A Middleware Framework for the Web Integration of Sensor Networks

Hervé Paulino and João Ruivo Santos

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia, FCT
Universidade Nova de Lisboa
2829-518 Caparica, Portugal
herve@di.fct.unl.pt

Abstract. This paper introduces SenSer, a generic middleware framework that allows for the Web access and management of sensor networks by virtualizing their functionalities as services, in a way that is programming language and sensor network development platform independent. We present the SenSer architectural model and provide a concrete Java-based implementation that exposes the framework as two Web services, cleanly separating regular user operations from administration operations. This prototype implementation has been validated with the development of two applications, and evaluated against the initial functional and non-functional requirements, namely modularity, performance and scalability. We have also performed two integration exercises targeting Callas [17] and SWE [3] networks.

Keywords: Wireless sensor networks, Web services, Middleware.

1 Introduction

Sensor networks are, currently, one of the hottest topics in computer science research, spreading areas as diverse as programming language design and implementation, computer networks, information processing, security, and physical sensor manufacturing, to name a few.

The focus of our work is on Web integration, namely on the remote Web access and management, of such networks. By being commonly deployed at distant and/or unreachable locations (e.g. environmental monitoring), the widespread use of wireless sensor networks is bound to the ability of presenting them, to the application developer, as just another easily composable software component.

The main challenges in this endeavor are to provide a generic Web accessible interface for common sensor network functionalities, and to cope with heterogeneity on both endpoints of the interaction.

The integration of heterogeneous sensor networks is a crucial concern that is still underachieved. There are many operating systems (e.g. TinyOS [14], SOS [13], Contiki [5], and Nano-RK [6]) and programming languages (e.g. Nesc [7], Pushpin [16], TinyScript [15], and Callas [17]) available to develop and support sensor networks, which makes integration in mainstream programming languages a cumbersome task. Moreover, heterogeneity must also be dealt with on the client side. The lack of standardized interfaces further contributes to the difficulty of systematically incorporating sensor networks in everyday applications.

The state of the art in the porting of wireless sensor networks to the Web is almost entirely restricted to the Sensor Web Enablement (SWE) specifications [3] of the Open Geospatial Consortium. As will be further detailed in Section 7, the scope of the SWE specifications is more on the integration of existing sensor networks in a Web of sensors and not on the actual client/network interaction. There are other platforms that enable remote access to a sensor network [10,11], but these are custom-made solutions that address a particular setting, and thus do not contribute to solve the problem at hand. Other proposals, such as [1] and [9], focus on sensor network integration.

In this paper we propose SenSer, a generic middleware framework that allows for the remote (Web) access and management of SENsor networks by virtualizing their functionalities as SERvices, in a way that is programming language and sensor network development platform independent. The featured operations are divided in two categories: *regular user* and *administrator*. Regular users may interact with a network by posting queries, requiring data-streams, subscribing notifications, and performing operations (if the network contains actuator nodes). Administrators are allowed to configure networks by registering and unregistering sinks, and by reprogramming the network's behavior.

The concrete implementation of the SenSer framework relies on the Web Service technology, solving client heterogeneity by providing a Web based interoperable platform. With SenSer, a sensor network can be integrated into an application as just another Web service, hence enabling their inclusion in business processes, by resorting to BPEL [18], a feature most welcomed in businesses whose work-flow comprises the monitoring of merchandise. The heterogeneity of the sensor network endpoint is handled by network specific drivers, which must be compliant with the SenSer sensor network interface.

The remainder of this paper is structured as follows: Sections 2 and 3, present, respectively, the SenSer architectural model and its concrete implementation; Section 4 focuses on an applicational example; Section 5 evaluates the framework against the functional and non-functional requirements; Section 6 presents two integration exercises with Callas networks and with SWE compliant sensor webs and, finally; Section 8 presents our conclusions and guidelines for future work.

2 The SenSer Middleware Framework

The main functional requirements of the SenSer middleware framework are: 1. to virtualize one or more sensor networks as a set of World Wide Web accessible services; 2. to seamlessly integrate multiple heterogeneous sensor networks; 3. to collect data from the registered sensor networks, either in real-time or by retrieving archived data, and; 4. to manage the middleware layer and the registered networks. The main non-functional architectural requirements are: 1. modularity

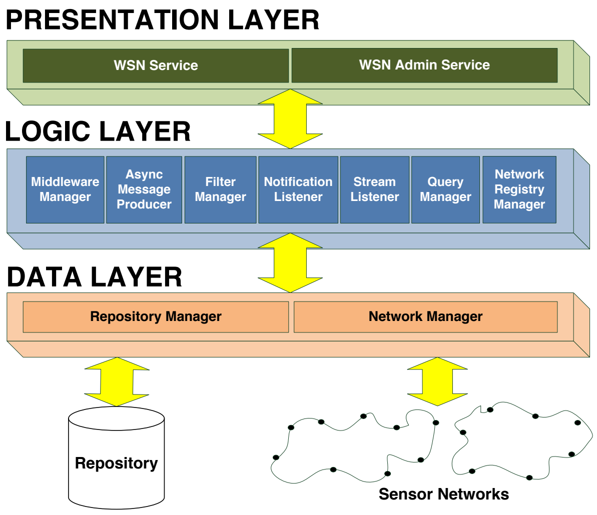

Fig. 1. The SenSer architectural model

in the design and configuration of the framework; 2. scalability in handling of large number of client requests and registered networks, and; 3. performance in the processing of client requests.

The proposed architecture comprises several components organized in a three-tier model that, as illustrated in Figure 1, cleanly separates client interaction, logic, and data management. The lower layer manages the two possible data sources: the registered networks or a repository of previously archived data.

In this section we briefly describe each component, listing the interface of only the ones that are visible from the outside world. The complete specification can be found in [20].

2.1 Presentation Layer

This layer exposes a set of sensor networks as two service interfaces that cleanly separate the operations available for regular users and administrators.

SenSer manipulates two concepts: *network type* that stands for a sensor network type, e.g. a temperature or air humidity monitoring network, and; *sink* that abstracts a sink of a particular network type. Sink identifiers are qualified with their network type (sinkId@networkType) being that, by itself, networkType denotes all the sinks of that given type. snId denotes the whole identifier (sink and network type included) and, for the sake of simplicity, is throughout this text abusively referred to as sink.

Regular User Service: Service WSNService (Listing 1)[1] features the operations available to regular users: getNetworkTypes returns the identifiers of the network

[1] For the sake of readability, the throwable exceptions were removed from all listings.

types currently registered; getSinks returns the identifiers of the sinks of either one or all network types; query posts a query on a given sink; requestStream and requestNotification request a topic upon which the stream or the notification service can be subscribed; subscribe and unsubscribe manage the actual topic subscription; getSensorNetworkInterface retrieves the interface supported by the sinks of a network type and the respective parameters, and; finally executeOperation executes an operation on network, via its sink.

```
interface WSNService {
 NetworkTypeId[] getNetworkTypes();
 SinkId[] getSinks();
 SinkId[] getSinks(NetworkTypeId networkType);
 Observation query(SinkId sink, Query query, Date date);
 Topic requestStream(SinkId sink, Condition condition, Date startDate,
                     Date endDate, int rate);
 Topic requestNotification(Condition condition, Date validityDate);
 boolean subscribe(Topic topic);
 boolean unsubscribe(Topic topic);
 NetworkInterface getSensorNetworkInterface(NetworkTypeId networkType);
 void executeOperation(SinkId sink, Operation op, Parameter[] params);
}
```

Listing 1. Interface WSNService

Conditions are used to parameterize both data-streams and notification requests. In the context of data-streams, the condition, along with the specification of a maximum data rate, operates as a filter on the data to be received. When it comes to notification requests, the condition specifies the scenarios on which a notification event is to be triggered. Notifications may encompass more than a single sink or network type, thus no explicit target sink argument is included. Condition specification borrows the conditional and logic expressions syntax used in Java and C. An example that operates on all sinks of type Temperature and Light follows:

((Temperature > 30 && Light >= 70) || (Temperature < 5 && Light < 20))

Administration Service: Service WSNAdminService (Listing 2) specifies the administration operations. This includes the getNetworks and getSinks operations described above plus: registerNetwork and unregisterNetwork to manage network type registry, the registration requires a set of properties and the network's interface; registerSink and unregisterSink to manage sink registry, the registration requires the sink's network type, an adapter to bridge the communication between the framework and the sink (more on this in Subsection 2.2), and the set of properties to provide, for instance, the location of the sink; install to reprogram a network by deploying a new program in the target sink; setFilter to set the filter for a given sink (more on this in Subsection 2.2) and, finally; operations to manage the configuration (properties map) of the middleware, of a registered networkType, or of a registered sink. We only list the operations available for managing the configuration of the middleware: getPropertyList, getPropertyValue, and setPropertyValue. The remainder are analogous.

Note that the management of the inclusion in the framework of the adapters to bridge sink communication is not included in the interface. This happens because these operations require the uploading of code to the middleware (more on this in Subsection 2.2). This is naturally implementation dependent, for instance a Java-based implementation will not accept C# code.

```
interface WSNAdminService {
 NetworkTypeId[] getNetworkTypes();
 SinkId[] getSinks();
 SinkId[] getSinks(NetworkTypeId networkType);
 void registerNetworkType(NetworkTypeId netType, NetworkInterface interf,
                          Map<String,String> props);
 void registerNetworkType(NetworkTypeId networkType);
 void registerSink(SinkId sink,NetworkTypeId netType, AdapterId adapter,
                   Map<String,String> props);
 void unregisterSink(SinkId sink);
 void install(SinkId sink, byte[] program);
 void setFilter(SinkId sink, Pipeline pipeline);
 Map<String,String> getPropertyList();
 String getPropertyValue(String property);
 void setPropertyValue(String property, String value);
 ...
}
```

Listing 2. Interface WSNAdminService

2.2 Logic Layer

The components of the logic layer provide the core functionalities of the framework. A brief description of each follows.

NetworkRegistryManager: all requests are processed by the logic layer that interacts with the data layer to relay the queries or to perform actions upon a target sink. Each kind of network (e.g. temperature monitoring network) constitutes a network type that may comprise many physical networks composed of one or more sinks, of distinct technologies, each of them explicitly addressable.

```
interface SensorNetworkAdapter {
 String query(String query);
 Map<String,String> getPropertyList();
 String getPropertyValue(String property);
 void setPropertyValue(String property, String value);
 void install(byte[] program);
}
```

Listing 3. Interface SensorNetworkAdapter

Sensor technology specifics are decoupled from the middleware and encapsulated in sensor network specific adapter that has the task of translating the communication on both directions (requests and data). The registry of a sink requires the existence of such an adapter that must be compliant with the framework (respect the SensorNetworkAdapter interface depicted in Listing 3) and with the respective network type. For instance, the adapter for a temperature control network that offers operation int maxTemperature() must implement a method with such signature.

The NetworkRegistryManager component manages the registry of sensor network adapters. How it is made available to administrators is considered an implementation detail and will be dealt with in Section 3.

FilterManager: supports the ability of SenSer to periodically collect data from a given sink and archive it in an history repository. This data is processed as a data-stream subjected to a pipeline of filters, whose purpose is to refine the information to be stored (Figure 2). In the end, the processed information may be either actual sensed data or simply computed statistics. Currently pipelines can be only associated to sinks, through the setFilter method of the administration interface. Ongoing work is extending their application to client requested streams (Section 8).

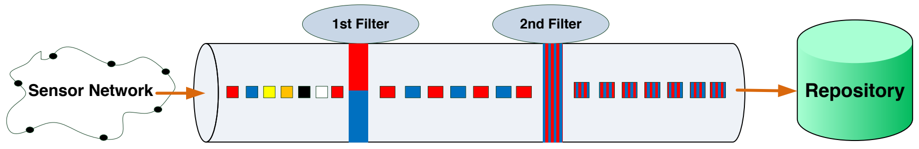

Fig. 2. Filter pipeline

The pipeline specification syntax is presented in Table 1, where pipeline identifiers, filter identifiers, and primitive values are denoted, respectively by p, f, and v. A pipeline (P) is a sequence of filters (F) that are applied sequentially, whereas records can be used to create complex values. Types (T) ensure compliance between two consecutive pipeline stages.

Table 1. Pipeline definition syntax

P	$:= p = F$	Pipeline definition
F	$:= f\ (\ \bar{v}\ ) : T$	Filter application
	$\mid F > F$	Filter composition
	$\mid [\ \bar{F}\ ]$	Record definition
T	$:=$ **String** $\mid$ **Boolean** $\mid$ **Integer** $\mid$ **Float** $\mid$ **Double**	Data-types

The following example denotes a pipeline, named MyPipeline, that aggregates all the data read from the network in clusters of 30 minutes (filter Aggregate with instantiated with value 30) and creates a record holding the minimum and average values (computed, respectively by filters Min and Avg) of each of these clusters: MyPipeline = Aggregate(30) : **Integer** > [Min : **Integer**, Avg : **Integer**]

StreamListener: opens a data-stream between the framework and the client application. According to the dates supplied in the stream's request (present or past) the data is retrieved from the target sink or from the repository. This decision is taken by the component, in a way that is transparent to the client. The date range may even require a rerouting of the data path, when crossing the boundary between the past and the present. The actual emission of the data is performed by the AsyncMessageProducer component.

NotificationListener: supports the subscription and handling of notifications. These events are triggered whenever the condition specified in the request is evaluated to true. Similarly to StreamListener, this component resorts to AsyncMessageProducer to create the notification topics and to publish the events.

AsyncMessageProducer: factorizes the support for producer/consumer relationships. It stores all the currently active data-stream and notification topics, allowing for the client applications to subscribe them, and for the StreamListener and NotificationListener services to publishing their items. The interface includes operations to *create*, *remove*, *subscribe* to, and *publish* on data-stream and notification topics.

QueryManager: processes queries to the registered networks. Once again the supplied date determines whether the query is forwarded to the repository or to the network. In either case, the operation is synchronous, in the sense that it only concludes when the result is sent back to the client.

MiddlewareManager: the behavior of the SenSer framework can be regulated through a set of parameters. This component is responsible for loading these configuration parameters and for allowing their on-the-fly modification, without disrupting the framework's availability.

2.3 Data Layer

This layer comprises three components: one responsible for the storage of the framework's configuration settings, and two more responsible for the integration of the system's data sources: sensor networks and the repository. Only the latter two justify a more detailed description.

NetworkManager: relays the upper level requests to the target sink. The network type identifier is used to retrieve the network type's interface and ensure protocol compliance between the request and the target sink. The sink identifier selects the required adapter (SensorNetworkAdapter implementation) from the network registry.

RepositoryManager: supports the persistent storage of the data collected from the sensor network sinks. It manages the integration of a specific repository (e.g. a database system or a file-system) in the framework, virtualizing it in a service interface accessible to the upper level components. Furthermore, it manages the connections for data storing and retrieval.

3 Implementation

This section presents the most relevant implementation specific details of a Java-based prototype instantiation of the SenSer architectural specification. We center our discussion on the following major topics: overall instantiation of the model; integration with the World Wide Web; sensor network registry; support for data-streams and notifications, and; implementation of filter pipelines.

Model Instantiation: to promote loose coupling between components, allowing these to be autonomous and to run on different machines, we instantiated the three-tier conceptual model in a Enterprise Service Bus.

All inter-component communication is built on top of Java-RMI, hence the bus takes the form of a Java-RMI registry.

Web Integration: all client-platform interaction is based on the Web service technology, providing an Internet-scale interoperable platform. The WSNService and WSNAdminService components are, thus, exposed as two Web services.

To provide a framework upon which it would be easy to access the SenSer middleware layer from Java applications, we have transposed these interfaces to the Java world by implementing a client side API that hides the Web service communication details. The interface of this API is almost identical to the one presented in Subsection 2.1 as can be checked in the example of Section 4.

Network Registry: the registry of sensor network adapters is performed by having a client that directly accesses the NetworkRegistryManager by plugging in the service bus. The registry requires the upload of all the user-implemented classes required by the adapter are uploaded to the framework and locally stored. Ongoing work focuses the integration of these operations in the Java client API by featuring a Web service dedicated to these operations.

Several of the functionalities that must be offered by these adapters crosscut the sensor technology, operating system and programming language, e.g. the properties map. We offer factorize these functionalities in an adapter development kit.

Data-streams and Notifications: both these features are supported by WS-Notification [19], a topic-based publish/subscribe Web service standard. Each distinct data-stream and notification request has a associated dedicated topic in the AsyncMessageProducer component. This topic is used both for subscription (client side) and for publication purposes (components StreamListener and NotificationListener).

The implemented Java client API features a component, AsyncMessageConsumer, that hides the details of the publish/subscribe protocol. As is illustrated in Figure 3, the client no longer has to be aware of the topic subscription protocol, simply posting the request and obtaining the stream or the notification event handler. The latter can be used to associate an action to the reception of the notification, such as display or persistently store them. An example will be given in Section 4.

Figure 3 illustrates the whole stream request process. The client invokes the requestStream method on the API. The request is relayed to the WSNService Web service that, in turn, relays it to the StreamListener logic layer component. A new topic is then created by the AsyncMessageProducer (if one does not yet exist) and sent back to the API, that automatically subscribes it. From this point

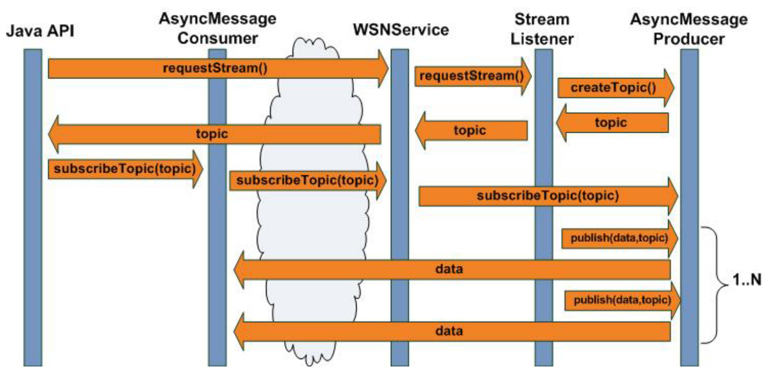

Fig. 3. Data-stream request diagram

on, the AsyncMessageConsumer component receives all the messages published on the subscribed topic.

Filter Pipelines: filters are instantiated as Java classes that implement the Filter interface depicted in Listing 4. Filter provides methods to define the filter's input arguments (setArgs), to program the filter's data manipulation process (process), and to obtain a stream to the filter's output (getStream). The last method is used to connect two pipeline stages, each filter processes the data it reads from the stream generated by the previous stage in the pipeline.

```
  interface Filter {
    void setArgs(String[] args);
    void process(java.io.InputStream in);
    java.io.InputStream getStream();
}
```

Listing 4. Interface Filter

The name of a filter in the pipeline definition string must match the name of the associated Java class. The latter will be dynamically loaded as soon as the string is successfully parsed. Thus, to add a new filter to SenSer is simply to place a Java filter class with the same name in a folder pre-determined by the framework's configuration parameters.

The output of the pipeline must be persistently stored in the repository (a MySQL database in our prototype). For this purpose, a Java class that encapsulates the record to be stored is dynamically generated. This class resorts to the Java Data Objects technology to abstract database records as Java objects.

4 Automated Home Management Application

This section illustrates how SenSer can be used to develop an automated home management application that incorporates lighting control and security systems. We have also used SenSer to interact with a network of medical body-sensors, such as the one described in [2], but the home management application better illustrates the framework's capabilities. The sensor networks used in these examples were simulated.

The lighting control system automatically regulates the lights of the rooms, according to human presence and daylight intensity, while the security system closes all outside doors and windows when no human presence is detected inside the house. Due to space restrictions, the application cannot be fully presented. We will restrict the code to two snippets that cover the registry of network types and sinks, and the management of notifications.

As is illustrated in Figure 4, this implementation assumes the existence of sensor networks for human presence detection and for the measurement of light intensity, both comprising a sink node for each house division. It also assumes the existence of networks to control the intensity of each illumination point (e.g. a X10 module network), and to control the lock of every outside door and window.

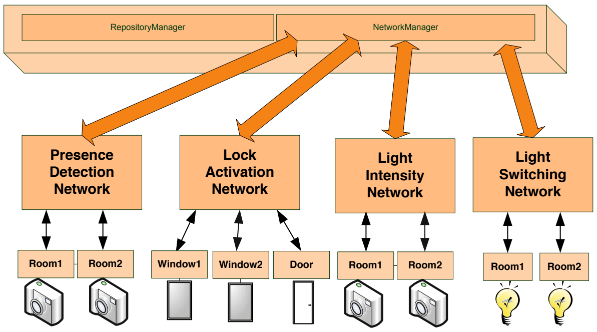

Fig. 4. The networks registeres in the automated management application

Once the frameowrk is equipped with all the adapters needed by the set up sensor networks, the remote interfaces can be used to manage the network types and sink nodes, and to program the intended behaviors. Listing 5 illustrates the registry of the LightSwitchingNetwork network type and its living-room sink. The network interface includes, among others, the switchOn operation that has no parameters. The properties map in the sink registry was used to indicate the latter's contact port[2].

```
SenserWSNAdminService wsnAdminService = new SenserWSNAdminService(senserURL
    );
wsnAdminService.registerNetworkType("LightSwitchingNetwork",
            new NetworkInterface() {{ ...; put("switchOn", null); ...; }})
            ;
wsnAdminService.registerSink("LivingRoom@LightSwitchingNetwork",
               "LSNAdapter", new PropertiesMap() { put("Port", "A1"); }})
                  ;);
```

Listing 5. Sink registry example

[2] The port value range is network type specific.

A notification request returns an handler to which an action can be bound. This binding will cause the AsyncMessageConsumer client API module to associate the behavior to the correspondent notification topic, and trigger its execution whenever a message is received.

Listing 6 exemplifies how to perform a notification request and to bind an action to its reception. These actions are programmed by providing a concrete implementation of the NotificationResponse abstract class, namely of its response method. In this particular example, the response is to switch on the lights on a given house division.

```
NotificationHandler livingRoomHandler = wsnService.requestNotification(
 "LivingRoom@HumanPresenceDetector==false && LivingRoom@LightIntensity <0.4"
      );
livingRoomHandler.setResponse(
   new NotificationResponse(wsnService,"LivingRoom@LightSwitchingNetwork") {
          void response() {
                 wsnService.executeOperation(this.sink, "switchOn", null);
          }
   };);
```

Listing 6. Notification setup and management example

5 Evaluation

In this section we evaluate SenSer against the requirements enumerated in Section 2. The functional requirements have all been met and individually and collectively tested [20]. We just highlight the sensor network adapter approach, which enables the interaction of sensor networks independently of their technology, and introduction of the date field in the user requests, a tool that allows the framework to route the requests to the target sensor networks or to the repository, providing for the transparent access of real-time and archived data.

Regarding the non-functional requirements, special focus was given on modularity, namely on the clean separation between client interaction, logic, and data retrieval and storage. The use of the ESB model was also an important contribution to meet this modularity requirement. Moreover, it enables the physical separation of the components. SenSer does not have to run in a single processor (core) or machine, thus enabling scalability. Another key aspect of modularity was parameterization. The framework is highly configurable and this can be done remotely, through the administration API.

To certify that SenSer does not pose as a bottleneck on user-sensor network interaction, we performed some performance tests. The first test measured the overhead imposed on client requests. Figure 5 presents the mean and the standard deviation for three scenarios: 30, 150 and 300 simultaneous queries. We can observe that for each of these the mean of the overheads is under 25 milliseconds, and the maximum value under 35 milliseconds. We can also observe that there are no scalability problems. the overhead remains steady with the increase of requests.

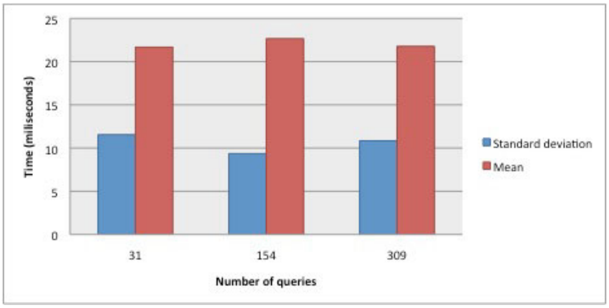

Fig. 5. Overhead imposed on query processing

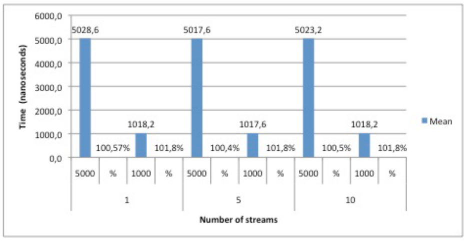

Fig. 6. Mean of the overhead imposed on stream processing

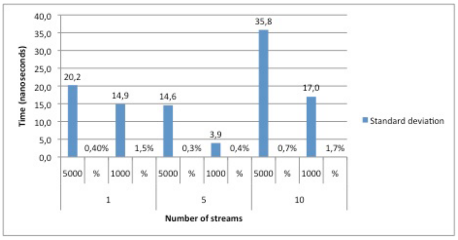

Fig. 7. Standard deviation of the overhead imposed on stream processing

The next two graphics, depicted in figures 6 and 7, present the mean and the standard deviation of the overhead that SenSer imposes on stream processing. We measured the time elapsed between the instant a new data item arrives from a sensor network and the instant it is sent to the client, i.e., the time the platform takes to process it. We analyzed streams requested with rates of 5 and 10 milliseconds. The graphics present the actual rate of the stream after being processed by SenSer and the overhead regarding the original requested rate.

We can observe that the overhead never reaches the 2% barrier. In fact, for streams with rates of 5 milliseconds it is lower than 1%. The impact of these is negligible when taking into account the latency of an Internet connection.

6 Integration

We realized two integration exercises. The first, directed to the Callas WSN programming language [17], consists on the implementation of a Java class that communicates with Callas network sinks. This class can now be extended to be compliant with the SensorNetwork interface and, thus, be used to register Callas networks with specific capabilities.

Callas sink nodes were equipped with a set of system calls that enables them to accept TCP connections at a configurable port. Requests are in the form of

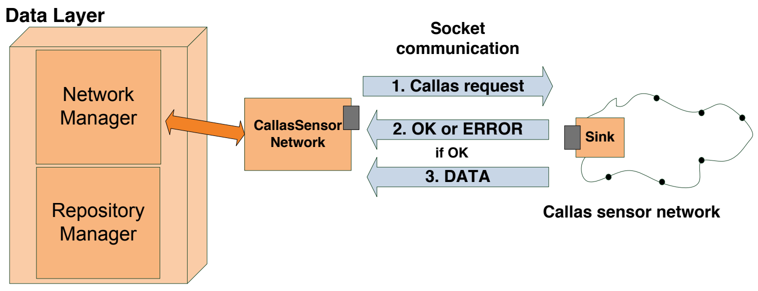

Fig. 8. Callas communication protocol

a Callas program that is submitted to the sink (Figure 8). The parsing of the program will generate an Ok or ERROR response, being that the former spawns the request to the network and replies the consequent result.

The second exercise targeted the SWE specification. It is a simple theoretical exercise that illustrates how SenSer and SWE can be integrated. The integration of a SenSer instance in a SWE network (Figure 9) only requires the addition of an extra component in the SenSer presentation layer, with the task of translating SWE to SenSer requests and SenSer to SWE data.

The opposite, i.e. to register a SWE network in SenSer (Figure 10) is also feasible. The motivation is to compare local measurements with data retrieved from other locations. For instance, a concrete example is the comparison of air pollution indicators: how do the local measurements compare with the mean of the remainder of the city or country. In this scenario, the translator would have to be SensorNetwork compliant and convert SenSer requests to SWE and SWE data to SenSer. Figures 9 and 10 illustrate both processes.

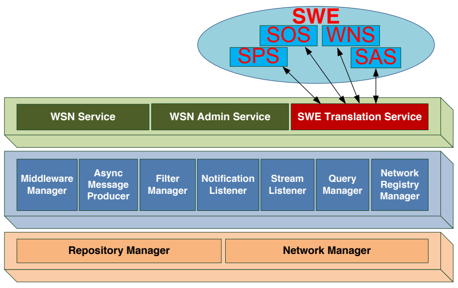

Fig. 9. Integrate SenSer in SWE

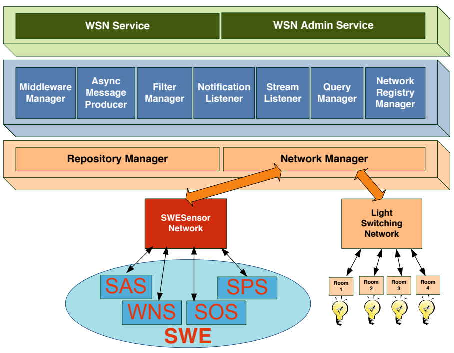

Fig. 10. Integrate SWE in SenSer

7 Related Work

IrisNet [8] stands out as one of the first projects to address sensor webs at an Internet-scale. It is a software infrastructure that supports distributed queries on a Internet-wide service-oriented platform that abstracts common sensing hardware, such as web-cams. Services are described through XML documents that are published on dedicated distributed databases. These operations, as well as the querying, are performed through a high-level API.

Sensor Web Enablement (SWE) [3] is a set of models and Web service interfaces proposed by the Open Geospatial Consortium for the Internet integration of sensor systems. The goal is to provide a specification for the integration of sensor networks in a Web of sensors accessible via Internet technologies. The focus is on the integration of several sensor networks in a web of sensors, rather than the actual remote access and management of such networks. The featured services have several limitations of which we emphasize: 1. the lack of management operations, such as network registry and network reprogramming; 2. the lack user levels or roles, such administration privileges, or different access permissions for different users; 3. the overall complexity when the purpose may be simply to setup a single network, and; 4. the disregard for community adopted Web service standards, such as service orchestration to perform planning, defining its own protocol. In our opinion, a sensor network should be virtualized as just another service on Web, and therefore easily integrable with standard Web service technologies.

Some SWE implementations are available[3] Among the most known are Nosa [4] and 52^o North [21]. Nosa is a research-oriented implementation of a first iteration of the specification, that had many limitations, some of which overcome with non-SWE extensions [12]. 52^o North is a more recent implementation with a more industrial focus. The documentation is mostly in German, thus not easily digestible for the majority of the scientific community.

GSN [1] also targets sensor network integration. The objecive is to build a sensor Internet by connecting virtual sensors, that abstract data-streams originating from either a sensor network or from another virtual sensor. SQL queries can than be performed on top of these virtual sensors. There is no focus on Web interoperability.

Aginome [9] integrates IP and sensor networks by resorting to mobile agents that communicate through tuple-spaces. The focus is on sensor network integration, agents may migrate between wireless sensor networks, rather than on Web exposure and interoperability.

8 Conclusions

We have presented SenSer, a framework that provides a generic middleware for the remote access and management of sensor networks. Two distinct interfaces

[3] `http://www.sensorsmag.com/networking-communications/government-military/new-impl ementations-ogc-sensor-web-enablement-standards-1437`

are defined to cleanly separate the operations available for regular users and administrators.

A major effort was placed on the support for modularity and heterogeneity. The framework's architecture follows a three-tier model, decoupling the presentation, logic and data layer, the latter comprising the two possible data sources: registered sensor networks and a history repository.

A Java-based prototype implementation exposes the framework's presentation layer as two Web services, providing a Internet-scale interoperable platform, thus solving client heterogeneity. Due to the lack of standards, sensor network heterogeneity is handled by a dedicated framework compliance interface, that specifies the SenSer/sensor network interaction protocol.

The prototype has been validated with the development of applications [20], of which one was briefly described in Section 4, and with an evaluation of the functional and non-functional requirements. Regarding the latter, a performance study revealed that SenSer is a lightweight framework that scales well with the increase of requests. Our final efforts were in the realization of two integration exercises, with Callas sensor networks and SWE sensor webs.

Regarding future work our goals include: 1. the addition of the registry of network types to the Java client API. This requires the migration of Java code on top of Web service technology, an ongoing work; 2. apply the filter pipeline functionality to modify client requested streams; 3. the definition of different access permissions, for instance to distinguish users that may only retrieve data, from the ones that can perform actions; 4. the integration with real-life applications, and; 5. the actual incorporation of an authentication module in the current implementation.

Acknowledgments. This work was partially funded by FCT MCTES under project CALLAS (contract PTDC/EIA/71462/2006) and the CITI research center.

References

1. Aberer, K., Hauswirth, M., Salehi, A.: A middleware for fast and flexible sensor network deployment. In: Dayal, U., Whang, K.-Y., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y.-K. (eds.) Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, pp. 1199–1202. ACM, New York (2006)
2. Baldus, H., Klabunde, K., Müsch, G.: Reliable set-up of medical body-sensor networks. In: Karl, H., Wolisz, A., Willig, A. (eds.) EWSN 2004. LNCS, vol. 2920, pp. 353–363. Springer, Heidelberg (2004)
3. Botts, M., Percivall, G., Reed, C., Davidson, J.: OGC Sensor Web Enablement: Overview and high level architecture. Technical report, OGC (2007)
4. Chu, X., Kobialka, T., Durnota, B., Buyya, R.: Open sensor web architecture: Core services. In: Proceedings of the 4th International Conference on Intelligent Sensing and Information Processing, pp. 98–103 (2006)
5. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: LCN 2004: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, pp. 455–462. IEEE Computer Society, Los Alamitos (2004)

6. Eswaran, A., Rowe, A., Rajkumar, R.: Nano-RK: An energy-aware resource-centric RTOS for sensor networks. In: RTSS 2005: Proceedings of the 26th IEEE International Real-Time Systems Symposium, pp. 256–265. IEEE Computer Society Press, Los Alamitos (2005)
7. Gay, D., Levis, P., von Robert Behren, Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 1–11. ACM, New York (2003)
8. Gibbons, P., Karp, B., Ke, Y., Nath, S., Seshan, S.: Irisnet: An architecture for a worldwide sensor web. IEEE Pervasive Computing 2(4), 22–33 (2003)
9. Hackmann, G., Fok, C.-L., Roman, G.-C., Lu, C.: Agimone: Middleware support for seamless integration of sensor and IP networks. In: Gibbons, P.B., Abdelzaher, T.F., Aspnes, J., Rao, R. (eds.) DCOSS 2006. LNCS, vol. 4026, pp. 101–118. Springer, Heidelberg (2006)
10. Hillenbrand, M., Verney, A., Muller, P., Koenig, T.: Web services for sensor node access. In: 5th CaberNet Plenary Workshop (2003)
11. Jiang, G., Chung, W., Cybenko, G.: Semantic agent technologies for tactical sensor networks. In: Carapezza, E.M. (ed.) Proceedings of the SPIE Conference on Unattended Ground Sensor Technologies and Applications V, vol. 5090, pp. 311–320. SPIE, San Jose (2003)
12. Kobialka, T., Buyya, R., Leckie, C., Kotagiri, R.: A sensor web middleware with stateful services for heterogeneous sensor networks. In: 3rd International Conference on Intelligent Sensors, Sensor Networks and Information, ISSNIP 2007, pp. 491–496 (2007)
13. Kumar, C.-C.H.R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: MobiSys 2005: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, pp. 163–176. ACM, New York (2005)
14. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: TinyOS: An operating system for sensor networks. Ambient Intelligence, 115–148 (2005)
15. Lewis, P.: The TinyScript Language. UC Berkeley (2004)
16. Lifton, J., Seetharam, D., Broxton, M., Paradiso, J.A.: Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks. In: Mattern, F., Naghshineh, M. (eds.) PERVASIVE 2002. LNCS, vol. 2414, pp. 139–151. Springer, Heidelberg (2002)
17. Martins, F., Lopes, L., Barros, J.: Towards Safe Programming of Wireless Sensor Networks. Electronic Proceedings in Theoretical Computer Science 17, 49–62 (2010)
18. OASIS. Web Services Business Process Execution Language (WSBPEL) TC, `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`
19. OASIS. Web Services Notification (WSN) TC, `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn`
20. Santos, J.R.: Um middleware para acesso e gestão de redes de sensores em ambientes web. Master's thesis, Faculdade de Ciências - Universidade Nova de Lisboa, Supervised by Hervé Paulino (2009)
21. Stasch, C., Walkowski, A., Jirka, S.: A geosensor network architecture for disaster management based on open standards. In: Digital Earth Summit on Geoinformatics 2008: Tools for Climate Change Research, pp. 54–59 (2008)