A Service-Based Universal Application Interface for Ad-hoc Wireless Sensor Networks

Marco Sgroi†, Adam Wolisz‡, Alberto Sangiovanni-Vincentelli† and Jan M. Rabaey†

University of California at Berkeley†, Technical University of Berlin‡

Abstract

This paper addresses the fundamental issue of defining a standard set of services and interface primitives which should be made available to an application programmer independently on their implementation on any present and future sensor network platform. As the definition of sockets has made the use of communication services in the Internet independent of the underlying protocol stack, communication medium and even operating system, the application interface we propose identifies an abstraction that is offered to any sensor network application and supported by any sensor network platform. The distributed service platform we introduce builds on the query/command paradigm already used in several sensor network implementations and includes time synchronization, location and naming services that support the communication and coordination among application components.

1. Introduction

Ad-hoc wireless sensor networks have the potential to become a major component of the Information Technology space. They can dramatically change the operational models of traditional businesses in several application domains, such as the building industry [1], power delivery [2], and environmental control [3]. Sensor networks are already the essential backbone of the "ambient intelligence" paradigm, which envisions smart environments aiding humans to perform their daily tasks in a non-intrusive way [4].

This evolution has not escaped the attention of both academia and industry and has led to a flurry of activities such as the exploration of new applications and the development of new radio architectures, low-power wireless sensor nodes, low-date rate wireless protocols, and ad-hoc multi-hop routing algorithms. It has been observed that the creation of some sense of interoperability between the myriad of hardware components and software protocols is essential for the full potential of these technologies to be achieved. In that light, a number of new wireless standards such as 802.15.4 [5] and Zigbee [6] are under development. Yet, it is our belief that these efforts created in a bottom-up fashion do not fully address the essential question of how to allow the interoperability between the multitudes of sensor network operational models that are bound to emerge. In fact, different operational scenarios lead to different requirements in terms of data throughput and latency, quality-of-service, use of computation and communication resources, and network heterogeneity. These requirements ultimately result in different solutions in network topology, networking protocols, computational platforms, and air interfaces.

To support true interoperability between different applications as well as between different implementation platforms and ensure scalability of the sensor network technology in future years, we advocate a top-down approach similar to the one adopted very successfully by the Internet community. We propose a universal application interface, which allows programmers to develop applications without having to know unnecessary details of the underlying communication platform, such as air interface and network topology.

This paper addresses a fundamental issue:

Define ***a standard set of services and interface primitives (called the Sensor Network Services Platform or SNSP)*** *to be made available to an application programmer independently on their implementation on any present and future sensor network platform.*

As the definition of sockets in the Internet has made the use of communication services independent of the underlying protocol stack, communication medium and even operating system, the application interface we propose identifies an abstraction that is offered to any sensor network application and supported by any sensor network platform.

Yet, while similar in concept, the application interface needed for sensor networks is fundamentally different from the one defined in the Internet space. In the latter, a seamless set-up, use, and removal of reliable end-to-end communication links between applications at remote locations are the primary concern. To understand the needs of wireless sensor networks, consider the operation of the network, as seen from the application perspective. The main purpose of sensor networks is to *capture information from distributed sources, and then report it where requested (monitoring applications) or use it to actively influence the environment (control applications).* This is accomplished by deploying sensor networks with (a large number of) nodes that are able to capture different physical phenomena (sensors), to make control decisions (controllers), and to act on the environment (actuators). The nodes of these networks fulfill their data gathering and control functionality working in a cooperative way; hence, inter-node communication (mostly over RF links) plays a key role. The requirements are such that nodes must be deployed wherever the applications require and must operate for long periods without human intervention. This creates significant design challenges, among which low energy consumption is of paramount importance. To meet the stringent power, cost, size and reliability requirements, the design of sensor networks needs optimizations throughout all the steps of the design process, including the application software, the layers of the communication protocol stack and the hardware platform. Thus, it is not surprising that most solutions that have recently emerged tend to have a monolithical structure rather than following the "layered" approach used in traditional networking applications. *This "unstructured" approach, while potentially efficient, is fraught with several major problems and disadvantages. Today, it is virtually impossible to start developing applications without having selected a specific, integrated hardware/software platform first.* Coupling applications with specific hardware solutions slows down the practical deployment of sensor networks: potential users hesitate to invest in developing applications that are intrinsically bound to specific hardware platforms available today, without having any perspective on the reusability of their investments on future platforms. In addition, these "stovepipe solutions" tend to hinder the overlay of multiple co-existing applications on a single hardware platform. *The potential overlay of applications (for example, environment control, and security in a building scenario) is one of the properties that make the wide deployment of sensor networks plausible and attractive.*

The first and most obvious application need when implemented with a sensor network is the availability of well-defined communication services between the three essential components of the network (sensor, controller, and actuator). In addition, an operational network requires the presence of services, such as naming, locationing, and time synchronization. Finally, as is the case for any network deployed over long periods, services should be available that help to evaluate, manage and diagnose the operation of the network. All these services are made available to the application programmer through a generic, well-defined *Application Interface (AI).*

Our goal in this paper is to define the services and the primitives needed to operate sensor networks, and that ***we are not concerned with how and where these services may be implemented***, as illustrated in Fig. 1.

Some services may be provided by a dedicated service layer, acting as a middleware, on top of the communication primitives offered by the protocol stack. Alternatively, the same services might be implemented by smart processing functions (e.g. aggregation, hop count) embedded in the communication protocol stack.

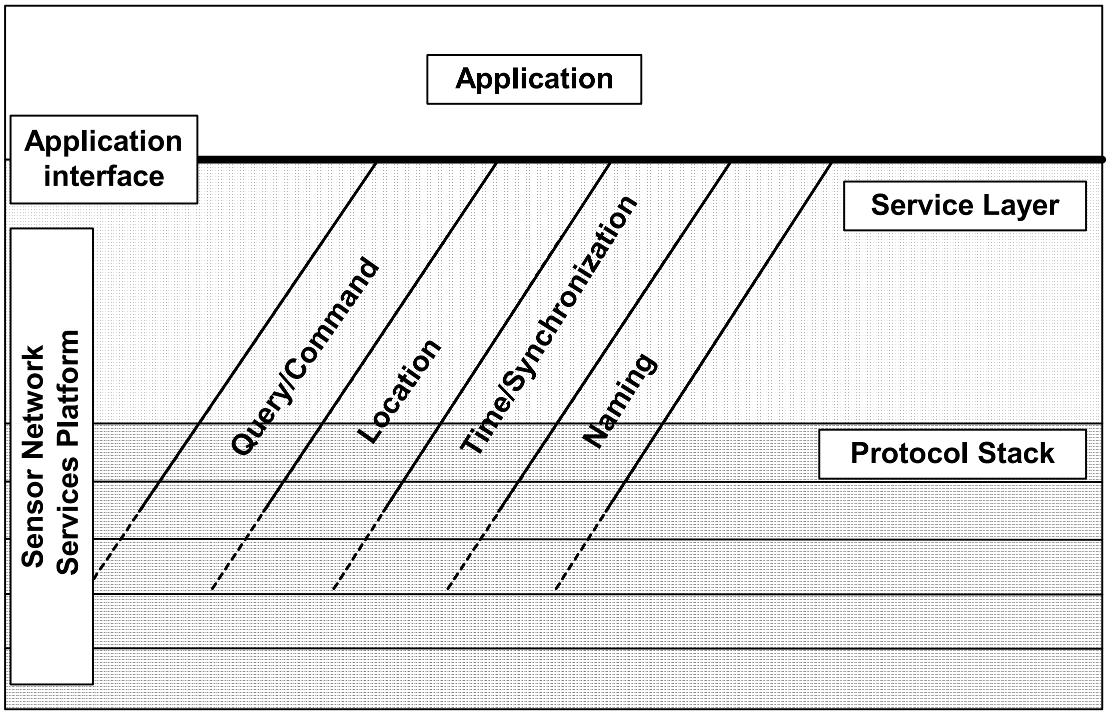

Figure 1: A Service-oriented Application Interface

The paper is structured as follows. After an overview of existing efforts in this field, we introduce the functional components of a wireless sensor network and create a framework in which we can define the services offered by the distributed Service Platform. A clear distinction is made between the logical functions and the physical components that are used to implement them. Next, we define the primitives that allow gathering (Query) of information as well as acting on the environment (Command) and auxiliary services such as Concept Repository, Location and Time Synchronization. Then, the interoperation between sensor networks and between sensor networks and global networks is defined. At the end the paper outlines how the defined abstractions can be mapped onto existing platforms such as PicoRadio, and TinyDB.

We reiterate that the main goal of this white paper is to initiate a broader discussion on a universal APIs for sensor networks, and that it by no means intends to be the defining and final word. The authors hence invite and welcome any constructive feedback that may help to accomplish the aforementioned goals.

2. Related Work

While we believe that this white paper is the first to attempt the definition of a universal service-based application interface for wireless ad-hoc sensor networks, this does not mean that no other efforts have already been made towards defining higher-level abstractions and interfaces for sensor networks. A number of standards for open communication in sensor networks have been proposed. The most known are the BACnet and LonWorks standards, developed for building automation [1]. These standards are geared towards well-defined application areas, and are built on top of fairly well defined network structures. Hence, many of the exciting new developments that currently are emerging from the wireless sensor network community cannot be accommodated within these frameworks. At the same time, the application-specific functionality of both BACnet and LonWorks can easily be overlaid on top of the services-based model proposed in this paper.

Networks of sensors (mostly wired) have further been used in automation and manufacturing. A number of standards have been developed within the IEEE to deal with different manufacturers. More specifically, the IEEE 1451.2 [7] standardizes both the key sensors (and actuators) parameters and their interface with the units that read their measures (or set their values). In particular, the standard defines the physical interface between the Smart Transducer Interface Module (STIM), which includes one or more transducers, and the Transducer Electronic Data Sheet (TEDS) containing the list of their relevant parameters, and the Network Capable Application Processor (NCAP), which controls the access to the STIM.

The standard defines the following interface between the STIM and the NCAP:

- the NCAP sends a *trigger* to command the STIM to take an action (e.g. make a measure). A trigger acknowledgment is sent by the STIM when the action has taken place
- for data *read* (and write for actuators) the NCAP sends the STIM an *address* that specifies whether data has to be read (or written), from which sensor, and which type of data, the STIM responds with the requested data
- an interrupt is used to indicate exceptional conditions in the STIM that require service from the NCAP

The standard identifies seven types of transducers: sensors, actuators, buffered sensors, data sequence sensors, buffered data sequence sensor and event sequence sensors. Event sequence sensors signal only the time of occurrence of an event by sending a trigger acknowledgment. The other types of sensors differ for whether they sample after a trigger or at their own rate and which sample (last or previously buffered sample) they provide upon request.

The standard lists the mandatory and optional parameters in a TEDS and distinguishes between those common to all the transducers in a STIM (Meta TEDS) and those related to individual transducers (Channel TEDS). Additional optional parameters concern calibration and IDs. Given the fact this standard is wide spread and industry-tested, we ensure that the services we propose and the parameters of the components comply with the IEEE 1451.2 standard. The IEEE 1541.2 standard defines a generic interface and a broad set of parameters for applications using transducers. Our scope includes only wireless sensor network applications. Hence, we take a subset of the IEEE standard and extend it where necessary. For the interface between the components reading the sensors and the sensors themselves we maintain the same protocol and primitives defined by [7] for the interface between STIM and NCAP. Then, we consider only the parameters defined in [7] that are relevant for wireless sensor network applications and add others that we believe are relevant in this domain.

The approach that is closest to our goals is TinyDB [8]. While TinyDB is also based on the Query/Command paradigm advocated here, its main goal is to define the interface and an implementation of a specific service, the query service, rather than defining a universal service platform interface. Hence, TinyDB does not include several auxiliary services necessary in many sensor network applications. A more detailed description of TinyDB will be given in Section 6.2.

Finally, it is worth mentioning that a number of standards are in the making at the ad-hoc wireless network layer. The best known is Zigbee, advocated by a consortium of companies [6]. ZigBee defines an open standard for low-power wireless networking of monitoring and control devices. It works in cooperation with the IEEE 802.15.4 standard, which focuses on the lower protocol layers (physical and MAC). Instead, ZigBee defines the upper layers of the protocol stack, from network to application, including application profiles. From our perspective, Zigbee represents only one possible way to realize a network. The services proposed here can be easily deployed in and on top of a Zigbee realization or alternative implementations such as Bluetooth Scatternets.

The research community has proposed several implementations for each of the services offered by the Sensor Network Service Platform.

Several attribute-based schemes that assign, modify, or translate names have already been proposed [9, 10, 11]. These schemes are different from the one used in the Internet, where nodes are usually addressed individually and are identified by a unique identifier called IP address. DNS (Domain Name Server [11]) holds an association between names and IP addresses and upon request provides a source node with the IP address of the destination host before message delivery (early binding). INS [9] proposes an overlay network of Intentional Naming Resolvers (INRs) that associates attribute-based names with IP addresses and binds them at message delivery time (late binding) rather than at request resolution time. In INS, naming is done using name-specifiers based on a set of attributes and their values. Attribute-based naming is also proposed in [10] where name matching, instead of being done by special-purpose network elements, is distributed across the network and names are associated with (attribute, value, operation) tuples, where operation specifies the type of operation ($=, <, >, \leq, \geq \ldots$) to be used for name matching.

While the time synchronization services are well understood in the Internet world [12], location services are a rather new concept and are closely related to the concept of mobility. Only recently cellular networks have started to provide location information to mobile devices based on the interaction with base stations. Of course, the best known and most wide-spread location service is the ubiquitously used Global Positioning System (GPS).

3. Anatomy of Ad-hoc Wireless Sensor Networks

Ad-hoc Wireless Sensor Networks (AWSNs) are used for *monitoring, analysis,* and *control* of the environment. Monitoring applications gather the values of some parameters of the environment, process them, and report the outcome to external users. Control applications go one step further: they gather the values of parameters of the environment and use them to influence the environment so that a required behavior is obtained.

3.1 Applications of Ad-hoc Wireless Sensor Networks

*An **AWSN Application** consists of a single algorithm or a collection of cooperating algorithms designed to achieve a common goal, aided by interactions with the Environment through distributed measurements and actuations.*

In light of the dominant activity of sensor networks, we name each algorithm that forms an AWSN a controller.

***Controllers** are components of AWSNs that read the state of the environment, process the information and report it or apply a control law to decide how to set the state of the environment.*

A controller is characterized by the following elements:

- *desired behavior* of the system (e.g. average temperature higher than a minimum value, or desired representation of the processed data)
- *input variables* that can be read (e.g. temperature, position)
- *output variables* that can be set (e.g. blinds open or close) or reported (e.g. temperature gradient)
- control *algorithm* that is an algorithm that attempts to accomplish the desired behavior through a combination of observation and actuation (e.g. threshold-based decision)
- *model of the environment* (e.g. how certain actuation actions such as closing the blinds affect the temperature in the room). This model may also contain information on the topological structure of the environment.

In addition, to ensure proper operation a controller places some constraints on parameters expressing the quality of the input data such as:

- timeliness
- accuracy
- reliability

Generally, a controller operates in full autonomy, and its behavior is observable only by looking at the outputs (control settings or reports). A controller can, however, make some data externally visible or modifiable by an explicit export of parameters.

3.2 The Sensor Network Services Platform (SNSP)

To perform its functionality, a controller (algorithm) has to be able to read and set the state of the environment. In an AWSN, controllers do so by relying on communication and coordination among a set of distributed functions (control, sensing, and actuation). We propose that the Sensor Network Services Platform (SNSP) provide these communication and coordination functions.

The ***Sensor Network Services Platform*** *decomposes and refines the interaction among controllers and between controllers and the Environment into a set of interactions between control, sensor, and actuation functions.*

Hence, the services that the SNSP offers the Application are used directly by the controllers whenever they interact among each other or with the Environment. This approach abstracts away the details of the communication mechanisms, and allows the Application to be agnostic on how exactly the interaction with the environment is accomplished, as is illustrated in Figure 2.

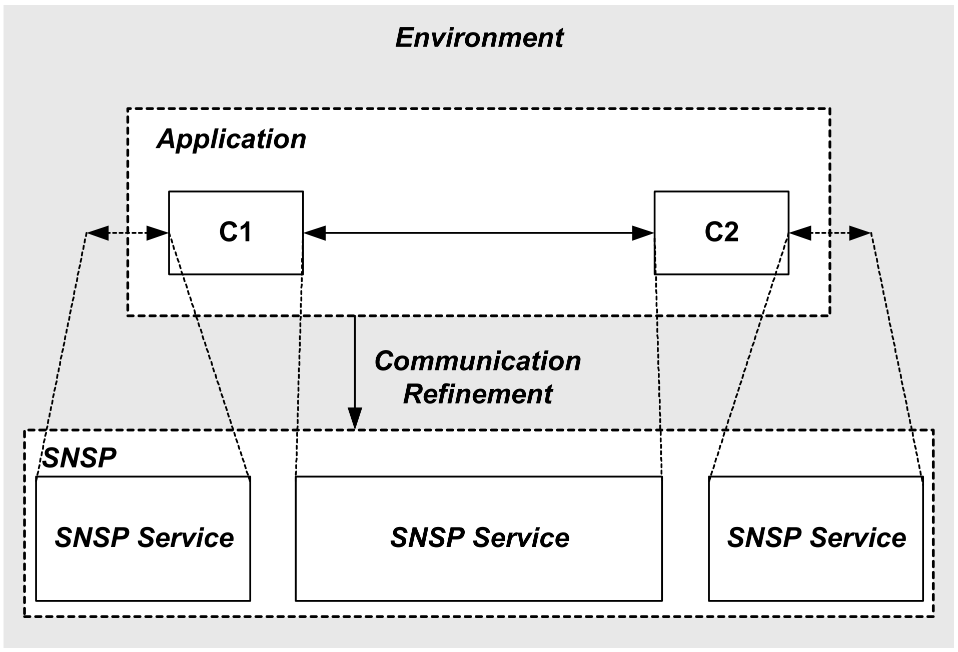

Figure 2: The SNSP refines interactions between controllers and the environment into communications between control, sensor, and actuation functions.

More in detail, the SNSP (as illustrated in Figure 3) is a collection of:

- algorithms (e.g. location and synchronization)
- communication protocols (e.g. routing, MAC)
- data processing functions (e.g. aggregation)
- I/O functions (sensor, actuation)

that cooperate and provide at least the following services (as described in Section 4):

- **Query Service (QS)** used by controllers to get information from other components
- **Command Service (CS)** used by controllers to set the state of other components
- **Timing/Synchronization Service (TSS)** used by components to agree on a common time
- **Location Service (LS)** used by components to learn their location
- **Concept Repository Service (CRS)** used to maintain a common definition of the concepts that are agreed by all the components in the system during the network operation. It also presents a repository of the capabilities of the deployed system.

The latter service is quite novel in the AWSN community, but is deemed essential if a true ad-hoc realization of the network is to be obtained. The repository includes definitions of relevant global concepts such as the attributes that can be queried (e.g. temperature, pressure), or of the regions that define the scope of the names used for addressing. It further allows to collect information about the capabilities of the system (i.e. which services it provides and at which quality and cost) and provide the Application with a sufficiently accurate description. The repository (Section 4.4) is dynamically updated during network operation.

Access to the SNSP services is provided to the Application through a set of primitives, combined in the **Application Interface (AI).** The AI primitives can also make available to the Application the relevant parameters that define the quality and the cost of the services.

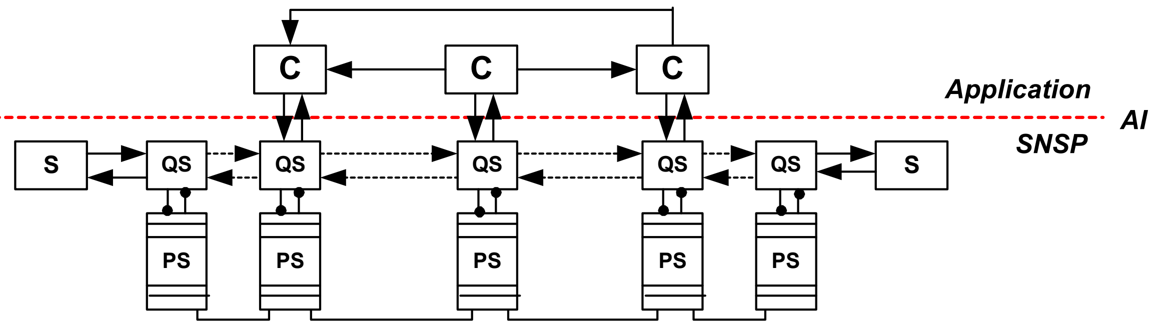

Figure 3: Refinement of an (Monitoring) Application into a collection of controllers (C), sensors (S), query services (QS) and protocol stacks (PS)

The goal of this paper is to formalize the primitives that allow the Application to access the services of the SNSP, and not to define the architecture of the SNSP itself. However, it is worth to mention that, within the SNSP, components implementing different services frequently interact and use each other's services. For example, Query Service components may invoke the Time Synchronization Service to activate the synchronization of the nodes involved in a query, or the Localization Service to aid in the data routing process.

3.3 The Sensor Network Implementation Platform

While the SNSP description suffices to capture the interaction between controllers, sensors and actuators, still it is a purely functional description, which does not prescribe how and where each of these functions will be implemented. Hence, information such as energy, delay, cost, and memory size, cannot yet be provided. This is only possible once the functional components of the SNSP have been mapped onto actual hardware nodes. A description of the actual hardware platform is given by the ***Sensor Network Implementation Platform (SNIP).***

The ***Sensor Network Implementation Platform*** *is a network of interconnected physical nodes that implement the logical functions of the Application and the SNSP.*

A ***physical node*** is a collection of physical resources such as
- Clocks and energy sources
- Processing units, memory, communication, and I/O devices.
- Sensor and actuator devices

The main physical parameters of a node are:

- list of sensors and actuators attached to node
- memory available for the application
- clock frequency range
- clock accuracy and stability
- level of available energy
- cost of computation (energy)
- cost of communication (energy)
- transmission rate (range)

Choosing the architecture of the SNIP and the mapping of the functional specification of the system properly is a critical step in sensor network design. The frequency of the processor, the amount of memory, the communication link parameters (e.g. bandwidth), and other similar parameters of the SNIP ultimately determine the capabilities of the network, i.e. **the quality and the cost of the services** it offers. The quality and the cost of the services are expressed by parameters such as the delay in obtaining the desired information, the lifetime of the system, and the reliability of the data, all of which are made accessible to the application through the Application Interface.

Figure 4 visualizes the concept of mapping Application and the SNSP functions onto the SNIP. Here, *QS* stands for query service, *S* sensor, *C* controller, and *N* node. The shaded boxes and the dotted arrows associate groups of logical components with physical nodes.

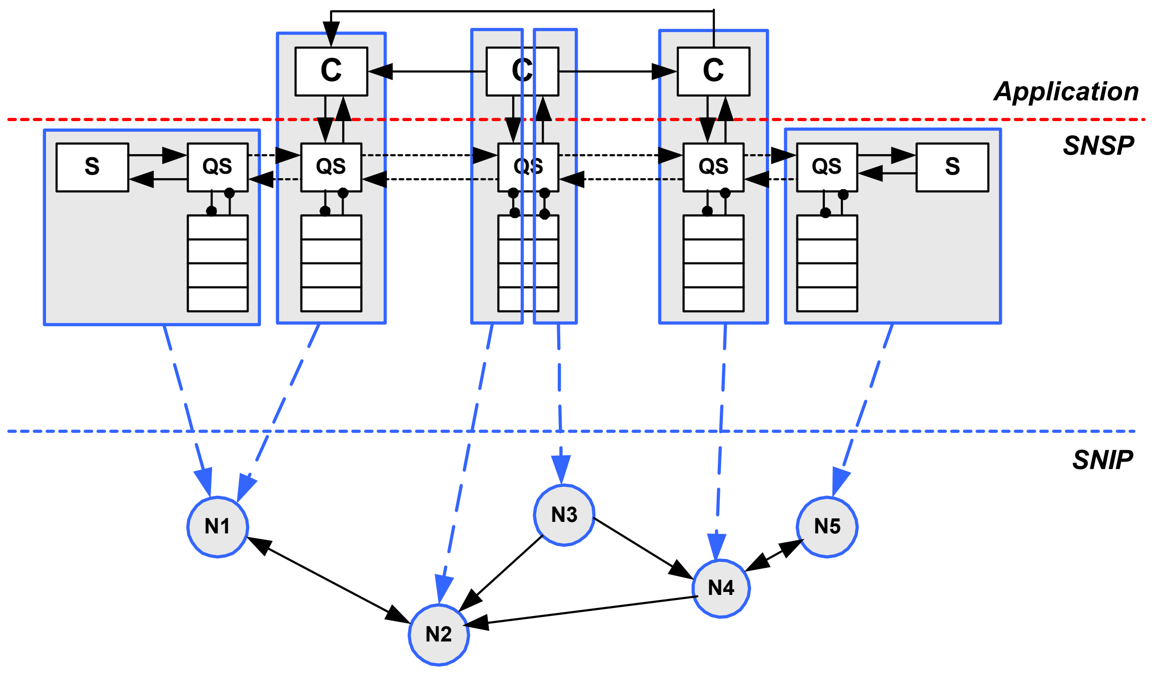

Figure 4: Mapping Application and SNSP onto a SNIP

An ***instantiated node*** binds a set of logical Application or SNSP functions to a physical node.

The instantiation is the direct result of the mapping operation. Observe that certain physical parameters such as location and time can only be defined after mapping. All functions mapped to the same physical node then share the values of these parameters. For example, a temperature and a pressure sensor attached to the same physical node inherit their location from that node. Similarly, their definition of time directly relates to the accuracy and the resolution of the time reference (clock) of the node.

The operation of the Application or the SNSP may depend upon the value of a physical parameter of the SNIP (for instance, amount of energy level, quality of the channel, etc). To make this information available in a transparent fashion, **an additional service called the *Resource Management Service (RMS)*** is provided, which allows an Application or the SNSP to get or set the state of the physical elements of the SNIP.
Typically, the RMS can access the following physical parameters (this list is not intended to be complete — some network implementations might require other parameters to be exposed):

- the energy source of a node. This can be of three types: unlimited power, battery, scavenging
- the energy level of a node
- power management state (can be read or set)
- the quality of the radio channel (e.g. can be measured by the rate of CRC failures, or by the number of retransmissions in a session)
- the energy cost of certain operations and their execution time
- the transmission power of the radio and its range (this parameter can also be set)

3.4 Sensors and Actuators

To interact with the environment the SNSP supports sensor and actuation functions. Observe that these are pure functional modules. Certain parameters of a sensor or actuation function can only be set after the function is mapped to a physical node in the SNIP. Examples of such are the energy dissipation per sensing operation, best resolution, accuracy, etc. In determining the parameter lists for sensors and actuators, as given below, we have attempted to stay compatible with the IEEE 1451.2 standard to the maximum extent. Other parameters such as energy cost were added as they are especially significant in the AWSN domain of applications.

3.4.1 Sensors

*A **sensor** is a component that measures the state of the Environment.*

A *logical sensor* is a functional component of the SNSP that measures a parameter of the environment and provides the measure when requested. Abstracting away the physical interaction with the Environment, sensors can be modeled as *data sources with no input signals carrying data*.
A *physical sensor*, on the other hand, represents an hardware component with well-defined characteristics in terms of accuracy, stability, range, etc.

We identify two types of sensors:

1. *Simple sensors:* devices that directly map a physical quantity into an electrical signal and provide the measure upon request. Devices that measure physical parameters, such as temperature, sound, and light, as well as input devices such as keyboards or microphones that allow external users to enter data or set parameters are examples of simple sensors.
2. *Virtual sensors:* components that overall look like sensors in the sense that they provide data upon an external request. Virtual sensors are defined by the list of parameters that can be read and by the primitives that are used for reading them. Examples of virtual sensors are
   - A sensor that provides an indirect measure of a certain environment condition by combining one or more sensing functions with processing (e.g., transformation, compression, aggregation).

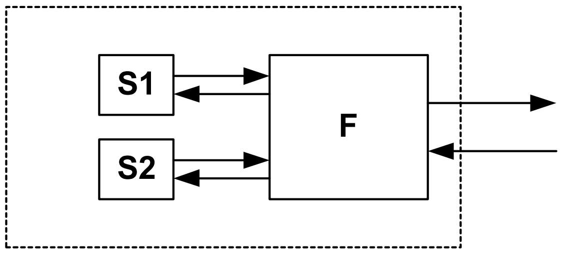

Figure 5: An Example of a Virtual Sensor

   - A controller when it is queried for the value of one of its parameters
   - An external network that provides requested data through a gateway (see Section 5).

The properties and the capabilities of a sensor are enumerated through a list of parameters. While some parameters are shared between the logical and physical incarnations of a sensor, a subtle but important difference between their meanings should be pointed out. A parameter corresponding to a logical sensor expresses *a constraint*; for instance, the accuracy and the dynamic range needed by the application. On the other hand, a parameter corresponding to a physical sensor expresses *the capability* of the deployed hardware; to use the same example, the accuracy and the dynamic range parameters now express the quality of the available hardware. *A successful mapping requires that the capabilities of the physical node meet the constraints imposed by the logical function*. Some parameters only make sense at the physical level and have been identified as such.

1. parameter being measured and units (*)[1].
2. range (min and max measured values) and resolution (*)
3. expected accuracy of the measure (maximum difference from the real value) (+)
4. sampling rate (+)
5. bandwidth (max speed of variation of the physical parameter that can be observed) (+)
6. cost in time and energy per sample (+)
7. name of the manufacturer (Physical Only) (*)
8. ID of the sensor (Physical Only) (*)
9. date of certification/calibration (Physical Only) (*)
10. number of past measures that are available (sensor buffer size) (Physical Only) (*)
11. maximum data rate supported by the interface (Physical Only) (*)

3.4.2. Actuators

*An **actuator** is a component that sets the state of the environment.*

A *logical actuator* is a functional component of the SNSP that sets a parameter of the environment. Abstracting away the physical interaction with the Environment, actuators can be modeled as *data sinks with no output signals carrying data.* A *physical actuator*, on the other hand, represents an actual hardware component with will-defined characteristics in terms of settings, accuracy, etc.

Again, we identify two types of actuators:

1. *Simple actuators* that are devices that map an electrical signal into a physical quantity and may return an acknowledgment when the action is taken. Examples of actuators are devices that modify physical parameters, such as heaters and automatic window/door openers, as well as output devices, such as displays and speakers.
2. *Virtual actuators* that overall look like actuators in the sense that they receive values to set some parameters. Virtual actuators are defined by the list of parameters that can be set and by the primitives that are used for setting. Examples of virtual actuators are:
   - An actuator that provides an indirect way of controlling a certain environment condition by combining one or more physical actuators with processing (e.g., transformation, and decompression).
   - controllers whose parameters are set by other controllers
   - networks that receive commands to take actions through gateways (See Section 5).

[1] (*) is used to indicate a parameter defined in the IEEE 1451.2 standard, (+) an additional parameter

The most relevant parameters of the actuators are (all the arguments raised with respect to sensor parameters pertain here as well):

1. parameter whose value is set (*)
2. range (min and max values detected) and the resolution (*)
3. expected accuracy (maximum difference from the real value) (+)
4. bandwidth (max speed of variation of the physical parameter that can be achieved) (+)
5. sampling rate (+)
6. cost (in time and energy) per operation (+)
7. name of the manufacturer (Physical Only) (*)
8. ID of the actuator (Physical Only) (*)
9. date of certification (Physical Only) (*)
10. number of commands that can be accumulated (Physical Only) (*)

4. SNSP Services and Application Interface

This section describes the services offered by the SNSP and defines their primitives. The execution of these services and their invocation by the application requires that the communicating components are able to identify each other and agree on a common naming scheme. Hence, before describing the individual services, we first present the naming scheme that they follow.

4.1 Naming

In sensor network application, components communicate because of their features, often without knowing a priori which and how many components share those specified features. For example, a controller may want to send a message to all the sensors measuring temperature in a region, say in the kitchen. In this case, the group of temperature sensors located in the kitchen may be named and addressed for message delivery using the attributes “temperature” and “kitchen” rather than using the list of IDs of all the individual sensors having those attributes.

*A **name** is an attribute specification and scope pair.*

*An **attribute specification** is a tuple $((a_1,s_1), (a_2,s_2),\dots (a_n,s_n), expr_1, expr_2,\dots expr_l)$,* where a_i is an attribute; s_i is a selector that identifies a range of values in the domain of a_i; and $expr_k$ is a logical expression defined by attribute-selector pairs and logical operators.

For example, a name can be defined by the pairs (temperature, ≥ 30 ºC), (humidity, ≥ 70 % R.H.), (sound, < 20 dB), and the expression ((temperature, ≥ 30 ºC) OR (humidity, ≥ 70 % R.H.)) AND (sound, < 20 dB). Examples of attributes commonly used for naming in sensor networks are the physical parameter being measured by sensors or modified by actuators (e.g. temperature, humidity), or parameters of the instantiated nodes such as location or energy level.

Attribute specifications are always understood within a scope.

*A **scope** is a tuple $(O_1, O_2, \dots O_n, R_1, R_2,\dots R_m)$, where O_i is an organization unit and R_j is a region.*

*A **region** defines a sub-space, i.e., a set of locations.*

We differentiate between two types of regions:

- a ***zone*** is a set of locations identified by a common name; e.g. the kitchen or the SF Bay
- a ***neighborhood*** represents a set of locations identified by their closeness to a reference point; e.g. all nodes within a radius of 10 m from a given location

*An **organization** is an entity that owns or operates a group of nodes.*

Organizations are essential to differentiate between nodes that operate in the same or overlapping regions, yet belong to different organizations (for instance, the police and the fire department).

The attributes and the types of selectors, which may be used for naming, and the organizations and the regions, which might be used to define a scope, are listed in the *Concept Repository Service* (Section 4.4).

In general, names are and must not be unique. In addition, names may change during the evolution of the network because of the movement of nodes or the modification of some attributes. A Node ID is just a special case of an attribute that is unique in a given scope. For example, IEEE MAC Addresses are IDs with worldwide scope of uniqueness. However, in virtually all cases of AWSN operation the use of unique node IDs is not required.

4.2 Query Service (QS)

*The **Query Service (QS)** allows a controller to obtain the state of a group of components.*

A *query* is a sequence of actions initiated by a controller (*query initiator)*, which requests specific information from a group of sensors *(query targets).* If the requested information is not available, QS always returns a negative response within a maximum time interval, which has been set previously. Figure 6 visualizes the interactions of QS with the query initiator and the query target. In sensor network applications, query targets are typically sensors that provide controllers with the requested measures but in some applications the target may be a group of controllers that are requested their current state.

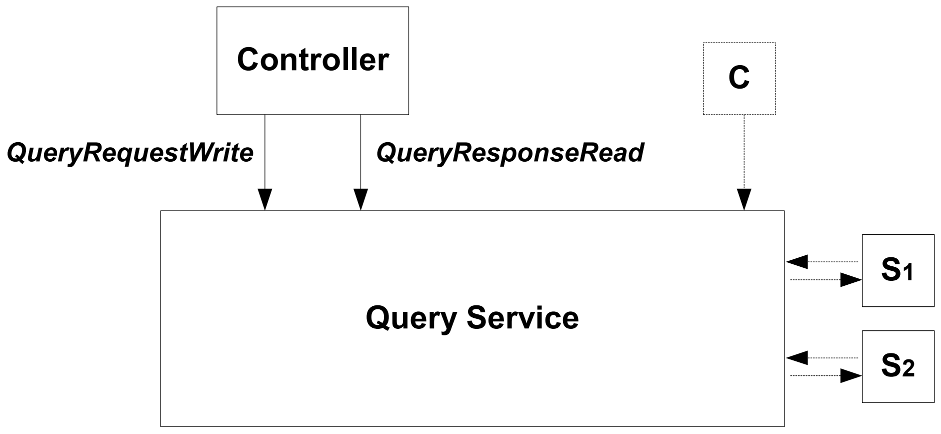

Figure 6: Query Service interactions

Queried parameters. In addition to the physical data being measured by a sensor, a controller may query parameters related to the node containing the sensor, such as security (if the measure comes from a trusted source), time (when the measure was taken), location (where the measure was taken), accuracy (how accurate was the measure). The list of attributes that are meaningful in a given sensor network are available from the *Concept Repository Service*. If no parameter is indicated in a query request, by default the response returns the data measured by the target sensors.

Query class. *A Query Class defines the context of a query by defining and constraining the response scope* (*any parameter that is not defined maintains the default value):*

- *Accuracy and resolution* of the sensor measures. Accuracy is the maximum error over time. Resolution is defined by the least significant unit that can be detected.
- *Timeliness.* It expresses how recent the measure of the queried parameter must be and is given by the maximum time allowed for a measure to be valid.
- *Maximum latency*. The maximum time interval between the beginning and the end of a query, i.e. between when the request is issued and all the expected query responses reach the destination.
- *Tagging requirements*. It indicates if the query response should include time and location tags.
- *Priority.* It is an integer indicating the relative priority that the QS should follow to process a query with respect to other queries.
- *Quantifiers* such as *all*, *at least k*, *exactly k*, *any*. They indicate how many nodes in the group are addressed (i.e. how many measures are queried), if the group contains more than one node.
- *Operations* such as *max*, *min*, *average*. They indicate the type of operation (transparent to the application and performed within QS) on multiple measures.
- *Security.* It indicates if the data must be secure or not.

All the query instances belonging to a certain class must follow the parameters of the class.

QueryID. Multiple queries, coming from the same or different controllers, can occur concurrently. QS uses a QueryID number to relate a query request with the corresponding responses. An arbitrary integer chosen by the QS (for example the *timestamp* indicating the time when the query request is sent) can be used as QueryID. The QueryID assigned to a query is always released as soon as the corresponding query terminates.

Response Type. In a query, the controller specifies the frequency of the responses it expects from the sensors. Three types of response patterns are especially relevant (other patterns, e.g. average or peak rates in bursty responses, can be optionally defined):

- *one-time* response
- *periodic* responses with interval period *p*
- *notification* of events whenever an event specified by an event condition occurs (event conditions are included in the *Concept Repository*)

Reliability. A query is *reliable* if the query initiator is guaranteed to receive *at least one* response, which has not been corrupted. In all other cases, the query is said to be unreliable. The default case is the unreliable query. If a reliable query is requested this has to be explicitly specified. The support for reliable queries is provided, within the query service, by means of specific reliability assuring mechanisms.

Primitives. Controllers use the following primitives to access QS:

- ***QSClassSetup*** creates a QueryClass, configuring one up to all the following parameters: accuracy, resolution, timeliness, security, max_response_time, location and time tags, priority, quantifier, operations.

*int QSClassSetup (struct accuracy *a, struct resolution *r, int timeliness, int security, int max_time, int loc_tag, int time_tag, int priority, int quantifier, int operation)*
returns the descriptor of the query class or ERROR

- ***QSClassUpdate*** updates one up to all the set of parameters of a query class previously defined

*int QSClassUpdate (int QueryClass, struct accuracy *a, struct resolution *r, int timeliness, int security, int max_time, int loc_tag, int time_tag, int priority, int quantifier, int operation)*
returns OK if the update is initiated or ERROR

- ***QSRequestWrite*** initiates a query of the type indicated in QueryClass to obtain a parameter from the components specified in *name_target

*int QSRequestWrite (struct name *name_target, char* parameter, int QueryClass, int ResponseType, int reliability)*
returns a QueryID as a descriptor of the query or ERROR

- ***QSResponseRead*** provides the parameter of the query when it is available for reading

int QSResponseRead (int QueryID)
returns the value of the requested parameter, if available, associated with the Query identified by QueryID. If it is not available, it returns a special value indicating the response has not arrived yet.

- ***QSStopQuery*** is used by a query initiator to stop a query and release the QueryID for use in other queries.

int QSStopQuery (int QueryID)
returns OK or ERROR

QS operation. Figure 7 plots a sequence of primitive function calls associated with a query. The QS execution follows the client-server model. First, the query class parameters are initialized using *QSClassSetup*. Then, the controller calls the *QSRequestWrite* function to initiate individual queries. QS returns the query descriptor (QueryID) to the controller which is blocked waiting for an immediate answer on whether the query can be initiated. If the answer is negative, an error message is returned. If the query is successfully initiated, QS begins the procedure of getting the parameter requested by the application.

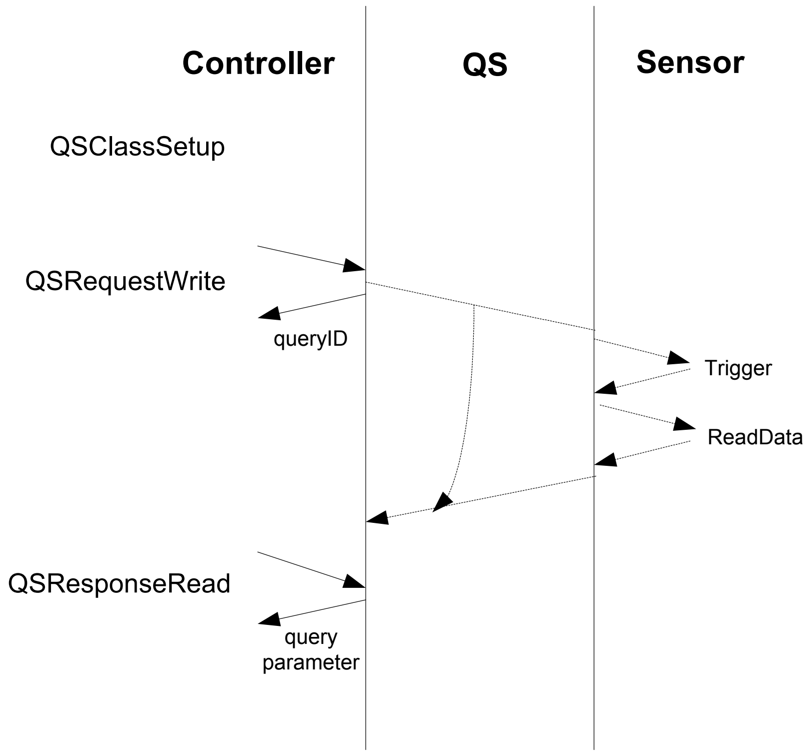

Figure 7: Query Service execution

QS may process a query in several ways:

1. The most common case is when QS needs to obtain a new measure from the sensors. If the controller issuing the query and the target sensors are located within different nodes, the process requires that the QS components located within the two nodes exchange a sequence of request and response messages. The interaction between the sensor device and the local QS component within the same instantiated node follows the specification of the IEEE 1451.2 standard described in Section 2. Once the sensor QS component gets the queried parameter from the targeted sensor, it encapsulates it in a packet and sends it across the network to the QS component located at the node including the controller that has initiated the query.
2. In addition to the normal gathering of data measures from sensors, the QS may perform additional functions such as aggregation to process the data coming from multiple sensors.
3. Instead of getting a new sensor measure, the QS may reuse local information previously gathered if the queried parameter is still available from a previous query and satisfies the timeliness requirement.

To get the queried parameter the controller calls the *QSResponseRead* primitive specifying the QueryID parameter. A query with a one-time response or an event notification terminates either when the response arrives or when a timeout of duration max_time set by the query initiator expires. The timeout prevents a query from staying active for an unnecessarily long time, especially when it is not known a priori how many responses will be received. It also allows the corresponding QueryID to be released. In the case of a periodic-response query, the query can be terminated at any time by the application simply calling the *QSStopQuery* primitive.

The QS architecture enables effective optimization of the network. First, messages over the network are eliminated if the requested parameter is already locally available. In addition, the number of parameters passed across the Application Interface is minimized. Since the parameters of a class of queries are set once using the *QSClassSetup* they have not to be specified by the application at each query invocation.

Additional primitives used by Virtual Sensors. Virtual Sensors necessitate the introduction of the following primitives, reading the parameter being queried, and writing the corresponding value, respectively — acting as counterparts for the corresponding functions in the controllers. As a special case, controllers acting as virtual sensors use them to interact with other controllers that have queried their parameters.

- ***QSRequestRead*** provides a virtual sensor with the parameter being queried

char QSRequestRead ( )
returns the requested parameter

- ***QSResponseWrite*** provides the parameter of the query and the value
int QSResponseWrite (char parameter, int value)*
returns OK or ERROR

Concurrency and synchronization semantics. For the proposed services to work correctly and for a correct operation to be verifiable, it is essential that their communication semantics are unambiguously defined.

- The communication among the AWSN components distributed across the network (controllers, sensors, actuators) is assumed asynchronous, because it takes place between components that do not necessarily have a common definition of time and communicate over a wireless channel with an unknown communication delay. While it is technically possible to synchronize all nodes in the network to a desired level of resolution — typically of the order of magnitude of the uncertainty of the communication delay—, and hence to realize a synchronous communication paradigm, the overhead of doing so generally outweighs the advantages.
- Based on these observations, the AI primitives have been defined with the following synchronization semantics in mind:
  (i) QS primitives called by controllers (as well as virtual sensors and actuators) return an immediate (local) response whether the call is accepted or is rejected due to an error. These calls are *blocking,* that is the controller waits until the response arrives. As the response time is immediate (that is, negligible with respect to the communication delay), the interactions between QS operations and the controllers can be considered to be synchronous.
  (ii) Interactions between distributed QS components are asynchronous, as observed above. To avoid controllers to be locked up for a long and unpredictable time while waiting for a response, remote interactions are *non-blocking*. This means that, while waiting for the response to QS query, a controller can move forward and perform other activities. Which type of asynchronous communication mechanism (e.g. message

passing, publish/subscribe) is used is an issue that concerns the service implementation rather than the Application Interface and hence is out of the scope of this paper. A discussion of different mechanisms to implement asynchronous communication can be found in [13][14].

Another implementation issue that is out of the scope of this paper is how primitives such as *QSResponseRead* are realized. Most implementations use one of the following two mechanisms:

- *Polling*—the primitive is repeatedly invoked until it returns the expected data.
- *Interrupt*—the primitive is decomposed into a *read_initiate* function that is used by the application to notify that it is ready to read the data in the input buffer, and a *read_interrupt* function that is called when the data reception event has taken place.

The same communication semantics apply to the primitives of the services described in the following sections.

4.3 Command Service (CS)

The ***Command Service (CS)*** *allows a controller to set the state of a group of components.*

A command is a sequence of actions initiated by a controller (*command initiator),* which demands a group of components (*command targets)* to take an action (Figure 8). The command is of type *confirmed* if the actuators send an acknowledgment to the controller after the action is taken (or when the request is received), it is *unconfirmed* if they do not send any acknowledgment.

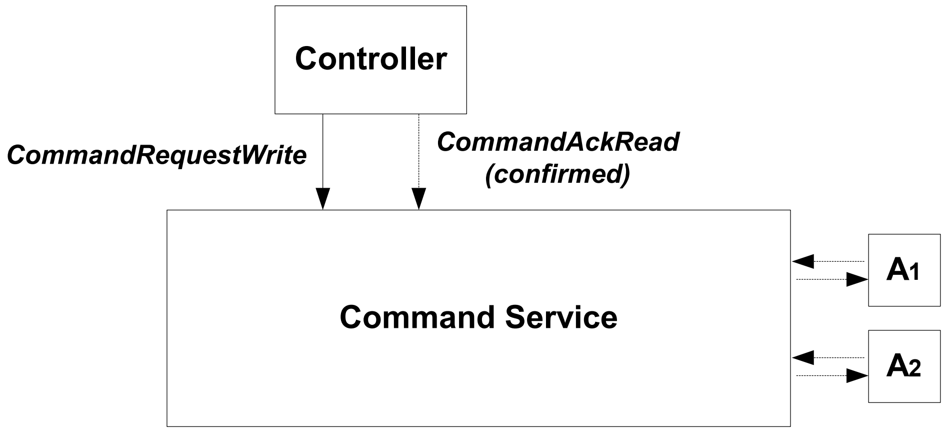

Figure 8. Command Service interactions

Primitives, command parameters, reliability, *CommandClass*, and *CommandID* are used in a similar way to the corresponding ones defined for QS. *CSClassSetup* and *CSClassUpdate* are used to configure and update the values of the parameters of classes of commands. *CSRequestWrite* is called by a controller to initiate a command, while *CSAckRead* is used in case of confirmed command to check if the acknowledgment has been received. *CSStopCommand* is used to terminate commands that require an actuator to take actions periodically.

The command primitives are:

- *int **CSClassSetup** (struct accuracy *a, struct resolution *r, int security, int max_time, int priority, int quantifier, int operation)*
returns the descriptor of the command class or ERROR

- *int **CSClassUpdate** (int CommandClass, struct accuracy *a, struct resolution *r, int security, int max_time, int priority, int quantifier, int operation)*
returns OK if the update is initiated or ERROR

- ***CSRequestWrite*** initiates a command of the type indicated in CommandClass to set a parameter of the components specified in *name_target
*int CSRequestWrite (struct name *name_target, char* parameter, int CommandClass, int confirmed, int reliability)*
returns a CommandID as a descriptor of the command or ERROR

- ***CSAckRead*** gets the (positive or negative) acknowledgment of a confirmed command
int CSAckRead (int CommandID)
returns OK in case of positive ack, ERROR in case of negative ack

- ***CSStopCommand*** is used by a command initiator to stop a command
int CSStopCommand (int CommandID)
returns OK or ERROR

Additional primitives used by virtual actuators. Virtual actuators use the following primitives to read the parameter being set and issue the corresponding acknowledgment. As a special case they are used to implement the interaction among controllers when one of them acts as a virtual actuator.

- ***CSRequestRead*** provides a virtual actuator with the parameter and the value being set
struct ParamValue CommandRequestRead ( )
returns the requested parameter and the value to be set

- ***CSAckWrite*** provides an acknowledgment if it is a confirmed command
int CommandAckWrite (int ack)
returns OK or ERROR

Communication between controllers. An Application may consist of multiple distinct controllers that cooperate and exchange information. An example is a pursuer-evader game where multiple pursuers exchange information on the control strategy they apply to chase an evader. When a controller needs to get the parameters of another controller, it uses the standard query primitives defined in Section 4.2. The target controller that is queried acts as a virtual sensor and provides the data being requested using the primitives provided in 4.2 for the realization of virtual sensors. Similarly, the CS is used to set the value of a variable in another controller and the targeted controller acts as a virtual actuator. No new primitives in addition to the ones defined in 4.3 are needed.

4.4 Concept Repository Service (CRS)

The **Concept Repository Service (CRS)** *maintains a repository containing the lists of the capabilities of the network and the concepts that are supported.*

This service plays a central role, as it assures that concepts are used in a consistent way by any component. Concepts may be introduced in the repository during the system configuration or during the system operation.

CRS holds the definition of the following concepts:

1. **Attributes,** used to define names. Examples of attributes defined in the domain of environment monitoring are temperature, light, and sound.
   Attributes are added to the repository (or updated) in two ways:
   - automatically by the CRS, when sensors (or actuators) that read or write a new attribute are added to the SNIP ("plug-and play"). The CRS reads the attributes listed in the sensor tag and adds the new ones to the repository. For example, if a temperature sensor is added to a network that does not have the attribute temperature yet, temperature is automatically added to the repository.
   - by an application that calls a primitive *CRSAddAttribute* and passes the name and the domain of the attribute as parameters

int CRSAddAttribute (char attribute)*
returns OK if the attribute is added successfully, ERROR otherwise.

The other primitives used by the application to access the list of attributes are:

char CRSGetAttributeList ( )*
returns the list of all attributes currently present in the repository

int CRSDeleteAttribute (char attribute)*
returns OK if the attribute is deleted, ERROR otherwise

int CRSCheckAttribute (char attribute)*
returns OK if the attribute is already present in the repository, NOT PRESENT otherwise.

2. **Regions**, used to define the scope of a name. A **zone** is a set of locations identified by a unique name. The name of a zone is added to the repository together with the zone boundaries, which are expressed in terms of spatial coordinates as described in Section 4.6. During the network operation, a node that knows its spatial location can get through the CRS also the name of all the zones that include its location.

The CRS uses a hierarchical tree-based structure to store and name zones that are included into each other, as this simplifies the search during the name resolution. For example, a zone of type "house" contains the zones corresponding to its floors, each of them containing the zones of its rooms. The full name of the zone corresponding to the DOP Center on the fifth floor in the EECS Department building in the Berkeley Campus is defined by the following tuple of identifiers (Berkeley Campus, EECS Dept., 5th floor, DOP Center). A more efficient way to identify a zone is to use the name of the leaf zone together with its parent in the hierarchy, e.g. (5th floor, DOP Center), if the name of the parent zone is already present in the repository. If no parent is specified in the primitive call, the defined zone becomes the root of a tree. If the CRS contains multiple zones having the same (parent, child) name pair, a primitive call always refers to the zone including the node that has called the primitive. An application may add new zones to the repository, delete, or list them using the following primitives that are based on the hierarchical naming scheme described above:

int CRSAddZone (char ZoneName, char* ParentZoneName, struct ZoneCoordinates *zc)*
returns OK if the zone is added successfully, ERROR otherwise.

char CRSGetZoneList ( )*
returns the name of all the zones currently present in the repository

int CRSDeleteZone (char ZoneName, char* ParentZoneName)*
returns OK if the region is deleted, ERROR otherwise

int CRSCheckZone (char ZoneName, char* ParentZoneName)*
returns OK if the zone is already present in the repository, NOT PRESENT otherwise.

Neighborhoods, on the other hands, are regions that are expressed by the proximity to a reference point, typically the (instantiated) node that invokes the call. Customized neighborhood definitions are added to or deleted from the CRS using the appropriate calls. Proximity is expressed either in terms of physical distance or routing distance.

int CRSAddNeighborhood (char NeigborhoodName, int radius)*
returns OK if the neighborhood is added successfully, ERROR otherwise.

int CRSAddHopNeighborhood (char NeigborhoodName, int NrOfHops)*
returns OK if the neighborhood is added successfully, ERROR otherwise.

int CRSDeleteNeighborhood (char NeighborhoodName)*
returns OK if the neighborhood is deleted, ERROR otherwise

char CRSGetNeighborhoodList ( )*
returns the names of all the neighborhood definitions currently present in the repository

int CRSCheckNeighborhood (char NeighborhoodName)*
returns OK if the neighborhood is already present in the repository, NOT PRESENT otherwise.

A number of neighborhood definitions are built-in, and do not need to be explicitly defined:

- The *one_hop_neighborhood* includes all the nodes that can be reached in one routing hop from the node invoking a query or command.
- The *two_hop_neighborhood* includes all the nodes that can be reached in two routing hops from the invoking node.

3. **Organizations,** used to define the scope of a name. Organizations indicate who owns or operates certain groups of nodes (e.g. the police, the fire department). The capability of distinguishing nodes by organization is necessary when there are nodes performing the same function (e.g. sensor nodes of the same type) in the same region, but are deployed and used to achieve a different task or belong to a different owner. Consider for example the case of a node that executes a management function and needs to address all the nodes that belong to the same organization to check their energy status. Another case is when the QS allows a controller to query only the sensor of its same organization to avoid the energy depletion of the nodes belonging to another organization or to reach data of a competing organization offering a similar service in the same area.
   Similarly to the region names, organization names are structured in a hierarchical fashion to simplify the search during name resolution. Organization names may be added, deleted, or read from the repository either directly by nodes using a plug-and-play mechanism (as described earlier for attribute insertion) or by an application using the following primitives:

int CRSAddOrganization (char OrganizationName, char* ParentOrganizationName)*
returns OK if the organization is added successfully, ERROR otherwise.

char CRSGetOrganizationList ( )*
returns the list of all the organizations currently present in the repository

int CRSDeleteOrganization (char OrganizationName, char* ParentOrganizationName)*
returns OK if the organization is deleted, ERROR otherwise

int CRSCheckOrganization (char OrganizationName, char* ParentOrganizationName)*
returns OK if the organization is already present in the repository, NOT PRESENT otherwise.

char CRSGetOrganization ( )*
returns a list of organizations to which the present node belongs

int CRSSubscribeOrganization (char OrganizationName, char* ParentOrganizationName)*
returns OK if ok, ERROR otherwise, allows nodes to subscribe to an organization

4. **Selectors**, **logic operators** and **quantifiers**, used to define names and targets of queries and commands. The following types are the most commonly used:
   - selectors: > n, < n, even, odd, =.
   - logic operators: OR, AND, NOT
   - quantifiers: all, at least k, any

The CRS plays a key role in the network operation because it allows distributed components to refer to common notions of attributes and regions. Moreover, it maintains agreement on these concepts in the presence of changes occurring dynamically during the network operation (e.g. new nodes join the network, or existing nodes move across region boundaries). **In line with the philosophy of this paper, we do not prescribe how the CRS is implemented.** Depending on the service implementation, it may be centralized or distributed over the network. In either case the correct operation of the system requires that the repository be updated in a timely manner when parameters change. For example, a node that moves from the kitchen to the living-room must update the region in which it is located and therefore must check which region includes its new coordinates. While the Location Service (Section 4.6) gives a node its new spatial coordinates, the CRS provides the node with the association of these coordinates with the "living-room" region. In addition to supporting the operation of a single network, CRS supports the interoperation of multiple networks (as discussed in Section 5) since it provides a complete and unambiguous definition of the capabilities of each network.

NOTE: Some concerns have been raised that the CRS may become unwieldy as the number of concepts grows. To reduce the overhead associated with setup and maintenance of the repository, it is possible to extend the current scheme and divide the concepts into usage classes that are configured and managed separately. A possible classification of concepts is: (1) *universal concepts*, specified for all sensor networks supporting this service model, (2) *domain-specific concepts*, specified for a given class of applications (e.g. environment monitoring, building automation), (3) *application-specific concepts*, specified for individual applications.

4.5 Time Synchronization Service (TSS)

*The **Time Synchronization Service (TSS)** allows two or more system components to share a common notion of time and agree on the ordering of the events that occur during the operation of the system.*

Typical application scenarios that require time synchronization among components are: "heat room at 6 pm," or "send me the temperature within 5 seconds".

Time is a quantity defined by a continuous and unbounded sequence of instants. It is measured counting the number of instants that separate the present instant from an origin instant chosen as starting point. Physical time can be described using a continuous or a discrete model. Typically, time is measured by counting the cycles of a reference clock. In addition, computers are able to store only finite numbers. Therefore, TSS is based on a discrete time model where the time granularity is determined by the clock resolution and the time intervals between clock ticks are equal.

The behavior of the system is defined by the sequences of actions taken by all its components. The occurrence of an action is called event. Each event occurs at a certain instant; therefore two distinct events can be simultaneous or ordered. In a timed model events are associated with timestamps whose values indicate the occurrence time and define an ordering relation.

NOTE: As time is a physical variable, its actual value can only be determined after the mapping process is completed. The description of TSS given below assumes that all the functional components (sensors, actuators, etc) mapped on the same physical node share a common clock. Hence, in the rest of this section the term node is used to refer to *instantiated nodes*.

TSS is used by network nodes to measure time and check the relative ordering among events. If the events to be compared belong to the same node (e.g. "retransmit message if acknowledgment is not received within 10 seconds") only local resources such as clock and timers are used. If the events belong to distinct nodes, TSS uses a distributed synchronization algorithm to ensure that the clocks of the nodes are aligned in frequency and time value.

A node can be in *synchronized* or *not-synchronized* state (Figure 9). If it is in not-synchronized state, time is measured by the local clock and is called individual time. If it is in synchronized state, the time value is agreed with one or more other nodes, which share a common reference. Synchronizing multiple nodes may require a time interval, called *synchronization time*, and can be achieved up to a certain specified accuracy. A transition between these two states may be triggered by the application or be the result of external events. In addition, due to clock deviations between nodes, nodes that are synchronized with each other may become not-synchronized after some time, unless a resynchronization operation is initiated.

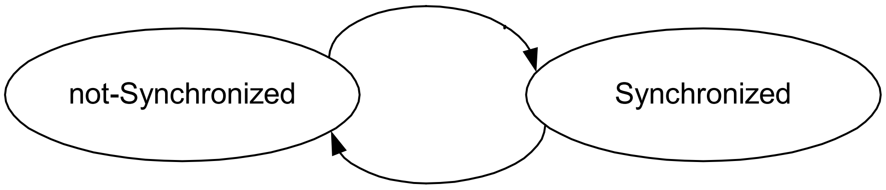

Figure 9. Synchronization states

Figure 10 shows an example of the time value of a node that is first in not-synchronized state and at time t1 starts synchronizing with another node. The synchronization time is Δt = t2 - t1 time units.

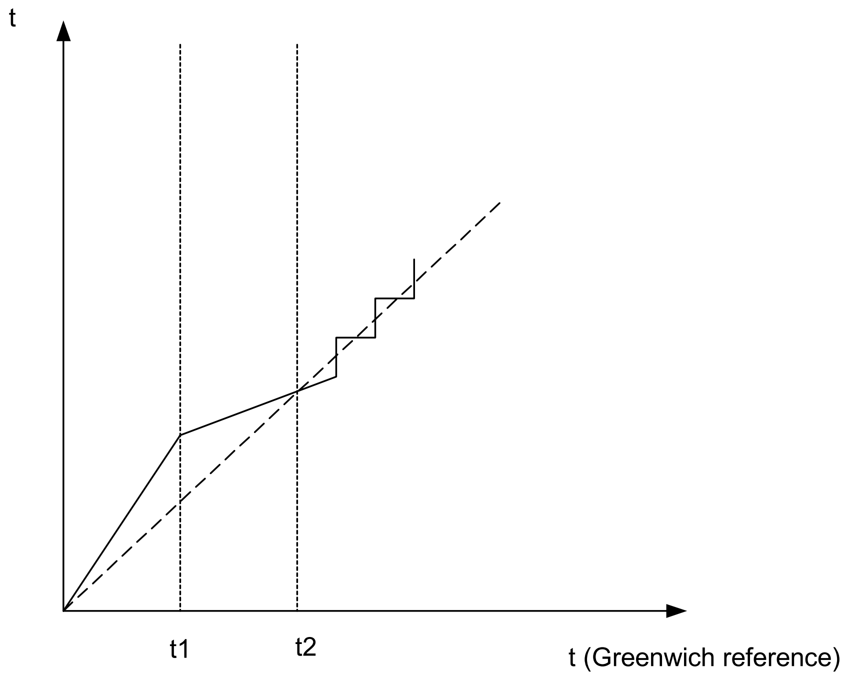

Figure 10. Time of a node in the Greenwich Reference Time

The following structure is used to represent time within TSS.

```
struct time {
        int        years;
        int        months;
        int        days;
        int        hours;
        int        minutes;
        int        seconds;
        int        milliseconds;
        int        microseconds;
        struct  window  timeWindow;
        int        isSync;
        struct name   *n;
};
```

Depending upon the application, different levels of time resolution and time span are needed. The members of the *struct time* present resolution accuracies from years to microseconds. The entry *timeWindow* of type *window* allows one to define the desired time resolution and span:

```
    struct window {
            int  *lsu;        /* least significant unit */
            int  *msu;        /* most significant unit */
    };
```

Members *lsu* and *msu* are pointers to the members of an instance of the time struct and indicate the beginning and the end of the *window* of the relevant units (e.g. a node requests time only in minutes and seconds). For a window of maximum size the most significant unit is year and the least significant unit is microseconds. For a window of minimum size lsu = msu.

The *time resolution* is defined by the least significant unit of the time window. Two events whose occurrence times differ for less than the resolution are considered simultaneous.

The *synchronization scope* of a node is defined by the set of nodes with which it is synchronized. Scope definition follows the naming scheme introduced in Section 4.1. A common approach is to enforce synchronization within a neighborhood, i.e. nodes that are within physical proximity. Requiring global synchronization over all the nodes in a system comes with a substantial amount of overhead.

By default, a node is not synchronized with any other node. The member *isSync* has value 1 if the node is synchronized with the nodes in the synchronization scope, 0 if it is not synchronized with any node. The last member is the pointer to a data structure containing the synchronization scope of the node, i.e. the region within which the present node is synchronized. A node can be synchronized with the Greenwich Reference Time. In this case an identifier of the Greenwich Reference Time is included in the scope of the node.

O*ffset* is the absolute time difference between the time of a component and the time of the component used as a reference. *Accuracy* is the maximum offset over time. Accuracy is defined with respect to the least significant unit (lsu) of the window and is expressed in parts per million.

```
struct accuracy {
        int max_offset;         /* max error */
        int *lsu;               /* least significant unit */
};
```

Availability and *security* are two optional parameters.

Availability is given by the upper bound of the period in which nodes that are using the service are not synchronized. This happens when synchronization is interrupted due to unexpected events such as network overload. The maximum time it takes the service to restore synchronization among the nodes defines the service availability.

Security is a parameter used to specify if the time provided by the service must be trusted and at which level of trust.

Table 1 lists the relevant parameters of TSS.

Table 1

Service Parameter	Domain
Synchronization scope	Names
Resolution	Lsu
Accuracy	Value lsu / ppm
Security (optional)	Levels of trust
Availability (optional)	Max time interval service not available

The TSS primitives are (the specified parameters are mandatory, optional parameters such as security and availability are omitted):

- *int TSSetup (struct resolution *r, struct accuracy *a)*
sets the resolution and accuracy of time and synchronization, returns OK or ERROR

- *int TSSUpdate (struct resolution *r, struct accuracy *a)*
updates the resolution and accuracy of time and synchronization, returns OK or ERROR

- *int TSSActivateSynchronization (struct name *n)*
activates synchronization with nodes defined in the scope *n, returns OK or ERROR

- *int TSSDisactivateSynchronization (struct name *n)*
disactivates synchronization with nodes in the scope *n, returns OK or ERROR

- *struct time TSSGetTime ( )*
returns the current time at the selected resolution and accuracy if available, otherwise it returns the time at the best offered resolution and accuracy

- *int TSSSetTimer (int t_interval, int *msu, int tid)*
sets timer tid to expire after t_interval units whose type is specified by *msu, returns OK or ERROR

- *int TSSResetTimer (int tid)*
resets timer tid, returns OK or ERROR

- *int TSSTimeout( )*
returns the ID of the timer which has expired

The sequence diagram in Figure 11 shows a scenario of invocations of primitives used by a node to synchronize with another node. First, *TSSActivateSynchronization* triggers the synchronization with another node. TSS handles the request exchanging messages with other nodes to achieve synchronization. *TSSGetTime* returns the agreed time if it is invoked after the end of the synchronization time period.

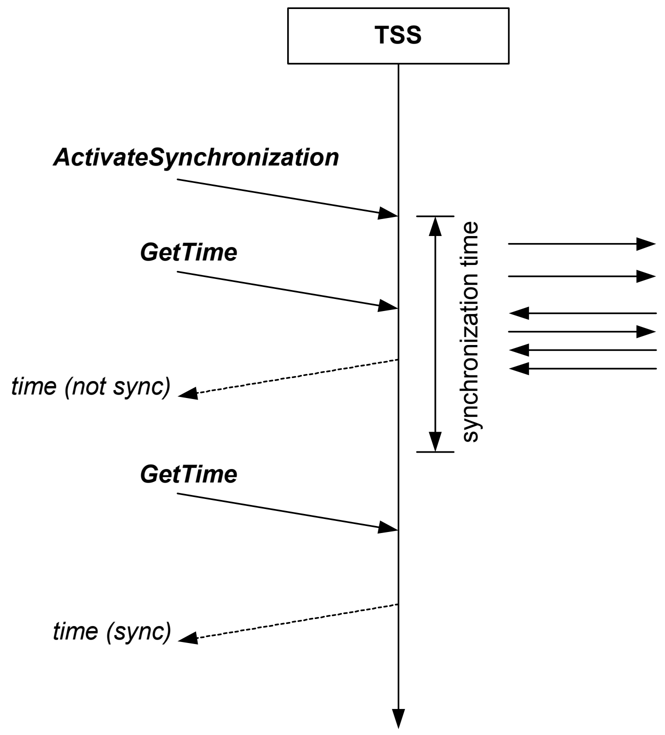

Figure 11. A TSS scenario

NOTE: Upon first reading, it may seem that the TSS definitions and procedures are quite "heavy-weighted", especially in light of the "light-weight" implementation requirements of AWSNs. A more detailed analysis will show however that the proposed approach combines *generality* with *efficiency*. Generality is essential given the fact that the range and scope of sensor networks is quite broad. However, the overhead incurred for generality is drastically reduced by the concepts of the CRS, *QueryClass* and *CommandClass*, which make it possible to reduce the size of generic query and command messages without giving up the generality. The same holds for the Location Service, introduced next.

4.6 Location Service (LS)

*The **Location Service (LS)** collects and provides information on the spatial position of the nodes in the network.*

Locations, Zones and Neighborhoods

The operation of sensor networks commonly uses the location of the instantiated nodes as a key parameter at several levels of abstraction. At the application level location is used for example to define the scope of names in queries (e.g. "send me the temperature measures from the kitchen"). Depending on the use, node location information can be expressed as a point in space or by a region where the node is located. Note that similarly to TSS, a node in this context refers to an instantiated node (or one of the logical components mapped onto it).

A point location, or simply ***location****, is defined by a reference system and a tuple of values that identifies the position of the point within the reference system.*

For the concept of region, we recall the definition given in Section 4.1: a region defines a set of locations. We have further differentiated between the concept of zone that is a set of locations identified by a common name, and the concept of neighborhood that is a set of locations identified by their closeness to a reference point.

LS currently supports the definition of location in a *Cartesian Reference System*, where location is expressed as a triple (*x*,*y*,*z*), with components *x*, y and *z* representing respectively the distance from the origin along the axis *x*, *y*, and *z*. A Cartesian Reference System is uniquely identified by the location of the origin and the orientation of the axis with respect to the Earth Reference System. These parameters are introduced at setup time, and may be updated during the network operation. LS can be easily extended to support other reference systems (e.g. Spherical Reference Systems) that in the present discussion are omitted for simplicity.

The following struct is used to represent a location in LS:

```
struct location {
        int        x;      /* expressed in cm */
        int        y;      /* in cm */
        int        z;      /* in cm */
        int        scale_factor;
        int        accuracy_type;
};
```

where members *x*, *y*, and *z* represent the values of the three coordinates expressed in cm, while *scale_factor* represents the exponent of the power of 10 by which one has to multiply *x*,*y*,*z* to obtain the effective location. The default value of the *scale_factor* is 0. *Accuracy_type* indicates whether the location is given at the requested or at the best possible level of accuracy.

A zone can have the form of a block, a sphere, a cylinder, or a more complex shape. A block is represented in terms of the coordinates of four vertices, while a sphere is represented by the center and the length of its radius expressed in cm.

The following struct is used to express the coordinates of a zone. *ZoneType* indicates if the zone is a cube, a sphere or of another basic shape, while the member *coordinates* points to a list including the coordinates of the corresponding zone. If *ZoneType* = ”cube”, the list includes the coordinates of four vertices; if *ZoneType* = ”sphere”, it includes the coordinates of the center and the size of the radius.

```
struct ZoneCoordinates {
            char ZoneType;
            int* coordinates;
};
```

More complex shapes of solid objects can be represented in terms of the composition of basic shapes.

Earlier, a *neighborhood* was defined as a region expressed by proximity to a reference point. Proximity is expressed either in terms of the Euclidean distance from a given node (spherical region) or by the number of routing hops from the node (See 4.4).

LS Parameters

The LS primitives use the following set of parameters.

- *Resolution* is defined by the scaling factor. If the scaling factor is 0, the resolution is in cm.
- *Accuracy* is the maximum location error over time. Accuracy is defined with respect to the least significant unit (lsu) corresponding to the smallest scaling factor and is expressed in parts per million.
- *Reference* is defined as a struct whose members contain the relative position of the origin and the orientation of the coordinate axis of the Cartesian Reference System with respect to the Earth Reference System (universally defined in terms of longitude, latitude, and height above sea level with respect to the center of the Earth).
- *Availability* and *security* are two optional parameters.
  Availability is given by the upper bound for the time period in which LS is not able to provide nodes location. This happens when LS is interrupted due to unexpected events such as failures or network overload. The maximum time it takes before the service restores synchronization among the nodes defines the service availability.
  Security is a parameter used to specify if the location provided by the service must be trusted and at which level of trust.

Table 2 lists the relevant parameters of LS.

Table 2

Service Parameter	Domain
Resolution	Lsu
Accuracy	Value lsu / ppm
Security (optional)	Levels of trust
Availability (optional)	Max time interval service not available

LS Primitives

- *int LSSetup (struct resolution *r, struct accuracy *a, struct reference *rs)*

sets the resolution and the accuracy of location measures, and the reference system, and starts the LS service. Returns OK or ERROR

- *int LSUpdate (struct resolution *r, struct accuracy *a, struct reference *rs)*

updates the resolution and accuracy of location measures and the reference system, returns OK or ERROR

- *struct location LSGetLocation ( )*

returns the location of the present node

- *char* LSGetRegions (struct location *loc)*

returns the list of regions (zone or neighborhood) of which the specified location is a member. If no location is specified, it returns the list of regions of the present node.

It is often useful for a controller to know what type and how many nodes are located in a given region. Also, it maybe useful to know if these nodes are static (that is, have not moved for a long time), or are mobile. One might consider setting up separate primitives for these important functions. However, we feel that they are already adequately covered by the primitives defined in the Query Service, namely *QSRequestWrite* and *QSResponseRead* using the parameters "location" and "mobility". To find the number of temperature sensors in the kitchen region, it suffices to launch a query for parameter "location" with name "(temperature, kitchen)" and to determine the cardinality of the returned list (which actually contains information on the actual location of each of these nodes as well).

4.7 The Resource Management Service (RMS)

*The **Resource Management Service (RMS)** allows a controller to read and/or set the value of a physical parameter of the SNIP.*

For example, using RMS a controller can read the amount of energy remaining in a node or query for the quality of the communication channels, or even set the clock frequency of a node. This service is particularly useful for system managers who want to diagnose a system or tune its parameters. However, the service can just as well be used by other services in the SNSP to automatically optimize their performance (for instance, the query/command services can use information obtained from the RMS to extend the lifetime of the network).

Strictly speaking, the RMS does not require any new AI primitives. The mechanisms offered by the QS and the CS are sufficient to support the requirements of the RMS. The main difference is that the parameters being set or queried are not of the environment but of the physical nodes constituting the network. Hence, the RMS operates on a different list of parameters than QS and CS. In addition, different naming attributes are used in the addressing. For instance, a typical query might be: "return the energy levels of all nodes in the kitchen zone".

The RMS is an important part of our proposal as it allows cost information to be visible to applications and provides transparency. This is crucial in energy and cost constrained sensor networks.

5. Interoperation of Sensor and Global Networks

A large number of applications require interoperation between sensor networks and global networks, such as the Internet. This is motivated for instance by the need for remote observation or management of the sensor network. This section presents how the functionality presented by the SNSP can be exported over network boundaries.

5.1. Interconnection of Sensor Networks

Sensor networks may be interconnected together and provide applications with a common set of services if they have the same understanding of the naming conventions. Each of the individual sensor networks that are interconnected may, however, support a different set of concepts and services parameters (e.g. different levels of accuracy, reliability, security…). Any controller can issue a query for any region or organization, also belonging to another network. When a network is not able to provide a service requested by an application (e.g. queries of a certain parameter, reliable/secure queries…), it returns a negative message to the requesting application. As a result, an application gets service support only from some of the networks and receives negative answers from the networks that cannot support it.

Sensor networks can be interconnected as shown in Figure 14: a) through a gateway, b) through a global network. When the interconnection is through a gateway, the service request messages issued by an application in one network are forwarded to the other network after the gateway has checked that the second network is able to support the service. When the interconnection is through a global network, packets carrying requests and responses are encapsulated and tunneled over the transport layer of the global network (e.g. UDP if the global network is the Internet). In this case, the gateway that interfaces the global network to the target sensor network (G2 in Figure 14b) is responsible to check whether the latter is able to support the requested service.

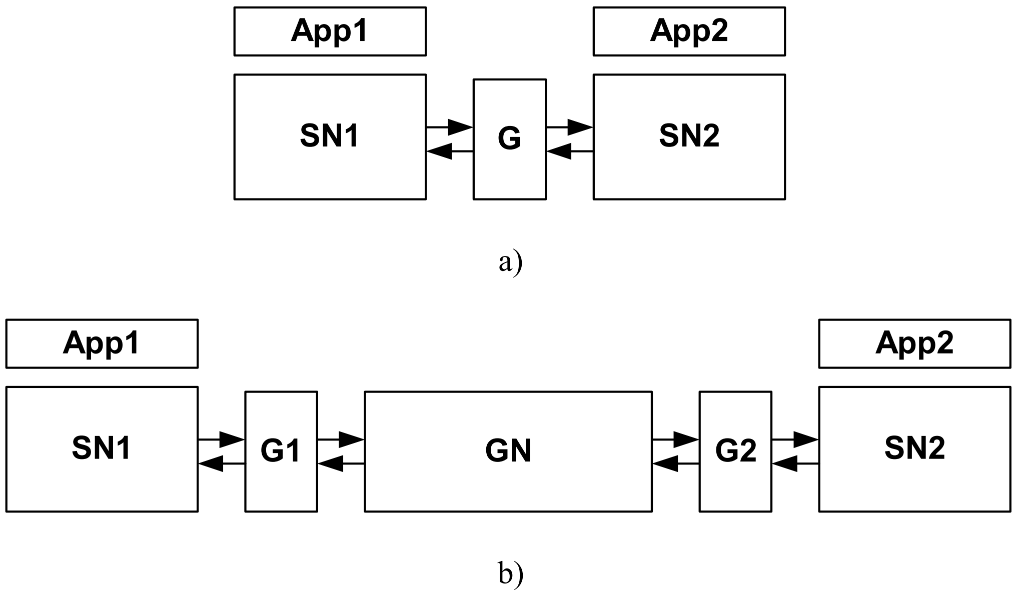

Figure 14: Interconnection of Sensor Networks

5.2. Access to global services from Sensor Networks

A sensor network application may want to use services offered by a global network. An example is for instance to display information obtained by a controller on a computer terminal connected to the Internet, or vice-versa to issue control commands to the controller. Another example might be to access a global weather report published on a web-server.

Global services are accessed through a gateway that interfaces the sensor network with the global network. Through the gateway, the global network appears to the rest of the sensor network as a virtual sensor or a virtual actuator: thus, it might be queried or set by a command. In the example of the query of the weather forecast over the Internet, the gateway and the rest of the global network define a virtual sensor, which delivers data to the sensor network application (Figure 15). How the virtual sensor is implemented is out of the scope of this paper.

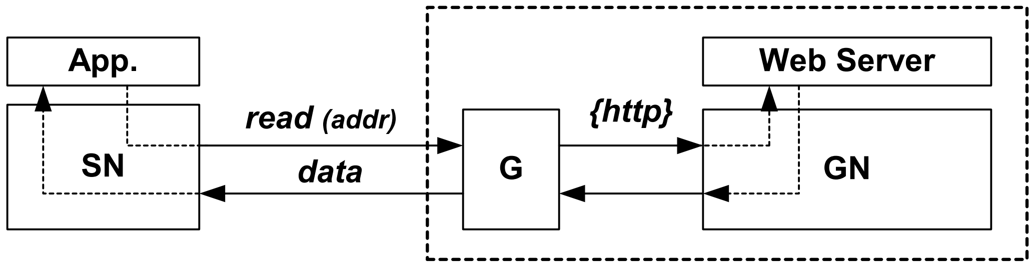

Figure 15: Access to global services

5.3. Usage of Sensor Networks for provision of global services

A global network may use sensor network services to support its own services. A typical application example is the remote monitoring from an Internet host. For example, consider the scenario of a user who wants to keep track of the status of the utilities and check the security in his home from a remote location. The sensor network includes sensors that are associated with the utilities and monitor their current energy and temperature level, sensors that are associated with doors and windows and detect the arrival of intruders. A home gateway connected and accessible from the Internet collects this information and makes it available to the remote user.

An application located in the global network has direct access to primitives of the sensor network service, and knows the address of the gateway that interfaces the global and the sensor network. The service request issued by the global network application component is sent to the gateway of the target sensor network. The gateway forwards the request to the sensor network, gathers the responses (possibly doing some aggregation), and sends them to the requesting application. Note that this scheme requires within the global network only one address for the gateway of a specific sensor network.

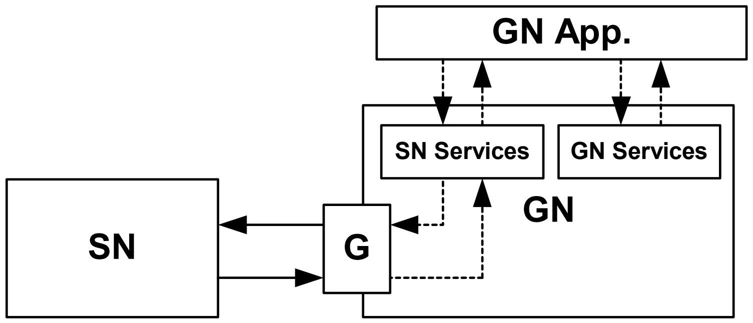

Figure 16: Accessing a Sensor Network from a Global Network

6. Mapping the Application Interface abstraction onto existing platforms

The present section shows how the Application Interface we propose applies and co-exits with two existing platforms: TinyOS and Picoradio.

6.1 Environment monitoring application based on Picoradio networks

Picoradio [15] is a low-power platform for sensor network applications targeting the following requirements:

- nodes must be smaller than one cubic centimeter, weigh less than 100 grams, and cost substantially less than one dollar
- the network must provide continuous and long term operation without frequent battery replacements maintaining a power-dissipation level below 100 micro-watts
- the network must be adaptive, self-organizing and decentralized, i.e. without single points of failure, and must support node mobility
- the network must support low data rate communication in the order of few bits per seconds

Picoradio meets these requirements by using highly optimized and energy efficient solutions at each design step both at the functional and at implementation level.

The Picoradio team at the BWRC has developed several generations of testbed and chip implementations over the years. Below we refer to the testbed implementation described in [16] and discuss its use to support an application that monitors the BWRC environment.

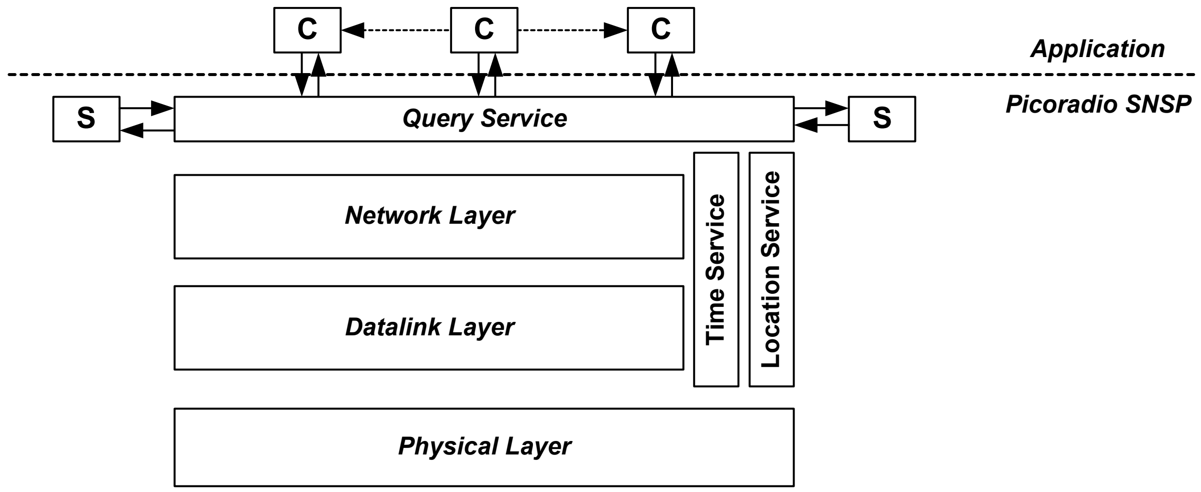

Figure 17: Picoradio sensor network platform

Picoradio is a distributed sensor network platform that offers services for control and monitoring applications at low energy cost. The Picoradio architecture, shown in Figure 17, includes:

- a network layer that supports multi-hop connectivity among end points,
- a data-link layer that performs error correction and medium access control for one-hop communication,
- a time service managing a local clock,
- a location service based on an algorithm that allows each node to compute its location with respect to the reference coordinates
- a physical layer based on a wakeup radio that allows physical connectivity among nodes within the transmission range

Applications relying on the Picoradio platform use an attribute-based naming scheme where nodes are identified by the type of sensors that is queried and by their location. Location is expressed either as a specific point in (x, y, z) coordinates or as a region identified by the tuple with the minimum and maximum values for each coordinate (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}).

The BWRC monitoring application is composed of a set of controllers that query light, temperature, humidity and sound sensors, and uses the following services offered by the Picoradio platform:

1. *Query*. Controllers issue a query request (sending an interest packet) and get the requested data as a response. Picoradio supports three types of response patterns: one-time response, periodic responses, and notification of change. Multiple queries of the same sensors can occur concurrently: each query request is stored within the target node in a local list and removed only when satisfied.
2. *Location*. Multiple layers use the location service. The routing and the MAC layers use location information to simplify the network discovery and maintenance procedures and save energy (e.g. forwarding packets only towards the destination). The query service uses the location of the controller as a source address, placing it in a field of the request packet to indicate where the response should be sent.
3. *Time.* The platform does not support synchronization among distinct nodes, but allows an application to access local time information and use timers. Controllers use timers to define the maximum time a response is to be awaited, while the query service component in a sensor

node uses time services to know when the next measure should be read in queries with periodic responses.

In conclusion, the testbed implementation of Picoradio provides limited support to an application offering only some of the services described in this paper. In particular, the platform does not use a CRS, and therefore the set of supported attributes is defined at system configuration time. The application interface of the Picoradio service platform is compatible with the Application Interface defined in this paper.

6.2 TinyOS and TinyDB

TinyOS [17] is an operating system specifically designed for sensor networks and based on an event-driven programming model suited for concurrent specifications. It allows the capture of a system specification as a graph of components that communicate through an interface of events and commands. Commands are non-blocking requests made to lower level components, while events model the responses coming from the lower to the upper layers. Components are implemented as tasks that have run-to-completion semantics and can be preempted only by events.

TinyOS provides an abstraction of hardware components such as sensors, radios and timers and defines a platform for application developers that allows them to specify the system using nesC. nesC is a system programming language for network of embedded systems derived from C reducing its expressive power (e.g. no dynamic memory allocation) to improve code safety and adding the component-based structure that greatly improves code development. In a sense, TinyOS can be considered an abstraction of the SNIP layer, introduced earlier in this paper.

Even though nesC and TinyOS are tailored for the sensor network domain, writing an entire specification including the application, the service platform and the protocol stack for a distributed system of a certain complexity is still a challenging task. First, the concurrent nature of the specification requires partitioning the specification into tasks at the proper granularity and defining efficiently the interfaces among components. Second, the details that the programmer has to deal with while programming in nesC are many and therefore the specification process would benefit from beginning at higher levels of abstraction.

The Application Interface primitives we defined in this paper are compatible with the split-phase communication model of TinyOS, and can be expressed as component interfaces in TinyOS.

To reduce the burden for the application developer [8] has defined a higher abstraction based on a data-centric approach called TinyDB, which views a sensor network as a distributed database accessible by an external source through declarative queries. Declarative queries in SQL-like form are certainly simpler and safer to write from an external user perspective then writing nesC programs with lots of implementation details concerning the use of shared resources in TinyOS. A user does not have to specify how data is acquired or which protocols are used in a query, but only what type of data, from which sensors, how frequently etc. In addition to queries TinyDB allows to specify also commands to set actuators by means of trigger actions that are associated with queries and executed when a predicate on the queried values is satisfied. One of the key features of TinyDB is its support of in-network data aggregation to reduce network traffic and minimize power consumption.

TinyDB defines a programming model that is at a higher level of abstraction then the one offered by TinyOS. The TinyDB primitives and set of parameters defined are compatible with our

Application Interface definition. However, our goals and our scope are quite different from the ones of TinyDB. We aim at the definition of a universal interface usable by all types of applications in the sensor network domain to access any type of services they may need and we do not worry about the service implementation. TinyDB defines a platform, along with its implementation, offering only a subset of these services and therefore has a more limited application scope. Hence, TinyDB should be seen as a component of a Sensor Network Service Platform implementing the proposed Application Interface. The existing query mechanism used in TinyDB could easily be made compatible to the primitives defined in the AI.

7. Summary

In this white paper, a service-oriented platform for the implementation of AWSN applications is presented. It is our belief that by defining an application interface at the service layer, it should be possible to develop and deploy sensor network applications while being agnostic about the actual network implementation yet still meeting the application requirements in terms of timeliness, lifetime, etc.

This document is clearly not intended to be the last word on this topic. Rather it should be considered as a baseline: feedback and discussion are welcome. It is our intention to realize in the coming year prototype implementations of a number of applications on at least two different network implementations using the concepts outlines in this white paper.

Finally, it is worth observing that some of the concepts introduced in this white paper have broader applicability than AWSNs. For instance, a concept such as the CRS would also be useful in the operation of ad-hoc multimedia networks. In fact, the development of a service-based application interface for this emerging class of applications in a style similar to the one presented here seems like a logical next step – the potential success of ambient intelligence hinges on the simple and flexible deployment of both media and sensor networks and the interoperability between the two.

Acknowledgments

This document is the result of an arduous process, including many long discussions and e-mail trains. The authors acknowledge and value the contributions and inputs of a wide range of people {enumerate}.

REFERENCES

[1] D. Snoonian, "Smart Buildings," IEEE Spectrum, pp. 18-23, September 03.

[2] J. Rabaey, E. Arens, C. Federspiel, A. Gadgil, D. Messerschmitt, W. Nazaroff, K. Pister, S. Oren, P. Varaiya, "Smart Energy Distribution and Consumption —Information Technology as an Enabling Force," White Paper, http://citris.berkeley.edu/SmartEnergy/SmartEnergy.html.

[3] G.Huang, "Casting the Wire," Technology Review, pp. 50-56, July/August 2003.

[4] F. Boekhorst, “Ambient intelligence: The next paradigm for consumer electronics”, Proceedings IEEE ISSCC 2002, San Francisco, February 2002.

[5] IEEE 802.15 WPAN™ Task Group 4 (TG4), http: //www.ieee802.org/15/pub/TG4.html

[6] The Zigbee Alliance, http://www.zigbee.org

[7] IEEE 1452.2 “Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats”, IEEE, 1997

[8] S. Madden, “The Design and Evaluation of a Query processing Architecture for Sensor Networks”, Ph.D. Dissertation, UC Berkeley, 2003

[9] W. Adije-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley, “The design and implementation of an Intentional Naming System”, in Proceedings of Symposium on Operating Systems Principles, Dec. 1999

[10] C. Intanagonwiwat, R.Govindan, D. Estrin, “ Directed Diffusion: a scalable and robust communication paradigm for sensor networks”, in Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCom 2000), August 2000, Boston, Massachusetts

[11] P.V. Mockapetris, K. Dunlap, “Development of the Domain Name System”, in Proceedings of SIGCOMM ’88, Stanford, CA, 1988

[12] D. Mills, “Internet Time Synchronization: the Network Time Protocol”, in Global States and Times in Distributed Systems, IEEE Computer Society Press, 1994.

[13] P. Eugster, P. Felber, R. Guerraoui, a. Kermarrec, “The many faces of Publish/Subscribe”, ACM Computing Surveys, Vol. 35, No. 2, pp 114-131, June 2003.

[14] S. Edwards, L. Lavagno, E. Lee, A. Sangiovanni-Vincentelli, “Design of Embedded Systems: Formal Methods, Validation and Synthesis”, Proceedings of the IEEE, vol. 85 (n.3) - March 1997, p366-390

[15] J. Rabaey, J. Ammer, J. da Silva, D. Patel, S. Roundy, “Picoradio Supports Ad-hoc Ultra-low Power Wireless Networking”, Cover Article, IEEE Computer Magazine, July 2000.

[16] F. Burghardt, S. Mellers, J. Rabaey, “ The Picoradio Test Bed”, White paper, December 2002

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, “System architecture directions for networked sensors”, ASPLOS 2000