienced by the turbines across a wind farm; if the average wind speed is over or under a certain value, the administrator may decide to shut down the wind farm. A centralised approach to obtaining the average wind speed is to ask the sensor nodes to send their wind speed readings to the PS, and then average all those readings on the PS. However, in this setting this solution is energy consuming[3]. So the solution of averaging the sensor data in a distributed manner within the network was investigated. A simple approach might require each

[3] In PROSEN, when a PN transmit data to the PS via GSM, the average current drawn is 110 mA with the *instantaneous* current drawn being as high as 2.5 amps; when a PN transmit data locally via low power 174 MHz radio, the average current drawn is 10 mA.

PN to broadcast its reading to some PN within radio range elected as the *data collector* to collect and average the data. However with unreliable wireless communication it is important that any loss of data is managed effectively.

To illustrate the ability of REED to implement such algorithms we chose an algorithm from the literature. Chen et. al. [27] propose an algorithm to calculate an accurate aggregation of data within a WSN environment. This algorithm assumes that not all nodes are in range of each other. A node can declare itself a leader if it has not been forced to be a slave by another leader node. As a partially connected network will form a set of overlapping clusters (each with its own leader), the algorithm's concurrency allows the sensed data and the resulting mean to ripple through the network. To demonstrate how some simple rules in each node can implement such a strategy we implement a simplified test bed that assumes all the PNs are one-hop away among one another. Hence, while the testbed supports message loss it is in practice implying a single cluster. This simplified implementation has been transformed into the REED rules that were executed by the *rule engine* on the PNs.

The algorithm works in the following manner. Any node can declare itself as a leader; and does so with a random probability. Once an individual node declares itself a leader it sends a signal to all of the other nodes (that happen to be in range) informing them that it is a leader. Each node that receives this signal responds by sending back their current value. Such nodes are denoted as *active nodes*. The algorithm accepts that not all nodes will respond and the leader calculates the mean using the data returned. The leader then sends this new mean back to the *active nodes*, and resigns. This cycle continues when a node declares itself a leader. The first time a node becomes an active node it returns a measured value, but on all subsequent occasions it returns the last mean value it received from a leader. It is assumed in our experiments that all nodes are in range of a leader, and where messages are lost they do not cause a PN to become isolated. In other words we assume one cluster.
Based on the algorithm, the rules were constructed as shown in Table 4. The rules employ the state of the node on which they execute. So, each node creates and maintains a state called My_State.

Table 5 lists the properties of the state My_State and includes a brief description.

This work is described in more detail in Ref. [28]. The aim is to show that the REED rules can capture a data aggregation algorithm and operate successfully by way of an exemplar. It is not the intention to verify the efficacy of the aggregation algorithm per se. Hence the experimental design is deliberately simple and straightforward.

The system to test the execution of these rules consists of nine Java applications running on PCs simulating nine PNs, and one running on the Gumstix. The gumstix is connected to the PS via USBnet. The sensed data is simulated via a random number generator with a uniform distribution between 0 and 100. In order to simulate 10 percent loss rate, another uniform random number ranged from zero to one is generated, and the value greater than 0.9 indicates a message loss. The tests were carried out in four message-loss settings respectively. In the first setting, there was no message loss, and an accurate mean was obtained after only one iteration; in the second setting, the message loss was 10 percent throughout the test, and an answer accurate to one decimal point was obtained after four iterations, as shown in Figure 7; in the third setting, the message loss was 20 percent throughout the test, and an answer with the same precision was obtained after 7 iterations, as shown in Table 6; in the fourth setting, previous three settings occurred sporadically, and an accurate mean was calculated if the leader received full connectivity before 7 iterations.

Table 4: rules for the sensor data aggregation

Rule 1 is triggered by the WindSpeed event to start the algorithm.

```
R-1  =  wind_speed
        [TRUE;                                  Send (self, AggregationStart)      ]
```

Rule 2 is for leader election.

```
R-2  =  aggregation_start
        [(!∃(My_State.Identity));               Set (My_State.Identity, "leader"),
         Send (Neighbour, LeaderSignal)                                            ]
```

Rule3 is in response to LeaderSignal event: the non-leader either sets itself as a slave and sends its sensed data to the current leader, or sends its stored mean to the current leader.

```
R-3  =  leader_signal
        [(!∃(My_State.Identity));               Set (My_State.Identity, "slave"),
         Send (Leader, SensedData)                                                 ]
        [(My_State.Identity == "slave")   &&    ∃ (My_State.Mean);
         Send (Leader, My_State.Mean)                                              ]
        [(My_State.Identity == "slave")   &&    (!∃ (My_State.Mean));
         Send (Leader, SensedData)                                                 ]
```

Rule 4 is for leaders: it calculates the average and sends it to itself for further processing.

```
R-4  =  data_set
        [My_State.Identity == "leader";         Average (DataSet, Mean),
         Send (self, Mean)                                                         ]4
```

Rule 5 is in response to the Mean event: for the leader, it sends the final result to the PS if the algorithm completes; otherwise, it sends the Mean event back to its members, designates a new, and sets itself as a slave; for the slave, it simply stores the mean value.

```
R-5  =  mean
        [(My_State.Identity == "leader")                    &&
         (My_State.Active_Node_Set == Node_Set);
         Send (PS, My_State.Mean)                                                  ]
        [(My_State.Identity == "leader")                    &&
         (My_State.Iteration_No == Iteration_No_Threshold);
         Send (PS, My_State.Mean)                                                  ]
        [My_State.Identity == "leader")                     &&
         !(My_State.Active_Node_Set == Node_Set)            &&
         (My_State.Iteration_No << Iteration_No_Threshold);
         Set (My_State.Mean, Mean),
         Send (Active_Node_Set, Mean),          Send (new_leader, BeLeader),
         Set (My_State.Identity, "slave")                                          ]
        [My_State.Identity == "slave";          Set (My_State.Mean, Mean)         ]
```

Rule 6 is for new leader to start another iteration.

```
R-6  =  be_leader
        [My_State.Address == new_leader;        Set (My_State.Identity, leader),
         Send (Neighbour, LeaderSignal)                                            ]
```

[4] DataSet is received from the neighbours. It can be either previous mean or the sensed data.

Table 5: properties of the object My_State

Property name	Description
Identity	Three values: “Null” (initial value), “leader” or “slave”
Mean	Current calculated mean value
Active_Node_Set	Set of nodes from which the data is received, the property maintained by the leader.
Iteration_No	Number of iterations carried out so far to calculate the current mean value
Address	The address of the node

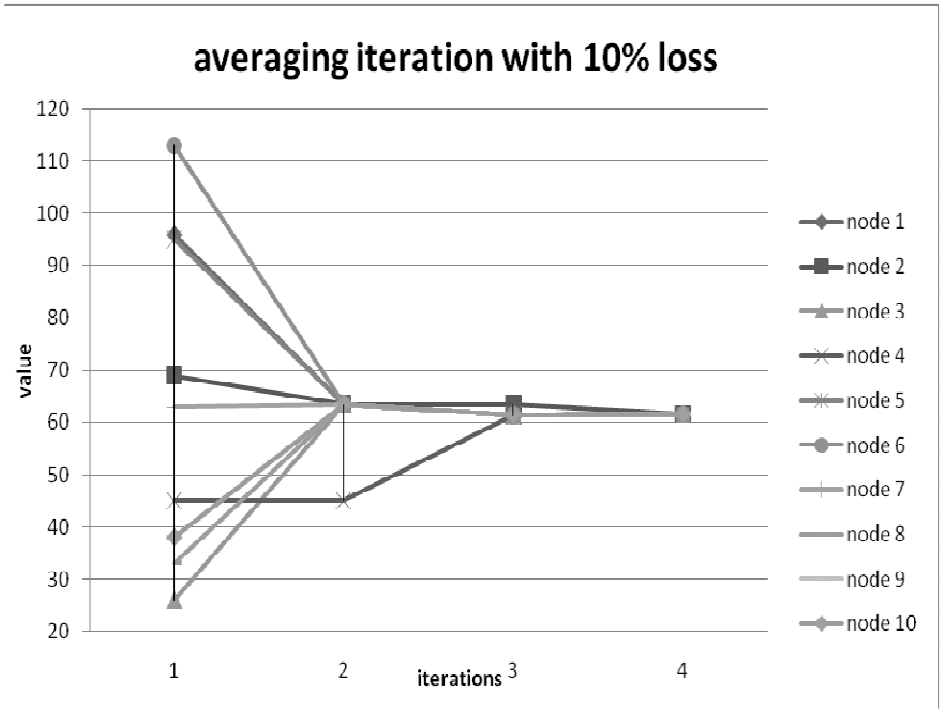

Figure 7: Averaging iteration with 10% loss

Table 6: Averaging iteration with 20% loss

NODE ID	ORIGINAL READINGS	ITERATION 1	ITERATION 2	FINAL AFTER 7 ITERATIONS
Node 1	18.0	17.5	17.56	17.29
Node 2	17.0	17.5	17.56	17.29
Node 3	16.0	17.5	17.56	17.29
Node 4	18.0	17.5	17.56	17.29
Node 5	15.0	15(MISSED)	15(MISSED)	17.29
Node 6	18.0	18(MISSED)	17.56	17.29
Node 7	18.0	17.5	17.56	17.29
Node 8	17.0	17.5	17.56	17.29
Node 9	17.0	17.5	17.5(MISSED)	17.29
Node 10	19.0	17.5	17.56	17.29

It should be noted that all these rules were run by the REED. The only rule for the *sensor rule engine* is:

R-7	=	wind_speed		
		[TRUE;	send (REED, WindSpeed)	]

4.4 Rules merging and filtering for data pub-sub and data aggregation

This section demonstrates the application of the rule merging and filtering mechanisms described in section 3.4. The description considers the rules employed for both the data pub-sub service, and the data aggregation service.

Initially in the experiments, the WSN system is deployed to carry out the data aggregation task. On checking the size of the Hashtable created for storing the rules on the Gumstix, **six** is shown. This is because six rules, R-1 to R-6, as described in section 4.3 are executed by each PN level *run engine*; likewise, checking the size of the rule table maintained by the *sensor rule engine* gives the result of **one** as only one rule, R-7, as described in section 4.3, is executed by each *sensor rule engine*.

A second stage of the experimentation captures the situation where a user (e.g. the wind farm operator) wants to employ the pub-sub mechanism in addition to the data aggregation rules. The experiment assumes a user plans to subscribe to wind_speed events by sending Rule 1 to the PNs, as described in section 3.2.

4.4.1 Rules merging

Without the rules merging mechanism in place, checking the current size of the rules Hashtable on the Gumstix gives a result of **seven**. This is because the PN level *rule engine,* upon receiving Rule 1, simply adds Rule 1 to its *rule base*. This results in a total of seven rules, of which six rules are for data aggregation and one for the sub-pub service being stored in the *Rule-Base*.

When the rules merging mechanism is applied, the size of the rules Hashtable on the Gumstix remains at **six**, and in the meanwhile both data pub-sub service and the data aggregation service are provided. This is because the PN *rule engine*, upon receiving Rule 1, carries out the rule merging explained as follows:

As "wind_speed.Value >> 60" (*condition* part in Rule 1) $\subseteq$ "TRUE" (*condition* part in R-1), by using the rule merging algorithm described in Table 3, Rule 1 and R-7 are merged as:

R-8	=	wind_speed		
		[wind_speed.Value >> 60;	Send(PS, WindSpeed)	]
		$\rightarrow$ [TRUE;	Send (self, AggregationStart)	]

As a result, the PN level *rule engine* replaces R-1, which has already been stored in its *rule base*, with R-8. Due to this rule merging, the number of the *rules* in the *rule base* is kept at **six**, and both data pub-sub service and data aggregation service are provided after the rules are merged.

4.4.2 Rules filtering

The rule filtering experimentation took a similar pattern. Without the rule filtering mechanism in place; to provide both data pub-sub service and data aggregation service, two rules which are Rule 6 (as described in section 4.2) and R-7 (as described in section 4.3) were running on the *sensor rule engine.*

In contrast when the rules filtering mechanism is applied, the size of the rule table on the *sensor rule engine* remains at **one**. This is because Rule 6 is filtered out by the PN level *rule engine* as described in section 3.4.3, and thus the rule is not sent to the *sensor rule engine*. This results in only **one** rule, R-7, instead of two, being stored and executed by the *sensor rule engine*. Again both data pub-sub service and data aggregation service are provided after the rule filtering.

In conclusion, by using rules merging and rules filtering algorithms, both the memory space for storing the *rules*, and the computing load (and thus the power consumption) of the *sensor rule engines* for executing rules are reduced without compromising the overall functionality. In other words, the overhead of filtering and merging once for each additional rule, is less than the repeated processing of events by the rule engine.

5 Related work

[3], [6] and [7] provided surveys across a broad array of WSNs and middleware. Well established mechanisms in the literature are LIME[10] (Linda in a Mobile Environment) and TinyLIME[11]. LIME and TinyLIME provide a Tuple Space based middleware. However, LIME is heavy-weight in that mobility management and data synchronisation are bandwidth and CPU consuming. TinyLIME is the extension of LIME, but it cannot be employed directly on currently available sensor processing nodes such as Tmote. A special interface has to be provided to bridge TinyLIME running on the base station and applications running on sensor nodes.

[13] proposed an event-based distributed middleware architecture, Hermes, that follows a type- and attribute-based pub-sub model. In [12], SIENA, an event notification service consisting of notification selection service and notification delivery service has been presented. SIENA exhibit both expressiveness and scalability. However, both Hermes and SIENA are for IP based Internet.

[14] proposes an ECA (Event, Condition and Action) rules based middleware model for WSN. In [16], a rule-based middleware architecture for WSN, called FACTS, was proposed, and [17] described its programming primitives and implementation using the Haskell programming language. While drawing inspiration from FACTS, our proposal is distinctive in that the rule set in FACTS is static while the *rule-base* in REED is dynamic as the rules for REED middleware can be updated at run time. FACTS does support changes of rule parameters at run-time but not the rules proper[18]. This limits run-time flexibility.

Snlog[2], [3] is also a rule based approach however it does not support run-time rule updates.

Mate[19] is a small virtual machine that enables sensor network programming at run time. However, it is important to stress that, compared to Mate, this paper is addressing run-time reprogrammability in terms of rules rather than low level programming constructs. More generally, run-time programmability has been addressed in terms of configuration data which although efficient is rather inflexible. It has also been addressed in terms of complete binary images which while flexible is very demanding of resources[19]. However virtual machines have been employed to

give efficient reprogrammability[19]. This approach is for a limited number of events. In addition, the programs for running on this virtual machine are of an assembler language style. In contrast our programming paradigm is that of rules. We believe this to be a more powerful notation. While reprogrammability at run-time for rule-based systems has been achieved through configuration data[18], we are not aware of any systems that address rule changes at run-time.

[22] and [23], written by S. L. Keoh, N. Dulay, E. Lupu, et. al, proposed a policy based middleware architecture for managing body sensor networks, in which the policies take on the same form (<*event*, *condition*, *action*>) used in REED. Indeed [23] concludes that the policy based middleware provides flexibility to reprogram the sensor with new adaptation strategies without requiring installation of new code. However, they do not demonstrate such reprogramming scenarios, nor do they provide dynamic policy management such as rule consistency, and rule merging and filtering during the reprogramming.

JESS is a rule-engine written entirely in Sun's Java language[26]. It is for general purpose and not dedicated for a WSN environment. As a consequence, the memory usage is not optimized [16] for running on sensor nodes. In addition, in JESS, all the facts are stored in its working memory before executing the rules while in REED, any received data event will be filtered by rules first and only those needing further processing will be saved to the *fact-base*. As a result, the overhead for memory consumption is expected to be lower than using JESS.

6 Conclusion

In this paper, the REED middleware is described. It supports both the distribution of rules and the events that trigger them. REED employs a rule-based paradigm to allow sensor networks to be programmed at run time, so that applications and users can programme the sensor nodes to allow their behaviour to be changed at run time. Such a rule-based approach allows, among others, data pub-sub service and data aggregation service to be constructed. To support this programmability, the rule management is also discussed, especially, a rule merging and filtering algorithm is proposed. The prototype implementation demonstrates the REED middleware functionality. By the time the PROSEN project was finished, REED middleware has been integrated with other components in the system, such as the PS, the UCM, and the sensor rule engine. The sensor data pub-sub service has been tested on the real system. The sensor data aggregation service has been tested on the prototype implementation.

In the future, REED is intended to be extended to other applications such as health care or other condition monitoring systems. Their domain knowledge will be collected and then expressed via rules for REED to provide data processing, filtering and collating services.

7 Acknowledgements

The authors would like to thank EPSRC (Engineering and Physical Sciences Research Council) for the funding the PROSEN project. We are also indebted to our PROSEN research colleagues and Graeme Ballie for their support.

References

[1] PROSEN: http://www.prosen.org.uk/

[2] D.Chu, L. Popa, A. Tavakoli, J.Hellerstein, P. Levis, S. Shenker, and I. Stoica. "The design and implementation of a declarative sensor network system", In Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys), 2007.

[3] L. Mottola, and G. P. Picco. "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art", ACM Computing Surveys (CSUR) Volume 43 Issue 3, April 2011

[4] K. Römer, O. Kasten, and F. Mattern, "Middleware Challenges for Wireless Sensor Networks", ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 6, Issue 4, 2002

[5] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey", Computer Networks,Volume 52 , Issue 12, pp: 2292-2330, 2008

[6] M. Kuorilehto, M. Hannikainen, and T. D. Hamalainen, "A Survey of Application Distribution in Wireless Sensor Networks", Journal on Wireless Communications and Networking 2005:5, 774–788

[7] E. Yoneki, and J. Bacon, "A Survey of Wireless Sensor Network Technologies: research trends and middleware's role", Technical Report www.cl.cam.ac.uk/techreports/ UCAM-CL-TR-646.html, 2005

[8] Fei, X., and Magill, E. (2008) 'Rule Execution and Event Distribution Middleware for PROSEN-WSN', Second International Conference on Sensor Technologies and Applications (SENSORCOMM), 580-585

[9] H. Li, M.C. Price, J. Stott, and I.W. Marshall, "The Development of a Wireless Sensor Network Sensing Node Utilising Adaptive Self-diagnostics", in Proc. IWSOS, 2007, pp.30-43.

[10] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman, LIME: A coordination model and middleware supporting mobility of hosts and agents: Vol 15 , Issue 3, pp: 279 – 328, July 2006

[11] A. L. Murphy and G. P. Picco. "TinyLIME: Bridging Mobile and Sensor Networks through Middleware", PERCOM (Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications), pp: 61 – 72, 2005

[12] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notication Service", ACM Trans. on Computer Systems, 19(3):332-383, Aug. 2001.

[13] P. R. Pietzuch and J. M. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture", In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 611-618, Vienna, Austria, July 2002.

[14] C. Zhang, M. Li and Q. Pan, "An ECA Rules Based Middleware Architecture for Wireless Sensor Networks", Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT), Pages: 586 – 588, 2005

[15] BNF: http://en.wikipedia.org/wiki/Backus-Naur_Form

[16] K. Terfloth, G. Wittenburg, and J.Schiller, "FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks", First IEEE International Conference on Communication System Software and Middleware (COMSWARE 2006), New Delhi, India, January 2006

[17] K. Terfloth, G. Wittenburg; and J.Schiller, "Rule-oriented Programming for Wireless Sensor Networks", International Conference on Distributed Computing in Sensor Networks (DCOSS) / EAWMS Workshop, San Francisco, USA, June 2006
[18] K. Terfloth. "Doctoral Dissertation: A Rule-Based Programming Model for Wireless Sensor Networks", Freie Universitat, Berlin. June 2009.
[19] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks", International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, October 2002.
[20] X. Fei and E. H. Magill Language Definition for REED, http://www.cs.stir.ac.uk/research/publications/techreps/pdf/TR189.pdf, May 2011
[21] G. A. Campbell and K. J. Turner. "Policy Conflict Filtering for Call Control, in Lydie du Bousquet and Jean-Luc Richier (eds.)", Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems, pp. 83-98, IOS Press, Amsterdam, May 2008.
[22] S. L. Keoh, N. Dulay, E. Lupu, et. al, "Self-Managed Cell: A Middleware for Managing Body-Sensor Networks", Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, Aug. 2007.
[23] S.L. Keoh, et. al. "Policy-based Management for Body-Sensor Networks", 4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN), 26 – 28 Mar 2006.
[24] Gumstix: http://gumstix.com/
[25] JamVM: http://jamvm.sourceforge.net/
[26] JESS, the Rule-Engine for the Java platform, http://www.jessrules.com/jess/index.shtml
[27] J. Y, Chen, G. Pandurangan, and D. Xu, Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis, Fourth International Symposium on Information Processing in Sensor Networks, pp: 348 – 355, April 2005.
[28] G. Baillie, Data Aggregation within a Rule-Controlled Wireless Sensor Network, Honours Dissertation, University of Stirling, 2008.
[29] G. A. Campbell and K. J. Turner. "Goals and Policies for Sensor Network Management", Proc. 2nd Int. Conf. on Sensor Technologies and Applications, pp. 354-359, IEEE Press, August 2008.