A design approach for application specific processing in wireless sensor networks

Arthur Holstvoogd
arthurholstvoogd@me.com

ABSTRACT
Programming applications for wireless sensor networks is becoming increasingly more complex, leading to the development of middleware to abstract these complexities. In this paper an approach combining customizable in-network processing, application knowledge injection and online configuration of sensor nodes in one design is presented and evaluated.

Keywords
Wireless sensor networks, middleware, context-aware systems, distributed computing, pipes-and-filters, composite filter pattern, in-network processing, application knowledge, configuration injection

1. INTRODUCTION
Wireless sensor networks are networks of many small sensors networked through a wireless network and can be used in a broad spectrum of applications. Current application avenues researched range from military applications, such as target detection and tracking, through research applications, like habitat and environment monitoring, to more general purpose applications, like health monitoring, home automation and traffic control. However, there is a lack of cheap and general purpose sensor nodes which holds back the development of any large-scale commercial applications.

Many potential applications can be seen as context-aware applications which attempt to enrich the user experience by adapting to changes in the environment. Wireless sensor networks form the ‘eyes and ears’ of these applications and provide the applications with the context information.

1.1 Wireless sensor network middleware
Implementing applications to interact with a wireless sensor network is far from trivial since the nodes provide only a very limited abstraction from the hardware level. A general purpose platform that provides easy access to the functionality of the sensor network could make the development of applications easier. Such a platform should provide a clear interface for the application developers to work with and offload the complexities of operating the nodes and network away from these applications.

Such a platform, an intermediate layer between the application and the operating system of the nodes and network, is commonly called middleware. The increase of interest in the development of wireless sensor networks and applications that use them has spurred the development of multiple middleware solutions for wireless sensor networks as well as the writing of multiple review papers in just a few years.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission.

11th Twente Student Conference on IT. Enschede 29th June, 2009
Copyright 2009, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science

However, as shown by Henricksen and Robinson in their review of middleware for wireless sensor networks, the approaches taken all seem to be a bottom-up approach that place emphasis on the limitations of the sensors[9]. But any application of wireless sensor networks will be in the context-aware domain, for which middleware has always been developed with a more top-down approach, focussing on the applications needs rather than the available resources.[2]

The two opposing approaches taken in the design of context-aware and wireless sensor network middleware resulted in a mismatch between the two categories of middleware and little recognition of the parallels between them or the research in their direction. There is however a close relation between the two categories, many of the problems are shared or at least very similar[8]. Exploring this relation might provide some solutions for problem still common in wireless sensor network middleware but already solved for context-aware middleware[9].

1.2 Middleware features
Masri and Mammeri reviewed current wireless sensor network middleware designs and found all have a different subset of design principles and features, but none offer a full solution[12]. Some enable in-network processing, some deal with dynamic network topology. Some provide security, some have quality of service support. Only one feature can be found across the board - resource and power management, which shows the focus on the hardware limitations of the sensors in the design considerations.

The following twelve features are found by Masri and Mammeri in current middleware designs:

Data-Centricity In sensor networks the nodes themselves, i.e their identities, are not relevant, only the data they collect and possibly the location and time it was collected.

Energy & Resource Management Wireless sensor nodes have very limited computational power, memory and energy to work with and can easily be depleted. Middleware needs to be able to manage these resources.

In-Network Processing In a distributed set-up the nodes need to actively take part in processing sensor data and decision making on network operations. Middleware should provide data fusion, encoding and compression functions and also allow for the injection of application specific functions.

Quality of Service Support Since nodes need to be able to process data of themselves and other nodes; and provide a data-centric interface, they have to be aware of the data in the network and the quality of the data.

Application Knowledge To minimize the footprint of the middleware and still allow for a wide variety of uses, the middleware should allow for the injection of application knowledge into the network. This enables the customization of
the network functions and operations to the requirements of the application.

**Scalability** Wireless sensor networks are currently relatively small, but in real applications the number of nodes could easily surpass thousands. Scalability is clearly an issue that needs to be tackled.

**Dynamic Network Topology and Robustness** Nodes are not very reliable and may fail or disappear at any moment requiring the middleware to change routes and fill in blanks where possible.

**Adaptability** Over time not only the conditions in and around the network can change, but also the requirements of the application can evolve. The middleware should allow for applications to adapt their requirements on data when such changes occur and thus adapt the network to these changes.

**Configurability & Maintainability** The middleware should allow for remote configuration and maintenance of nodes since they can be expected to be deployed for long periods of time in which the nodes cannot be retrieved for manual configuration. Also the potentially large quantity of nodes makes it simply unacceptable to require manual configuration and maintenance.

**Security** Security is a complex issue in wireless sensor networks: nodes are deployed in a potentially hostile environment and can be accessed physically; encryption can require heavy computations; many nodes can be involved in the end-to-end communication and the limited power makes nodes vulnerable to attacks designed to deplete their power.

**Heterogeneity** Like any distributed system a wireless sensor network can consist of many different nodes. They can be different in the resources they have, their sensing and processing capabilities and even their hardware platform.

### 1.3 Problem Statement

The most obvious problem areas are in dealing with heterogeneous networks, providing a secure network and supporting Quality of Service considerations[12]. These requirements are only fulfilled by few designs and even then only to a very limited extent. These three problem areas stand out because they are very well defined, other problem areas might exist that have been ignored in the reviews because it is unclear what the requirements are for them.

Three features that seem skimmed over in Marsi and Mammeri’s review are in-network processing, application knowledge injection and network configuration. The definition of these three features given in the review is very generic and vague, has overlap with other features and ignores the relation between the features. This seems to be the case because the three features are not found together in any one of the current designs.

All three are prerequisites for a truly distributed, adaptable and light-weight middleware system. More importantly though, they share a pattern: the injection of application information and logic, or knowledge, into the network. All with the goal of adapting the network to the application requirements, either by providing the knowledge needed for the network and middleware to adapt to the applications requirements, or by directly modifying the configuration and code used by the network and middleware.

The approaches to applying the injected information might differ, the ultimate goal and the means for injecting it are not. When the actual application of the information is taken out of the equation, the first area of overlap becomes evident: the injection of information into the network. A second similarity is between the in-network processing and use of application knowledge, albeit less obvious.

Using application knowledge aims at tuning network and processing operations to the current application requirements. By combining both application knowledge and customizable in-network processing, this can be taken to a next level. The processing can be adapted to the exact application that runs on-top of the network and be built to tune very exactly to the applications requirements injected into the network, thus providing much higher specialization then a design that provides only one of the two.

To harness the potential from these two areas of overlap potential, a new approach is needed that fits with the nature of wireless sensor networks and also accommodates the way context-aware applications interact with middleware. It has to be distributed, scalable and dynamic to fit with the network, but also extensible and flexible to fit with the applications. The goal of this paper is to provide a framework for designing these bits of a middleware solution in a way that fits with these properties.

### 1.3.1 Research method and content

First, based on the reviews of existing middleware for wireless sensor networks as discussed in section 1. A set of requirements the design will have to satisfy is then defined in section 2, followed by the actual design in section 3, based on these requirements and the sensible application of patterns and best-practices.

The design is then validated by testing a prototype implementation with test cases for all prototype functionality and a basic example application. The prototype is a very limited implementation scoped to only include the bits designed in this paper as discussed in 4.1. Based on the outcome of these tests and any relevant experience from implementing the prototype, the design is evaluated in section 4.3.

Finally, in section 5 the results of this paper are discussed and some future research suggested.

### 2. REQUIREMENTS

The main goal of the design is to provide a framework for building middleware that supports a sub set of the features discussed in section 1.2, specifically: support for in-network processing, application knowledge based network operation and network configuration injection. These three features introduce three indispensable requirements as follows, which are the starting point for further, more detailed, functional requirements defined in sections 2.1 through 2.3.

**R1** Support the use and injection of application knowledge.

**R2** Support flexible in-network processing of sensor data.

**R3** Support the distribution of configuration into the network.

However the features found in section 1.2 are heavily intertwined, some precede R1, R2 and R3; and some affect the realization of R1, R2 and R3. The design needs to accommodate the inherent heterogeneous and dynamic character of the networks; scale properly when used in large networks; be adaptable to changes in the applications needs; and most importantly understand the scarcity of resources available to nodes.
R4 Minimize resource consumption.

R5 Support the dynamic and heterogeneous nature of wireless sensor networks.

R6 Scale from very small to very large networks.

R7 Adapt to changes in the applications needs and in network conditions.

R4 lies at heart of R1, R2 and R3, by tuning network and sensor operations to the exact needs of the application, power usage can be scaled down. Moreover, since wireless communication is, in general, far more power intensive than computation, processing as much data on the node itself also serves to preserve power.

Moreover R7 partially encapsulates R1, R2 and R3. Adapting the network to changes in the applications requirements requires application knowledge to be injected, and updated. Being able to dynamically customize the processing of sensor data could just be the ‘next step’ in this approach.

R5 and R6 are less concrete and have effect across many parts of the design.

2.1 Application knowledge

Application knowledge is a broad concept and can be used in many aspects of middleware, one of the main goals of injecting it into the network is to tune the network parameters to the applications requirements and thus increase the networks efficiency and reduce power consumption[6]. MiLaN[7] is a middleware design that focusses on application knowledge and Quality of Service requirements by continuously letting nodes update their configuration to match with the applications QoS requirements.

Mapping the QoS requirements to node configuration is beyond the scope of this paper, but application knowledge goes beyond QoS requirements. The principle of continuously adapting the network configuration to the application can also be applied to the in-network processing, especially in a distributed approach where application specific aggregation and caching are possible[14]. In section 2.2.1 the actual implications on processing are discussed further.

R8 Apply application knowledge based tuning to in-network processing.

Finally there is the question of injecting the actual application knowledge into the network. This can directly be linked to the configuration injection discussed in section 2.3.

R9 Include application knowledge in the configuration for injection.

For the purposes of this paper the inclusion of application knowledge in the processing and configuration parts, as specified in 2.2.1 and 2.3, satisfies R1.

2.2 In-network processing

There are many algorithms and approaches for sensor data processing. The processing algorithms can range from signal processing and filtering to data transformations and classification.

Besides supporting some generic processing functions, the design will need to support the use of custom algorithms for processing[12]. To prevent duplication of functionality in these algorithms, it should be possible to compose multiple algorithms into one.

R10 Support custom algorithms for processing sensor-data.

R11 Support composites of multiple algorithms.

To be able to easily reuse these algorithms and use them independently of the sensor nodes involved, the algorithms should be self-contained modules and decoupled as much as possible from the actual nodes and network. By enforcing a common interface for the custom algorithms and the generic algorithms, they become interchangeable which would improve the flexibility of the processing mechanism.

R12 All processing algorithms should be contained in interchangeable modules with a common interface and be fully decoupled from the physical nodes.

Again the configuration injection mechanism can be used to inject the custom algorithms into the network, producing the final requirement to satisfy R2.

R13 Include custom algorithms in the configuration for injection.

2.2.1 Distributed processing

As mentioned in section 2.1 application knowledge can be applied in processing sensor data, amongst other by introducing customized aggregation and caching. This requires the customization of processing to be applied on the network level instead of just on a single node and for the processing customization to be based on application knowledge.

To bring the custom processing algorithms to the network level, a distributed process that governs the data flow and processing is needed. It should be possible to combine data from multiple neighboring nodes and apply more processing algorithms to aggregate and filter the data. Decisions about the data flow between nodes and caches, what is to be propagated, stored or dropped, should be made based on application requirements[6].

R14 Support a distributed process that controls the flow of information through the network and further localized processing based on application knowledge.

An approach for specifying the processing beyond the confinement of one node, is to define it through global and localized behavior[5]. The behavior of the entire network or groups of nodes is programmed rather than the behavior of a single node. This approach allows for high-level specification that is not contingent on the actual network topology or specific properties of nodes, but concepts like regions based on connectivity or geographical location[17].

This does not only allow the application developer to work with concepts that are relevant to the application domain, but also provides a proper fit with R5 and R6 as it is not directly linked to the physical network.
R15 Provide an abstract interface for defining the distributed processing behavior of the network.

This approach is closely linked to event detection and target tracking applications, but other applications could require direct streaming of data[1]. Depending on the application and network specifics, any of these two approaches could provide the most efficient balance between computational and communications load. A middleware platform should not limit the application developer in which approach is used for extracting the needed data from the network.

A third, more fundamental, approach to extracting information from the network is direct querying of data for a specific set of nodes. This approach is employed in most of the middleware designs that have a purely data-driven design[12].

R16 Support event detection and data streaming for extracting information from the network, in addition to directly querying a set of nodes.

The actual specification of processing behavior can be included in the configuration and should also include the configuration of composite processing algorithms as required by R11 to consolidate the processing configuration and definition concerns.

R17 Include the definition of the in-network processing behavior in the configuration for injection.

The addition made to in-network processing in this section R8 is defined in more detail and, in line with R4, resource consumption can be reduced again by enabling more localized processing tuned to the exact requirements of the application.

2.3 Configuration injection

Once a network is deployed the nodes are not retrievable for manual reconfiguration and in large networks the amount of work involved in manual configuration is simply unacceptable, so a mechanism is needed for dynamically updating the node configuration. The scope of these configurations goes beyond basic settings like, for instance, the radio channel or frequency to use for communication and should also include the application knowledge and customizations required by R9, R13 and R17.

R18 Support the injection of settings, application knowledge and custom code through one mechanism.

Processing algorithm could rely on application specific configuration, the format of the configuration should not limit the extend of these settings in any way. A format is required that can easily be extended to include different sets of configuration without becoming ambiguous.

R19 A flexible and extendible format shall be used to express the full configuration.

There are two scenarios for the configuration of nodes, the first is a full (re)configuration which can occur when a new node is added to the network, a node somehow has a corrupted configuration or reconnects to the network with an outdated configuration. The second is an update of the existing configuration, to minimize load on the network, these updates should not have to include the full configuration but just the configuration changes that have to be applied.

R20 Support (re)initializing nodes with a full configuration.

R21 Support updating existing configuration with just the changes to be made.

Some updates may interfere with the functioning of the network and might lead to unwanted behavior if the nodes are actively running, thus requiring the entire network to be updated at once. On the other hand, some updates will not have such an impact and can be applied one node after another. To minimize the impact of updates on network operations the mechanism should support both updates that need to be done synchronized and updates that can take place asynchronously.

R22 Configuration updates can either be applied synchronously or asynchronously.

Within the scope of this paper, these requirements for the configuration injection mechanism cover R3.

3. THE DESIGN

In this section a design based on the requirements defined in the previous section is presented. There are two aspects to the design, the first is the in-network processing using custom algorithms and application knowledge; the second is a mechanism for the injection of configuration, application knowledge and custom algorithms.

3.1 Processing mechanism

The processing mechanism of the design needs to satisfy R2 and some aspects of R1, to do this more than simple custom functions are needed. R16 requires the mechanism to support querying and streaming of data; and on top of that event detection and handling. When these requirements are combined it becomes clear one mechanism is needed that incorporates the entire processing chain from sensor to application.

Querying data and streaming data are rather similar, except that the former might have some parameters to refine the results and is triggered once by the application while the latter is triggered in the network itself on a fixed interval. However, in both cases some data has to be reported to the application and to minimize the communication load this data has to be processed first. To satisfy R14 it has to be possible to process data in transit from a node to the application again on every intermediate node.

Event detection does not differ much from streaming data beyond the characteristics of the actual processing and the semantics of the data received by the application. Where a streaming process always reports the value of some variable(s) at a steady interval, an event detection process only reports back to the application when some specific conditions are met, i.e. an event is detected.

A pattern specially designed for processing streaming data and sensor data is the composite filter pattern, which is used as the foundation of the design. The pattern, where multiple filters are applied to data and the output of the one becomes the input of the next filter, allows for very flexible designs and modular, extendible implementations[3]. To adapt to the distributed context the pattern can be extended to explicitly include pipes, used to encapsulate more complex communications, from one of its ancestor, the pipes-and-filters architecture[13].

This pattern do not fully fit with the context of a wireless sensor network yet, since it does not explicitly support branching and
aggregation. Aggregation is needed because there can be multiple data sources, branching is required to have control of flow in the process as required by R14 and be able to stop the processing as required to support event detection. The extensions needed to accommodate these requirements are described in section 3.1.2.

The actual build-up of the processing chain, that identifies which pipes connect to what filters etc, needs to be configurable through a clear programming interface, as per R15, in a format that can be included in the configuration injected into the network.

### 3.1.1 Custom filters and pipes

Filters can take many forms, some will perform low-level tasks like filtering and processing raw sensor data, some will operate at a much higher level and perform much more abstract tasks like deciding if an event has been detected or not. The custom algorithms as required by R10 can be used as such filters, allowing for filters that are fully adapted to the specific application.

Because filters, as used within a composite filter pattern, have a standard interface, they can easily be reused in different situations, thus satisfying R11 and R12[3]. However to allow the application developer to adapt the filters to multiple contexts, the filter will need to be configurable. By including configurations for filters in specific parts of the processing chain, highly reusable filters can be developed and used in multiple applications.

Pipes have as only task to transport data from one filter to the next, they do not modify or even inspect the data they are transporting. In most cases pipes connect two local filters and perform nothing more than a simple function call, but some pipes will perform more complex tasks like connecting to filters on one or more different nodes in the network.

Again, to enhance reusability and enable the tuning processing, it should be possible to configure the pipes and use custom code for pipes in the same manner as this is possible for filters.

Both custom code for filters and pipes is included in the configuration injected as described in section 3.2 without any context as to the actual processing chain. However, the further configuration of both filters and pipes should be included in-line with the configuration that specifies the chain because this immediately provides the context in which the configuration is to be applied.

### 3.1.2 Aggregators and routers

Extending the pipes and composite filters pattern to include aggregation and branching can be accomplished with some special filter patterns commonly found in enterprise integration solutions: an aggregator that collects multiple messages from multiple input pipes and forwards them once a sufficiently complete set is collected; and a router that has multiple output pipes and can choose to which of them to send received input or drop it if none is relevant.[10]

The exact behavior of these filters needs to be configured to suit the application and should thus be part of the same configuration that specifies the filter chain. This allows the same basic design to be used as is needed to configure custom filters and since the filters have a generic interface, the application developer is even able to create a fully customized versions of aggregators and routers, or even create new filters like it.

### 3.1.3 Processing interfaces and model

The composite filter pattern enforce a strict model and interface for filters, but it does not include pipes by default. The model, extended with pipes, has been specified in the class diagram found in figure 1. The application of this pattern in the context of in-network processing has some side-effects on the exact functioning of certain elements in the model. The role of the nontrivial elements is explained in the following list.

**SimpleFilter** This is a special type of filter that does not send the data it processed any further, it effectively is the sink of a pipeline of filters. In the wireless sensor network this would typically be the end-point in an application or the end of a branch where some side-effects on the node are needed.

**CascadingFilter** This filter is used in the actual pipeline, it is the only type of filter that can forward the data it processed through a pipe. Ideally these filters have no side-effects beyond altering the processed data.

**CompositeFilter** This filter uses composition to combine multiple child filters. Each of these filters is applied to the data the composite filter receives, effectively splitting the pipeline into multiple pipes when cascadingFilters are used as child filters. SimpleFilters can also be used as child filters, however since they only function as sink this would only make sense if the filter has some side-effect on the node it is executed on.

**Source** This is where the processing starts, it knows the first filter to apply when data is sensed.

![Class diagram of the composite filters and pipes design](Image)

**3.2 Injection mechanism**

To satisfy R3 the mechanism needs to efficiently, but more importantly effectively inject configuration into the network. Similar problems have been solved for high-availability clusters in CoroSync, a membership and configuration engine for clusters, which can initialize and synchronize configuration in a cluster of many nodes[15]. A similar approach can be used as a basis for the configuration injection mechanism needed here.

The format needed to satisfy R19 can be pretty much any format that has the desired attributes, but for clarity a standard format
would be best suited. An example of a suitable format is JSON, it is very flexible and custom code can be serialized and inserted as text[4]. A similar format assumed throughout the rest of the paper where relevant, but the choice ultimately depends on the implementation language used and the programmers preference. R18 can be satisfied by including allowing configuration to include serialized code and have flexible schema.

To satisfy R20 a node must be able to provide any other node with the full configuration if it is new to the network or has an outdated configuration, which could occur if the node ‘misses’ some updates. This is reflected in the state diagram of nodes in figure 2 by the transition from clean through unsynced to active and described in pseudocode in listing 1.

---

**Figure 2: State diagram of the configuration of a node**

- **clean** Node is not yet connected to the network, attempt to join the network.
- **unsynced** Connected to the network but not fully configured yet, request last version from neighbor.
- **active** Processing and forwarding asynchronous updates, operations running.
- **syncing** Operations halted, waiting to apply update and forwarding updates.
- **dead** Any state in which the node cannot process configuration updates.

To satisfy R21, once a node is configured, it should be able to receive and apply updates to its configuration. These updates need to be pushed through the network, which possibly needs to be synchronized and ‘paused’ before the updates are applied, as specified in R22. The update process is based on the two-phase commit protocol of the synchronized updates, the algorithm is outlined in listing 2[16]. The process is also reflected in the state diagrams for both nodes and the network configuration in figures 2 and 3 by the sync related states.

---

**Figure 3: State diagram for the network during synced configuration**

- **active** Normal network operations running.
- **syncing** Paused operations while update propagates and nodes synchronize.
- **synced** All nodes report ready for update.

This outlines the configuration injection mechanism, details like the exact syntax of the configurations and updates can be determined when implemented as they have no impact on the architecture of the injection mechanism.

## 4. EVALUATION OF THE DESIGN

The design is evaluated with a prototype and a set of test cases to test if the requirements are met by the prototype. These test cases cover all functionality of the design included in the prototype. The prototype development approach and testing process are discussed in section 4.1, followed by an evaluation of the implementation process in section 4.1.2.

In addition to the test cases used during development of prototype a test suite based on an example of application usage is included to illustrate the applicability of the design. The example is specified in detail in section 4.2.

Finally, the results of testing, both from the basic test cases and the example application are discussed in section 4.3. Based on these results the design is validated in section 4.3.1.

### 4.1 Prototype

For the development of the prototype part of the Ruby on Rails web-development framework, which is specially know for enabling rapid prototyping and test driven development of web applications, is used. RoR uses a Model-View-Controller pattern and has a testing framework included. For the purposes of this prototype only the model part of MVC is used since there is no need for a web-based front-end.
Since the design scope is limited to only a part of what makes up a full middleware design, the prototype is too. All other aspects, like communications, routing, security etc, are ignored in the development of the prototype. Only a very basic model of nodes is used to form a network, in which nodes only have configuration, a processing chain and some functionality to simulate sensing data. This effectively produces an executable and testable version of the design.

The prototype does not have an user interface that allows direct interaction and output, but provides a model that can be tested using automated unit testing tools. Although this produces less tangible results as there is no interaction with the user, the results are more formal and thus allow for a more formal validation of the design.

4.1.1 Testing
The testing of the prototype is done through test driven development, for every piece of functionality some test cases are defined before the functionality is implemented. Once these test cases can all be successfully executed, the functionality can be considered fully implemented. To ensure the test cases cover all the aspects of the functionality a DSL closer to natural language is used to specify the behavior, which makes it easier to reason about the completeness of the test cases.

Each test case is defined as some behavior that should occur in a certain context. The context is setup from scratch for each test case, starting with the setup of a clean network of one hundred nodes, followed by the injection of any relevant configuration in a nested manner until the exact state is reached in which the behavior can be tested.

Once the context has been fully setup, providing the preconditions for the tests, the behavior in this context is tested as with a set of assertions testing the state of the network. For instance, after setting up a network, applying a base configuration and then followed by an configuration update, it is asserted whether all nodes in the network have the latest version of the configuration and the changes specified in the update have actually been applied.

An overview of the relevant test cases can be found in appendix B. The context can be derived from the naming and the assertions made for each case are explicitly described. Beyond these test cases, more low level tests are done to ensure specific methods are working, these serve only to prevent the introduction of bugs during the development of the prototype and do not provide added value beyond that.

The exact implementation of tests is included with the prototype sources, as well as instructions on executing the test suite[11].

4.1.2 Implementation
During the implementation of the prototype, it became clear not all aspects of the design are specified in enough detail for a direct implementation, leaving these details to be filled in during the implementation. This led to some unforeseen complications and obstacles during the implementation of the prototype. Fortunately, only one obstacle was insuperable in the time available: the composition of filters.

In principle composition of filter, as used in the composite filter pattern, is not a very complicated aspect to implement, but when combined with a distributed processing chain, it added a lot of complexity and was thus left out of the prototype. Although this limits the extent to which the design can be validated, it is not strictly necessary to show the working principles of the design as it only serves to make more complex filter chains possible. This, in turn, is sufficiently shown by the composite filter pattern itself.

Another source of trouble during implementation was the lack of isolation between nodes. Since the design assumes a truly distributed context in which every node will have its own memory and object space, there was a definite mismatch with the prototype where the nodes all are mere objects in one object space. This lead to some unexpected behavior and required some extra considerations for instance in implementing the use of custom code elements.

4.2 Application Example
To prove the applicability of the design, a second set of test cases is developed based on an example application using the prototype. The application is aimed to make use of all aspects of the processing mechanism and the test cases also make full use of the configuration mechanism. The relevant part of the application is the configuration of the network, this includes the definition of all application specific filters and the filter chain. The example configuration can be found in listing 3.

Listing 3: Configuration for example application

```python
# The definition of the needed filters and pipes
definitions = {
    ' salari': 'CascadeFilter',
    'name': 'TemperatureChangeFilter',
    'code': '{\n        factor: \[data_last, data_first\],
        location = \[self.node.x, self.node.y\]}.
}

# The definition of the filter chain
filter_chains = [
    'temperature',
    'location',
    'neighborhoodPipe',
    'agggregator',
    'Pipe',
    'endPoint',
]
```

```python
# The definition of the filter chain
filter_chains = [
    'temperature',
    'location',
    'neighborhoodPipe',
    'agggregator',
    'Pipe',
    'endPoint',
]
```

```python
# The definition of the filter chain
filter_chains = [
    'temperature',
    'location',
    'neighborhoodPipe',
    'agggregator',
    'Pipe',
    'endPoint',
]
```
of testing. Higher coverage provides better tested code and thus a lower error rate and a higher failure detection rate. The overall code coverage for the prototype is 98.8%, which means only 1.2% of the application code has not been executed in the tests. Typical coverage targets are between 90 and 95%, with 98.8% the prototype is has a very high level of test coverage.

<table>
<thead>
<tr>
<th>Listing 4: Test execution output and metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test results:</strong></td>
</tr>
<tr>
<td>Total: 70 tests with 189 assertions</td>
</tr>
<tr>
<td>Code Coverage:</td>
</tr>
<tr>
<td>Node</td>
</tr>
<tr>
<td>node</td>
</tr>
<tr>
<td>sample_filter</td>
</tr>
<tr>
<td>cascade_filter</td>
</tr>
<tr>
<td>abstract_filter</td>
</tr>
<tr>
<td>abstract_pipe</td>
</tr>
<tr>
<td>abstract_aggregator</td>
</tr>
<tr>
<td>node</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

| Code statistics: |
| Name | Lines | LOC | Classes | Methods | M/C |
| Models | 303 | 260 | 8 | 3 | 31 |
| Example application | 409 | 351 | 4 | 1 | 1 |
| Unit tests | 533 | 632 | 4 | 1 | 0 |
| Total | 1248 | 1043 | 11 | 33 | 2 |

| LOC: 260 | Test LOC: 783 | Code to Test Ratio: 1:3.0 |

4.3.1 Validation of the design

By analyzing the extend to which the requirements are met in the prototype, the design can be validated. To aid this, the functional requirements are matched to the test cases in an traceability matrix in appendix A. Besides from the lack composite filters obviously leaving R11 uncovered, all functional requirements are covered by the tests. However, not all requirements are met to the extend intended, the caveats left there became apparent during implementation of the prototype.

First of all, not implementing the composite filters not only prevents the design from being fully validated against R11, but also limits the extend to which especially R14 is met. The composite filters are a prerequisite for adding branching to the processing chain, thus missing them limits the processing possibilities of the prototype.

Secondly, the specification of processing chain, its configuration and the definition of the custom filters and pipes, was not very user-friendly in the prototype. A proper format for this should have been part of the design as it is the access point for application developers with the core functionality of the design. Moreover the design should have included several default filters and pipes to assist in the rapid development of applications.

Another concern that emerged in relation to the custom processing features was debugging the custom algorithms. This proved rather hard and confusing in the prototype. Some facilities should be provided to make debugging easier for the application developer, not only for the custom algorithms but for the entire processing chain as well. Ideally it should be trivial to track data though the processing chain and see what happens where.
5. CONCLUSION

Reviews of current wireless sensor network middleware designs show all designs seem to focus on their own specific set of features. We argue a lot of these features have overlap and an approach that takes the overlap of some of the features found across different design can be beneficial. Although the reviews suggest the obvious areas to focus research on are security and Quality of Service concerns, we argue these standout because they are very well defined and go one to select three more ambiguous features to research, specifically in-network processing, the injection of application knowledge and the online configuration of nodes.

We then show the overlap between these three features and develop a set of requirements for implementing them. Based on these requirements we present a design using on well-known patterns that fit with the context and problems that need to be solved. The design provides a framework for injecting configuration, application knowledge and processing algorithms through one mechanism and an generic framework for implementing application specific in-network processing.

To validate the design we develop a well tested prototype and implemented an application on it. Based on the results of testing and the experience of implementing the prototype we can conclude that the design provides a workable approach for implementing the three features we wanted to include, but also the prototype is far from any real implementation and the design only contains a small part of what would be needed to develop a real implementation.

5.1 Future work

As discussed, the presented design provides only a limited set of features and there are many more aspects to include before any real implementation is possible. Even the aspects that are part of this design can be designed in far more detail and tested with far more substance. This clearly points out a large bulk of future work, but some more concrete research directions can be found as well.

The design is heavily based on filters and pipes which are commonly used in security patterns as well. It could be relatively easy to implement a security scheme on top of the pipe and filter system used in this design, for instance by using secure pipes for communication and using filters to prevent leaking private data. It might also be possible to implement other features, like Quality of Service, using the pipes and filters. Concern like enabling customized processing etc for multiple applications while still enforcing security and privacy policies, could be part of research in this direction.

A lot of work can also be done in extending this framework to provide useful default pipes and filters. Generic filters could be developed for filtering out sensor glitches, compression and encoding of measurements, automated aggregation of measurements, etc. Pipes to accomplish complex communication tasks, like connecting to a node with a certain responsibility or the application end-point will also be useful in a lot of situation and require low-level access and intimate knowledge about the network.

Providing a set of generic filters and pipes to encompass such tasks, will greatly reduce the effort needed to develop an application using middleware based on this design.

REFERENCES

APPENDIX A: TRACEABILITY MATRIX

The following table shows the correlation between the functional requirements and the test cases used when developing the prototype. Descriptions of the test cases are given in appendix B, the requirements can be found in section 2. The requirements preceding those formulated in this paper and non-functional requirements have been excluded for clarity.

<table>
<thead>
<tr>
<th>Test case</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
<th>R13</th>
<th>R14</th>
<th>R16</th>
<th>R17</th>
<th>R18</th>
<th>R20</th>
<th>R21</th>
<th>R22</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4.1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4.2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1.1</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1.2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

APPENDIX B: DESCRIPTION OF TEST CASES

The following list provides an overview of the test cases used to test the prototype during development. Exact phrasing of test cases may differ from the actual tests to enhance expressiveness. The example application is not included since its test specification is included in the concerning section, 4.2.

1. With a distributed filter chain when streaming data should output data from multiple nodes. Asserts each line in the output contains the expected amount of data; Items all from different nodes; and the expected data for all nodes.

2. A node with a filter chain when
   2.1. detecting events should report event when the configured threshold is met. Asserts all output contains values exceeding the configured threshold.
   2.2. measuring data should apply the filter chain to measured data. Asserts the full chain is applied to the data in that was put out.
   2.3. streaming data with a 0.1 second interval should
      2.3.1. start measuring. Asserts all nodes are running the measuring thread.
      2.3.2. report the measured data on the set interval. Asserts one node outputs approximately 10 ‘frames’ per second.
   2.4. querying a range of nodes, all nodes in the range should
      2.4.1. be queried. Asserts there is exactly one ‘frame’ in the output for each node in range.
      2.4.2. report the measured data. Asserts the outputted data matches the nodes that are in range.

3. A node that failed should have newest configuration after rejoining. Asserts all nodes have the latest configuration.

4. A node that newly joins the network should get the latest configuration. Asserts all nodes have the latest configuration.

5. A configuration update that
   5.1. does not require synchronization should get pushed through the network. Asserts all nodes have the latest configuration.
   5.2. requires synchronization. Asserts no nodes are immediately updated.
      5.2.1. doesn’t fail should get pushed across the network. Asserts all nodes have the latest configuration.
      5.2.2. fails should not get pushed across the network. Asserts all nodes have the previous configuration.

6. Configuration, on evaluation by a node containing
   6.1. a filter and a pipe definition should create
      6.1.1. a filter with the specified custom code and type. Asserts the custom filter exists under the correct name; and applying the filter has the expected output.
      6.1.2. a pipe with the specified custom code and type. Asserts the custom pipe exists under the correct name; and transferring data through the filter has the expected output.
   6.2. a filter chain definition should
      6.2.1. configure the filters. Asserts the filter has the specified configuration;
      6.2.2. create the filter chain for the correct sensor. Asserts a filter chain exists for the sensor;
      6.2.3. create the correct filter chain. Asserts the filter chain has the correct amount of elements; the elements match the specification; and the elements are connected correctly.