Discovery, monitoring and management in smart spaces composed of low capacity nodes

Citation for published version (APA):

Ozcelebi, T., Lukkien, J. J., Bosman, R. P., & Uzun, H. Ö. (2010). Discovery, monitoring and management in smart spaces composed of low capacity nodes. *IEEE Transactions on Consumer Electronics*, 56(2), 570-578. <https://doi.org/10.1109/TCE.2010.5505972>

DOI:

[10.1109/TCE.2010.5505972](https://doi.org/10.1109/TCE.2010.5505972)

Document status and date:

Published: 01/01/2010

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Discovery, Monitoring and Management in Smart Spaces Composed of Low Capacity Nodes

Tanır Özcelebi, Johan Lukkien, Remi Bosman and Önder Uzun

Abstract —*This paper introduces a lightweight architectural solution for discovery, monitoring and management of nodes, services and resources in smart spaces composed of low capacity nodes. A wireless sensor network software architecture is used as the baseline and is extended to a scalable smart space architecture. Discovery, monitoring and management tasks are realized by means of a Resource Manager that acts as just another node. The proposed architecture is deployed on physical wireless sensor nodes. Experimental results show that the proposed smart space architecture is indeed lightweight and suitable for very low capacity nodes¹.*

Index Terms —*Lightweight smart space architecture, low capacity node, service discovery, resource discovery*

I. INTRODUCTION

Recent developments in miniaturization, wireless networking and sensor technologies combined with profound contemporary research reveal the vision of *smart spaces*. A smart space is a physical space (e.g. home, office) that contains embedded networked electronics which act and interact in a highly dynamic and context-aware way, providing automatic and adaptive services to its users. This idea is inspired by pervasive computing and especially context aware computing and extends them, envisioning seamless integration of smart devices into users' everyday life. Thus, the focus has shifted from devices and technicalities to users and their tasks.

Nodes within the smart space communicate and collaborate in order to provide advanced services to users with minimal user interaction and maximal autonomous operation. Nodes include devices that users wear or carry and thus nodes can join and leave the smart space arbitrarily. Therefore, the smart space must automatically adapt to changes in network composition and topology as well to a change in context. This requires a node to expose its existence, services and resources so that applications can be formed from this.

Contemporary frameworks in the field (e.g. OSGi [1], NoTA-M3 [2]) provide a smart space platform or framework. Their layered architectural design gives rise to abstraction and modularization, which eases application development

significantly. This approach works if the smart space is composed of rather capable nodes, e.g. PDAs, mobile phones and laptop computers. However, since low-capacity nodes cannot interpret the correspondent protocols (often xml-based) or run powerful application stacks these devices are often included using a proxy, leading the communication through some gateway. This leads to an increased latency, to software systems that are difficult to install and manage and, most importantly, to systems in which there is no in-network processing since everything is based on point-to-point connections between proxy and node.

The class of low-capacity devices satisfies the following criteria: 1-25 MHz CPU clock frequency, 1-10 kbyte RAM, 1-128 kbyte programmable ROM, a communication bandwidth of several kilobytes per second typically using the frequency band of 433MHz-2.4 GHz, a 2.0 – 2.5 Ah or less electric charge, and i/o devices such as sensors and actuators. Typical wireless sensor nodes are a very good representative of this category of low-capacity nodes as shown in Fig. 1. Hardware elements of a wireless sensor node are as shown in Fig. 2.

Fig. 1. Low capacity wireless sensor node. Shown is the Sensixa node marketed by a spinoff from Imperial College London.

Fig. 2. Hardware Elements of a Wireless Sensor Node

The main focus of this work is to facilitate low capacity nodes in smart spaces by genuinely integrating them as a distributed computing platform. We show that our Wireless Sensor Network (WSN) architecture called *Open Service*

¹ This work was supported by Smart Objects For Intelligent Applications (SOFIA) project funded by the European Artemis programme.

T. Özcelebi, J. J. Lukkien, R. P. Bosman and Ö. Uzun are with the Department of Mathematics and Computer Science at Eindhoven University of Technology, 5612 AZ, Eindhoven, The Netherlands. (e-mail: t.ozcelebi@tue.nl, j.j.lukkien@tue.nl, r.p.bosman@tue.nl, h.o.uzun@student.tue.nl).

Architecture for Sensors (OSAS) [3] is a suitable architecture and framework for a smart space with low capacity nodes. We develop lightweight methods for automatic discovery of smart space components and for monitoring their services and resources at run-time. This functionality is extended to map service requests to devices that support these services by selecting the most suitable provider and consumer of the service. We propose a Resource Manager (RM) node to carry out these tasks at run-time. The main contribution of this paper is to leverage a low capacity network into a smart space network by realizing the fundamental smart space functionalities, i.e. node, service and resource discovery (node registration), monitoring (reporting introspection information) and management (adaptation and task assignment).

The paper is further organized as follows. In Section II, related work is discussed. In Section III, the OSAS framework is presented. In Section IV, the proposed solution is introduced. In Section V, experimental results are presented. Finally, Section VI concludes the paper.

II. RELATED WORK

There is a profound research on smart spaces; nevertheless solution approaches are designed mostly for high capacity nodes. In [4], an approach oriented towards high-mobility smart space environments is presented using Bluetooth service discovery. JCAF [5] implements an event based subscription model and uses JINI [6] for service discovery. These approaches lack in supporting highly dynamic environments with high availability and low resource usage. Due to burdensome requirements, such as the Java Virtual Machine and IP based addressing, cost of communication is increased. This results in hindering low capacity nodes to join the network. In [7], a centralized gateway solution is proposed, having an asynchronous event-subscription model using Service Locating Service for dynamic service discovery. The solutions proposed in [8] and [9] focus on decoupling the application layer from the middleware layer providing integration of management into applications. This layered architecture approach is useful as it brings ease for application developers through abstraction. However, this framework limits optimization in energy expenditures and resource utilization as cross layer optimization option is sacrificed.

Architectural solutions such as [8], [10] and [11] propose XML based data models for messaging. Many other solutions introduce even more complex data formats for better structure and more expressive power. Nevertheless this practice brings an extra burden on the nodes and boosts up energy consumption with message format overheads.

Solutions which focus on contextual information gathering and management [12], have poor or no support for resource discovery. Resource discovery mechanisms are rarely used in contemporary frameworks [13]. Nodes are assumed to be stable and permanently available. Therefore, these solutions disregard mobility needs, as discussed in [14], and dynamicity is mostly not taken into account.

III. OSAS FRAMEWORK

We see the use of a smart space roughly as follows. Each node entering the space can discover available services and resources. Using this information, the node can impose a behavior on this space by employing the found services (*service sharing and coordination*) and by deploying new services (*programming and configuration*). For this, it must decide where to run these services based on available resources. Finally, there must be seamless *integration with the IP infrastructure*. As a rather simple example, the new node can discover a set of lighting controls and sensors, with location information and install a service that manages the light according to a private policy [15]. In order to implement the proposed smart space functionalities in a lightweight way, we base ourselves on an existing framework for programming sensor networks, i.e. OSAS [3]. The mentioned concepts of service sharing, dynamic programming, configuration and coordination are an integral part of the framework.

An OSAS network consists of nodes equipped with a specific run-time system. An OSAS application is a single program for the entire network that defines services as well as connections between these services. Services expose time-triggered eventing to the network and actions that can be called from the network; connections (*subscriptions*) connect an event generator with an action (called a *handler* in that case), most often on different nodes. Both services and subscriptions can be installed and removed on-the-fly by any node that is sufficiently authenticated. The single program for the entire network can be extended on-the-fly as well.

Local node functionality is made available to the run-time system by a set of *system calls*. The run-time system can be run also on a powerful node, e.g. a laptop. A connection with the IP infrastructure is formed by a gateway node – having access to both regular IP infrastructure and the wireless sensor protocol like 802.15.4- and running the run-time system on that node as well. In this way we overlay the sensor network into the IP infrastructure.

In more detail, a service has a state, generated events and event handlers. The flow of the program is determined by the events and a service can generate events through event generators or consume events through event handlers. The process of linking the event generator of one service to the event-handler of another is called a subscription. When a subscriber subscribes to an event, the event generator is triggered periodically. The triggering period is determined by the subscriber. There could be many subscriptions on a single service and the subscription period may differ per subscription. When an event generator is triggered, it tests its condition to fire an event. A generated event takes the form of calling the handler as an *asynchronous remote procedure call*. In Fig.3 we show a picture of the run-time organization of a single node. The services run as part of the run-time system and are written in a specially designed bytecode [3]. For the communication, the availability of a link layer (i.e. a neighborhood communication facility) is assumed, binding OSAS messages to a sensor MAC and a UDP overlay.

Fig. 3. OSAS runtime organization

The programming platform is realized using a toolchain as shown in Fig. 4. A program is written in WASP [3] Service Composition Language (WaSCoL), to define services and compose them (subscriptions). The output of this is a set of service definitions and subscriptions that are packed into messages and can be handled as independent objects.

Fig. 4. The OSAS Toolchain: .wsp files contain the network program and are compiled into byte code (.wbc) as well as an .xml description. The latter is used as input to the compiler if someone wants to add new services. It also contains the available system calls. The run-time system can be run on a simulator as well thus giving a powerful integration with the IP-based infrastructure, allowing to integrate more powerful nodes into the sensor network. Simulators are connected using UDP connections that extend the sensor broadcast domain. The simulator is configured using a description in a .wnl file.

Program deployment is done by emitting the generated messages to the network. This is typically done by the loader but can be done by other nodes as well. For example, a node may store a message to establish a certain subscription *at run-time* upon a condition, by simply emitting this message.

The power of the system comes from the fact that there is a single program, hence no need for complicated interpretation and lookup inside the nodes. The symbol table (the .xml file in Fig. 4) maps names of system calls, services, subscriptions, handlers, events and variables onto short IDs such that messages remain small. These IDs are static and globally defined in the network. The updated symbol table generated by the compiler can be used to extend or reconfigure an already running network.

There are two high level constructs for communication in OSAS. *SendMessage* is used to flood a message (representing an asynchronous remote procedure call) to the whole network whereas *SendToSubscribers* is used to route a packet to subscribers for a service. A route is set from the service provider to the subscriber either via a subscription request (automatically) or by flooding of a *SetRoute* message. Therefore, a subscription can only be issued by the subscriber. Each node exposes a *SubscriptionHandler*. When called with the proper arguments, this handler will issue a subscription. WaSCoL provides content based addressing (CBA) besides regular addresses. In a CBA the destination is determined by a predicate. Besides the mentioned *SubscriptionHandler* there is a handler to install bytecode. Together, these make up the basic functionality of the system.

IV. PROPOSED SOLUTION

We propose a central *Resource Manager (RM)* for discovering, monitoring and managing nodes and their resources/services in the smart space. RM has access to information regarding existing network nodes, their services and subscriptions, and types of resources that are available on them. RM may make this information available again as a service for other nodes to subscribe, or use it for taking service and resource mapping decisions. The optimal usage of such information is beyond the scope of this paper.

Service Discovery

RM is a specialized node extended with dedicated system calls, event generators and event handlers, administering the network to exhibit a smart space behavior. At the architectural level, this dedicated functionality is called *Smart Space Administration*. The *RM Repository* contains node entry objects, every one of which represents a node. When nodes are detected, information regarding node type, services and resources is stored in the repository as a node entry object.

All service discovery functionality is accomplished by a soft-state service discovery protocol, known as the binary state protocol or the *heartbeat protocol* [16]. A node sends its heartbeat signal to RM periodically to denote that it is present and operational. Hence, the node presence has to be maintained by periodic refreshments of its state and not by explicit declaration (as compared to a hard state protocol).

When RM receives a heartbeat signal, if the node is already registered, RM updates the state of the node by resetting the timeout counter of the corresponding node entry. Else, RM creates a node entry in its registry and prompts for relevant node information. Upon missing a specified number of heartbeat signals, RM infers that the node has either failed or left the network. Thus, the node is unregistered, i.e. its node entry on the repository is deleted.

The heartbeat protocol maps trivially onto the OSAS framework. Nodes run the HeartBeat service (it is then called HB node) on which the RM subscribes with its HeartBeatHandler. The HeartBeat frequency is given in the subscription. The content of the HeartBeat is filled in by the HeartBeat service when it fires.

Node discovery takes place upon initiative of the RM. It periodically advertises its presence by flooding a subscription message through the network. When a new node arrives in the smart space it receives and stores the subscription and starts reporting its presence periodically through heartbeat messages. If a node is already in the network and receives the advertisement message it overwrites the existing subscription. Periodic advertisement makes sure that any new node entering the system is discovered: i) it constructs a route to reach RM, ii) it starts reporting its presence.

Conditional route reconstruction occurs when a node moves within the smart space, changing the network topology such that the node cannot reach RM via the same route, which prevents heartbeat message delivery to RM. This situation is depicted in Fig. 5. Eventually the node entry for the relocated node in RM times out and RM triggers route reconstruction by flooding SetRoute messages. Every addressed node reconstructs their route to the sender of the SetRoute message, which is RM. This could be more efficient than just sending a new subscription. A sequence diagram of RM periodic advertisement and route reconstruction is given in Fig. 6.

Fig. 5. Conditional Route Reconstruction

Discovery of Services and Resources

Discovery of services and resources is a one-time process: As long as the node is not dropped and rediscovered, the discovery of services and resources occurs once only and it is triggered upon detection and registration of a node in the smart space network. This results in energy savings since node information is not reported periodically. Ideally, RM should prompt for node information specifically from the newly joined node in unicast. However, this requires a known route from RM to the node which is possible if the node would subscribe to a service provided by the RM node. However, this is an assumption that may or may not hold depending on the application. Therefore, we take a different approach in our implementation: whenever a new node joins the network, RM requests node information from all nodes by flooding.

Although flooding is costly and limits scalability, in this way there is no need to construct routes from nodes to RM. One advantage of the flooding approach is that this enables RM to update its entries for nodes other than the newly joined node and to maintain the most recent information for all nodes.

Fig. 6. RM periodic advertisement and route reconstruction

The sequence diagram in Fig. 7 depicts the flow of the proposed smart space discovery mechanism. First, the node is discovered as it reports its heartbeat for the first time. Upon creation of a node entry in RM, the node information is prompted as a one-time subscription. HB nodes report their services and resources to RM and the related node entry is updated.

Fig. 7. Discovery of node and node information

Monitoring of Services and Resources

Service and resource monitoring is used for maintaining consistent information concerning the utilization of services/resources during run-time. Regarding smart space

nodes, we define three distinct types of information that a node holds which are relevant and necessary for monitoring:

Node Type and Services: This group of information refers to unchanging or slowly changing information over time. From a smart space perspective, a node is of a permanent type (e.g. capable of temperature monitoring) and possesses well defined services which are not likely to change in due course.

Subscriptions to Services: This group of information refers to dynamically changing service utilization on a node, i.e. subscriptions.

Resources: This information refers to dynamically changing resource utilization on a node, e.g. utilization of processing power, memory and battery. Utilization of resources is a key factor for assigning service requests to nodes.

RM Node Entries

The elements of a node entry object and their data types are shown in Fig. 8. *NodeID* is an integer value which uniquely identifies a node in the whole network. *Timeout* and *Counter* values keep precise timing. These three values are maintained by the heartbeat protocol. Node Type is an integer value which is uniquely identified throughout the network. *Services* and *Resources* are data structures representing further details about the set of services and the resources. The information regarding the subscriptions can be used by RM to determine the schedule on the node and to subsequently estimate the load on that node.

Fig. 8. Node Entry Object Overview

Reporting Intervals & Triggers

It is an essential design problem to decide what information needs to be monitored with what frequency or conditions. If the information on RM is outdated, user experience will be diminished due to wrong or late information and hence wrong decisions in management. On the other hand, if the information is reported too frequently, energy is wasted. The aim is to balance this tradeoff and achieve a system spending the least energy while providing required functionality.

Fig. 9 depicts the relationship between triggering conditions, types of information to be communicated and corresponding appropriate reporting periods. For services, reporting is done with low frequency, since node type and its installed services are assumed to be unchanging over time. Likewise, subscription information is reported with low frequency as well, since they rarely change beyond RM control. The Map Request trigger shown in Fig. 9 represents mapping of external requests to provide services to selected nodes which is carried out by RM. This causes changes to the subscription information, which needs to be reported.

On the other hand, node resources are reported at higher frequency as this information is rather dynamic. As an

example, the remaining energy of a node battery is ever changing and it is crucial information in order to take management decisions. Furthermore, whenever a node leaves the smart space network its duty will be redirected to other nodes. This decision is best made with fresh information regarding resources. The HeartBeat signal has to be reported regularly and with the highest frequency. Although it would reduce messaging overhead, performing a lookup of nodes whenever needed is not feasible for many applications since message losses would introduce unacceptable delays.

Fig. 9. Reporting conditions and periods

Management

In this work, management functionality is implemented at the level of simple service mapping, as a proof of concept. In principle, the management system could be further extended to a much more complex degree, since the node information is already stored in RM in a structured and consistent way.

We model an external service mapping request as a one-time subscription to the SmartSpaceSubscriber service of RM. Therefore, a service request could be loaded into the network (by any node, or by the loader) and be routed to RM (by content based addressing). Four parameters are needed in this subscription: the service requested, the subscriber node type, the provider node type and the period of subscription. The job of RM is to search for two nodes which would be fit to satisfy the service request: a provider and a subscriber. The RM node implements the following two system calls to select the 'best' candidates: *MatchProvider(provider_type, service_id)* and *MatchSubscriber(subscriber_type)*, either as system calls or as part of the OSAS program of RM. These methods first examine the nodes registered in the repository which match the requirements. Afterwards, they select the 'best' candidate among these. This selection procedure can be implemented using the already existing information regarding the utilization of the service (number of subscriptions) and the utilization of resources. In our implementation the selection is made according to the remaining energy levels of the nodes, i.e. the node with the most remaining energy is selected. This prolongs network lifetime if there are many candidates for providing requested services. In order to complete the service mapping, the selected subscriber needs to invoke the *SubscriptionHandler* of the selected provider, with the requested service and period. This would, however, require

that the nodes have a route to one another. This approach is not adopted by the OSAS routing protocol since the potential size of the routing tables is an issue.

In order to overcome this problem, we use another routing protocol, namely, path-based routing. Path based routing creates a hierarchical tree representation of the network on the root node. The process starts with the root node broadcasting a message stating that it is the parent of the receiving nodes. This message contains a phase number, stating the recentness of the initiation operation. The nodes which receive the message select the root node as their parent; and they broadcast themselves as being parent nodes. If a node has already selected a parent (in the same phase), it ignores the message. Thus, the selection process dissipates through the network like a wavefront. Afterwards, every node reports its presence to its parent and forwards reports of their children and grandchildren to its parent and appends its NodeID in front of the report message stack. The root node collects these messages and constructs a tree structure of nodes reporting to it. In a new initiation process, the phase number is increased so that the nodes know they have to reselect a parent for the new phase. Fig. 10 presents an example scenario of path based routing construction.

Fig. 10. Path Based Routing Initiation Example

The root node can send a message to every node by specifying the entire path in the message. For that it needs to store the entire tree. All other nodes simply store their parent. Since the root can be reached from every node, this scheme ensures node-to-node unicast communication, albeit in a suboptimal way. A message being sent is guaranteed to travel for at most two times the routing tree height. When the tree is balanced this scales logarithmically. However, the root is a ‘hot spot’ in the communication. We implemented path-based routing with RM being the root node. On top of that we implemented a function *SendToTarget*, that performs (application-level) routing as described above. As a result, the provider properly gives its service to the subscriber upon service mapping by RM.

The top level architecture diagram of the developed lightweight smart space solution with service, node and resource discovery/monitoring/management capabilities is depicted in Fig. 11.

Fig. 11. Lightweight smart space architecture

A smart space node has a smart space service unit which is composed of heartbeat, services monitor, subscriptions monitor and resources monitor. The heartbeat service is used for discovery purposes. Introspection system calls query and report information regarding node services, subscriptions to these services and resources.

The entire behavior of RM and other nodes is implemented in WaSCoL. The simulator is written in Python; for each function and service of RM we have the option to implement it either in Python or in WaSCoL. We implemented some of them in Python in order to have some nicer user interfacing.

V. EXPERIMENTAL RESULTS

In this section, experiments with the proposed system in the OSAS simulator and on physical nodes are presented, using a simple scenario. The implementation footprint is analyzed in terms of memory allocation (i.e. RAM and ROM usage) and messaging overhead (i.e. message size and number).

Simulation Environment

The OSAS simulation environment has been used for prototyping and testing the implementation during the development phase. The simulator mimics a given network scenario with defined number and placement of simulated nodes in a virtual space. Moreover, it provides a Python environment to define nodes of different types (i.e. with different sensors) and functionalities for testing. System calls and event handler functions can be implemented in Python more easily than in WaSCoL because of the more advanced language constructs. Nodes specified in Python are fully compatible with the system architecture as they are programmed by compiled bytecode of a network program file. They mimic real nodes in behavior and use the same messaging protocol. It is possible to program a network consisting of both simulated and physical nodes interacting with one another. Simulated nodes are indistinguishable from the physical nodes in terms of their behavior.

The OSAS simulator provides a graphical user interface with a tabbed view pane. The *Smart Space* tab provides a 2D bird-sight view of the simulation space. Types and locations of the simulated nodes are determined by the scenario description file. This tab is interactive, i.e. nodes can be dragged within the canvas space by a mouse, reflecting

changes in the network topology. A textbox is used for printing data by invoking *PrintHandler* of a simulated node (Notice that in this way *each node* has a channel to output some message.) RM and other smart space nodes can be visually identified and nodes participating in the network can be distinguished from the ones that are not participating. Nodes joining and leaving the smart space can be viewed in real-time, in addition to the contents of the RM repository.

Fig. 12 demonstrates a screenshot of the Smart Space Tab, running an example scenario, which there are six nodes simulated and displayed on the grid, with randomly assigned NodeIDs. The nodes can be clearly distinguished by their shapes and colors. RM (node 45) is blue round shaped, whereas the nodes that are in the network and nodes that are out of the network range appear as green triangles and gray squares, respectively. The nodes 35, 38, 41 and 43 are in the smart space network. The signaling distance is up to four squares on the grid; consequently node 32 is out of the network range. On the right side of the view, there are two text boxes. Both of these boxes written by RM where the top box prints changes to the smart space network over time and the bottom box prints the present state of the RM repository.

Fig. 12. OSAS Simulator screenshot running a sample scenario

The RM repository log (top text box) shows that all five nodes join the network (PUT) as the simulator starts. Following that, the content nodes of the repository are printed. Afterwards, node 32 moves out of the vicinity of the network and becomes unreachable; therefore it is dropped (DEL 32) out of the smart space network. The contents of the repository after the drop are printed, representing the latest state. Then, we see that a service request is made on the system. This is done manually by the smart space user, by subscribing to RM's "smart space subscription request service" via the loader. In the loader display, subscriptions defined are listed and they can be loaded to the network on demand. Upon the

service request, RM matches a provider (node 38) and a subscriber node (node 41) and maps the service accordingly. Node 41 starts receiving temperature measurements and prints received values (not visible in the figure). Finally, latest repository contents are printed again. The text box in the bottom displays detailed information about nodes in the smart space though not all of them are visible in Fig. 12. The display format for a node is shown in Fig. 13.

Fig. 13. Node information on the Smart Space Tab

All the nodes in the network, of course, have the *HeartBeatService* installed and the RM (node 45) has subscribed to this service on all nodes with a 1 sec period. In this scenario, there are two types of HB nodes: gateway and temperature node. Nodes 32, 35 and 38 are temperature nodes, and nodes 41 and 43 are gateway nodes.

TABLE I
ASSIGNED TIMING PARAMETERS IN EXAMPLE SCENARIO

Parameter	Value
HeartBeat Period	1 second
Timeout Period	6 seconds
Route Fix Trigger	3 seconds to Timeout
Max. Route Fix Frequency	Every 3 seconds
Refresh Registry Period	1 second
Advertisement Period	10 seconds
LowFrequencySubscription Period	60 minutes
HighFrequencySubscription Period	30 seconds
Triggering Conditions Check/Response Period	1 second

Temperature nodes have temperature sensing capability and consequently a Temp service installed on them which reports the temperature measurements to the subscribers. Temperature nodes only have an "energy" resource which indicates the energy remaining on the node battery. Gateway nodes collect readings from temperature nodes and have "energy", "CPU" and "RAM" resources. The measurements method of these values are beyond the scope of this paper. The model used in this example is that the energy resource decreases in a steady fashion (different per node) and the other resources (i.e. CPU and RAM) have constant values. In this example, a service mapping is made in between nodes 38 and 41, which can be seen in the top textbox. We can see that node 38's Temp service is subscribed by node 41 with a period of 5 seconds. Timing parameters in the simulation run are as in TABLE I.

This scenario runs stably in the simulator with the above parameters. The number of messages *periodically* exchanged per node per minute, is shown in TABLE II. Disregarding

triggering conditions, every node in the smart space network sends and receives a total of about 70 messages per minute. This count can become problematic with larger numbers of nodes, but from the method it is clear how to trade accuracy for message volume. The number of messages exchanged for various triggering conditions is shown in TABLE III.

TABLE II
MESSAGE COUNT OF REGULAR INTERVAL MESSAGES

<i>Sent Messages</i>	<i>Received Messages</i>
HeartBeat: 60	Advertisement: 6
Report Resources: 2	Resource Report Subscription: 2
Report Services: 1/60	Services Report Subscription: 1/60
Report Subscriptions: 1/60	Subscriptions Report Subscription: 1/60
Total: ~ 62	Total: ~ 8

TABLE III
MESSAGE COUNT OF TRIGGERING CONDITIONS MESSAGES

	<i>Messages Sent</i>	<i>Messages Received</i>
Node Entry	3	3
Set Route	1	1
Node Departure	1	1
Service Mapping	2	2

With the parameter settings of TABLE I, the majority of message exchanges is made up of heartbeat messages only; especially if the smart space network is not active (i.e. no or few triggering conditions happen every minute). The messages exchange frequency (all types of messages) in the smart space network versus the heartbeat period is depicted in Fig. 14. In order to avoid unnecessary messaging overheads, the heartbeat period needs to be fine tuned to the specific service/user needs.

Fig. 14. Heartbeat period vs. network message traffic

Deployment on Physical Nodes

The proposed system has been successfully tested on physical nodes as well. The current implementation runs on top of the TinyOS [17] operating system on node hardware as well as on MANTIS. OSAS footprint on nodes without the smart space implementation is shown in TABLE IV. OSAS also uses RAM dynamically: 160 bytes for execution stack and 132 bytes for message buffers.

TABLE IV
FOOTPRINT OF TINYOS AND OSAS

	<i>ROM (in bytes)</i>	<i>RAM (in bytes)</i>
TinyOS (w/o OSAS)	15912	947
TinyOS (w/ OSAS)	28068	1289

The memory requirement from a low capacity node for the smart space application can be expressed as the sum of the service memory requirements stated in TABLE V.

TABLE V
MEMORY ALLOCATION OF SMART SPACE APPLICATION

<i>Service</i>	<i>Size (in bytes)</i>
HeartBeatService	28 (14 ROM + 14 RAM)
ReportServicesService	30 (16 ROM + 14 RAM)
ReportResourcesService	30 (16 ROM + 14 RAM)
ReportSubscriptionsService	30 (16 ROM + 14 RAM)
Total	118 (62 ROM + 56 RAM)

The smart space application is quite lightweight, even when compared to some WSN applications given that we have the run-time installed. For example, a considerably complicated healthcare application results in approximately 250 bytes of code for services.

The message payload sizes with 8-bit encoding for the smart space application are presented in TABLE VI. The MAC protocol and OSAS messaging protocol occupy 10 and 5 bytes, respectively, in the message header.

TABLE VI
MESSAGE PAYLOAD SIZES

<i>Message</i>	<i>Size (in bytes)</i>
HeartBeat Report	2
HeartBeat Subscription	17
Services Report	3 + (# services)
Resources Report	2 + 2 * (# resources)
Subscriptions Report	2 + 4 * (# subscriptions)
Services Report Subscription	17
Resources Report Subscription	17
Subscriptions Report Subscription	17
Service Mapping	9
Service Request Subscription	17

RM functionality could also be mapped fully to WaSCoL with some system calls (which are in fact library extensions to the run-time system) supporting the more complicated operations. However, the resource needs of RM are proportional to the network size. This makes it less amenable for running on a low resource node. In order to address this issue we would need to make RM distributed and hierarchical.

VI. CONCLUSION AND FUTURE WORK

The main contribution of this work is a lightweight and scalable smart space architecture composed of low capacity nodes (e.g. wireless sensors) and a Resource Manager (RM), as an extension of the Open Service Architecture for Sensors (OSAS). The proposed smart space architecture truly integrates low-capacity devices by allowing service sharing, re-programming and re-configuration by new nodes thus admitting the dynamic blending in of these new nodes. Our smart space implementation offers means to discover, monitor and manage smart nodes, their services and their resources via an RM node, which is implemented as a simulated node. The system has been successfully tested on low-capacity nodes as well as simulated nodes that function in the IP infrastructure.

The system utilizes a binary soft-state protocol, i.e. the heartbeat protocol, and periodic RM advertisements for service discovery and monitoring. Three types of information are monitored: *i*) node type and services, *ii*) subscriptions to services and *iii*) resources. For management, service mapping that makes use of discovery and monitoring information is implemented as a simple application. Path-based routing protocol tables are stored on RM to draw the responsibility of routing to a higher capacity central node rather than the low capacity nodes. A sample scenario has been experimented with, footprint values and messaging overheads were shown to be minimal, i.e. comparable to wireless sensor networks.

The proposed solution is centralized. For avoiding a single point of failure and load balancing among multiple RMs, a hierarchically distributed approach is considered. Message sending periods and conditions can be automatically adjusted by the nodes according to their type and run-time situation, instead of being determined by RM. Energy expenditure can be minimized by automatically adjusting timing parameters to the dynamicity of the network. The proposed management mechanism is very basic and could be improved with service recovery, complex service mapping and advanced arbitration for minimal resource usage.

REFERENCES

- [1] Marples, D.; Kriens, P.; "The Open Services Gateway Initiative: An Introductory Overview," *Commun. Mag., IEEE*, vol. 40, no. 4, pp. 110-114, Dec. 2001.
- [2] Kronlof, K.; Kontinen, S.; Oliver, I.; Eriksson, T.; "A Method for Mobile Terminal Platform Architecture Development," in *Advances in Design and Specification Languages for Embedded Systems*, pp. 285 – 300, DOI - 10.1007/978-1-4020-6149-3 17. Springer Netherlands, 2007.
- [3] Bosman, R.; Lukkien, J.; Verhoeven, R.; "An Integral Approach to Programming Sensor Networks," *Consumer Comm. and Networking Conf.*, 2009. CCNC 2009. 6th IEEE, vol., no., pp.1-5, 10-13 Jan. 2009.
- [4] Brodt, A.; Sathish, S.; "Together we are strong," *IEEE Conference on Pervasive Computing and Communications (PerCom)*, pp.1-4, 2009.
- [5] Bardram, J. E.; "The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications". Technical Report CIPC Technical Report 2004–PB–61, Centre for Pervasive Computing, Aarhus, Denmark, 2003.
- [6] Arnold, K.; O'Sullivan, B.; Scheifler, R.W.; Waldo, J.; Wollrath, A.; "The Jini Specification," Addison-Wesley, 1999.
- [7] Gu, T.; Pung, H.K.; Zhang, D.Q.; "A service-oriented middleware for building context-aware services"; *J. Network and Computer Applications*, vol.28, pp.1–18; 2005.
- [8] Van der Meer, S.; O'Connor, R.; Davy, A.; "Ubiquitous Smart Space Management"; *International Workshop on Managing Ubiquitous Communications and Services (MUCS)*, M-Zones whitepaper; 2003.
- [9] Henriksen, K.; Indulska, J.; "A Software Engineering Framework for Context-Aware Pervasive Computing"; *Second IEEE Conf. on Pervasive Computing and Comm.*, IEEE Computer Society, pp. 77–86; 2004.
- [10] Kagal, L.; Korolev, V.; Chen, H.; Joshi, A.; Finin, T.; "Centaurus: A Framework for Intelligent Services in a Mobile Environment"; *Conf. on Dist. Computing Systems Workshops (ICDCSW '01)*, pp.0195; 2001.
- [11] Efstratiou, C.; Cheverst, K.; Davies, N.; Friday, A.; "An architecture for the effective support of adaptive context-aware applications"; *Mobile Data Management (MDM)*, Hong Kong, Springer, pp. 15–26; 2001.
- [12] Chen, H.; Finin, T.; Joshi, A.; "An ontology for context-aware pervasive computing environments"; *The Knowledge Engineering Review*, Volume 18, Issue 03; 2003.
- [13] Baldauf, M.; Dustdar, S.; Rosenberg F.; "A survey on context-aware systems"; *Int. J. Ad Hoc and Ubiquitous Comp.*, 2(4):263-277; 2007.
- [14] Helal, S.; "Standards For Service Discovery And Delivery"; *IEEE Pervasive Computing*, 1:95.100; July-Sept 2002.
- [15] Bhardwaj, S.; Ozcelebi, T.; Lukkien, J.; "Smart Lighting Using LED Luminaries" to appear, *IEEE PerCom 2010, SmartE Workshop*, Mannheim, Germany, March 2010.
- [16] Tjong, M.; Lukkien J.; "An Investigation Into Soft-state Protocol Parameters," *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, July 14, 2008.
- [17] Levis, P.; Madden, S.; Polastre, J.; Szewczyk, R.; Whitehouse, K.; Woo, A.; Gay, D.; Hill, J.; Welsh, M.; Brewer, E.; Culler, D.; "TinyOS: An operating system for wireless sensor networks," *Ambient Intelligence*, Springer-Verlag; 2005.

BIOGRAPHIES

Tanır Özçelebi was born in Konya, Turkey, in 1980. He received the B.Sc. degree in electrical and electronics engineering from Bilkent University, Ankara, in 2002 and the Ph.D. degree in electrical engineering from Koc University, Istanbul, in 2006. In 2006, he joined the System Architecture and Networking Research group at Eindhoven University of Technology as a postdoctoral researcher, an became an assistant professor in the same research group in 2008. His research interests are resource and service discovery and management, quality of service management for networked systems and design of smart distributed systems.

Johan J. Lukkien is head of the System Architecture and Networking Research group at Eindhoven University of Technology since 2002. He received M.Sc. and Ph.D. from Groningen University in the Netherlands. In 1991 he joined Eindhoven University after a two years leave at the California Institute of Technology. His research interests include the design and performance analysis of parallel and distributed systems. Until 2000 he was involved in large-scale simulations in physics and chemistry. Since 2000, his research focus has shifted to the application domain of networked resource-constrained embedded systems. Contributions of the SAN group are in the area of component-based middleware for resource-constrained devices, distributed coordination, Quality of Service in networked systems and schedulability analysis in real-time systems.

Remi Bosman was born on the 15th of June in Kapelle Biezelinge, the Netherlands. He received his Master degree (cum laude) in Technical Computer Science at the Eindhoven University of Technology in 2005 on the topic of approximate pattern matching algorithms in strings. He is currently a PhD student within the Software Architecture and Networking group at the Eindhoven University of Technology. During his PhD project he has been researching coordination within service oriented architectures with a specific focus on sensor networks. His research interests are component technology, programming languages and sensor networks.

Önder Uzun was a member of the System Architecture and Networking Research group at Eindhoven University of Technology in year 2009 as an MSc student. His master thesis is on service discovery, monitoring and management in smart spaces composed of very low capacity devices.