Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2012, Article ID 324935, 15 pages
doi:10.1155/2012/324935

Research Article

Multiagent-Based Data Fusion in Environmental Monitoring Networks

Samuel Dauwe,[1] Timothy Van Renterghem,[2] Dick Botteldooren,[2] and Bart Dhoedt[1]

[1] *IBCN-IBBT, Department of Information Technology, Ghent University, Gaston Crommenlaan 8, Bus 201, 9050 Ghent, Belgium*
[2] *Acoustics Group, Department of Information Technology, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium*

Correspondence should be addressed to Samuel Dauwe, samuel.dauwe@intec.ugent.be

Received 24 November 2011; Revised 22 March 2012; Accepted 6 April 2012

Academic Editor: Donggang Liu

Advances in embedded systems and mobile communication have led to the emergence of smaller, cheaper, and more intelligent sensing units. As of today, these devices have been used in many sensor network applications focused at monitoring environmental parameters in areas with relative large geographical extent. However, in many of these applications, management is often centralized and hierarchical. This approach imposes some major challenges in the context of large-scale and highly distributed sensor networks. In this paper, we present a multilayered, middleware platform for sensor networks offering transparent data aggregation, control, and management mechanisms to the application developer. Furthermore, we propose the use of multiagent systems (MASs) to create a computing environment capable of managing and optimizing tasks autonomously. In order to ensure the scalability of the distributed data fusion, we propose a three-step procedure to balance the workload among machines using mobile agent technology.

1. Introduction

Over the years, sensor networks (SNs) have gained a lot of attention in both public and research communities. Advances in embedded systems and mobile communication have led to the emergence of small, energy-efficient, low-cost wireless sensing units. This technology makes it possible to deploy a large number of sensors in a widespread geographical area forming a self-organizing and self-healing wireless sensor network (WSN). Examples of application domains include logistics and transportation, agriculture, health care, and environment.

Despite the technological advances, individual sensor nodes have still limited processing, storage, and communication capabilities. Furthermore, wireless sensor networks are typically used in highly dynamic environments, and therefore, they must be tolerant to hardware failures and communication problems of individual nodes. Consequently, software developers have to deal with many complex low-level system issues as well as with the design of protocols facilitating the communication in error-prone sensor networks [1]. Sensor middleware [2] is exactly addressing this matter, abstracting the internal operation and heterogeneity of the sensor network, and offering standard interfaces, suitable abstractions, and services to the developer.

As of today, most of the environmental sensing applications follow the traditional client/server architecture. In such a system, data is collected in a (possibly distributed) database, and processing is offered by a separate cloud infrastructure, offering the required scaling and cost-effectiveness. However, WSN management in such a centralized and hierarchical approach is a real challenge. System size, hardware heterogeneity, dynamic environment, and the volume of data contribute to the complexity. In our view, a multiagent systems (MASs) approach is much better suited in an environment which must be self-configuring, self-healing, and self-optimizing. According to the software agent paradigm [3], agents are autonomous problem solvers that cooperate to achieve the overall goals of the system. Furthermore, they have the ability to physically migrate from one device to another including dynamic behaviour, actual state, and specific knowledge. Because of the decentralized nature, no single point of failure exists, and the solution is scalable with respect to the number of devices in the network.

In this paper, we present a middleware platform that addresses several key issues in modern sensor networks such as autonomy, scalability, and adaptability. Operating and networking details are hidden from applications, by adopting a multilayered software architecture. In order to handle the scalability and reliability challenges of the distributed data fusion, we propose to use a multiagent-based collaborative information processing.

The remainder of this paper is structured as follows. Section 2 presents related work in the context of managing complexity in WSN environments. We also refer to existing multiagent approaches to achieve an intelligent and adaptive sensor network. In Section 3, we give an overview of the important challenges related to modern monitoring applications by the discussion of a use case focused at building an extensive multisensor urban measurement network for noise and air pollution. In this section, we also present the designed architecture which supports straightforward measuring tasks, but also acts as a platform hosting more complex functionalities. Implementation details are discussed in Section 4 followed by the description of a three-step procedure to balance the workload using mobile agents in Section 5. The evaluation results of the currently deployed prototype are presented in Section 6. Conclusions and ideas for future research are given in Section 7.

2. Related Work

Recently, a number of lightweight component models have been proposed as a promising approach to managing complexity in WSN environments. Middleware platforms such as RUNES [4], OSGi [5], and LooCI [6] enable the dynamic deployment and rewiring of components to support adaptation and reconfiguration.

The RUNES approach to middleware provision is to offer an adaptive middleware platform based on a two-level architecture. The foundation is a language-independent, component-based programming model that is sufficiently small to run on any of the devices typically found in networked embedded environments. A second software component layer offers the necessary functionality to configure, deploy, and dynamically reconfigure both system and application level software. The loosely-coupled component infrastructure (LooCI) introduces a novel event-based component and binding model for networked embedded systems. The LooCI middleware is designed for Java devices that support standard Java ME [7], aiming for maintaining a minimal memory footprint and offering good performance.

Over the years, there is a growing interest in the integration of multi-agent systems (MASs) into sensor networks, because of their intelligence and adaptation to the field. In this context, agents can be seen as entities responsible for executing tasks such as incoming measurement validation, custom alarm identification, data processing, and management functions. The BiSNET middleware platform [8], for example, hides low-level operating and networking details from applications and implements a series of mechanisms to support autonomous, scalable, adaptive, and self-healing applications. Agents on the sensor node are responsible for autonomously increasing power efficiency, collectively self-heal (i.e., detect and eliminate) faulty data, aggregating data, and react to environmental changes. Biswas et al. [9] propose the use of mobile agents for scalable and energy-efficient data aggregation. In such an approach, software code can migrate from node to node performing data processing autonomously.

Existing frameworks for environmental monitoring typically offer basic measurement services but fall short to offer accompanying processing facilities. The main contribution of this paper is presenting a platform enabling a tight coupling between the data collection and data processing functionalities. By adopting this approach, several optimisation opportunities are possible such as dynamic filtering on the sensor nodes, distributed cooperative data fusion, and adaptive deployment strategies.

3. Autonomous Environmental Monitoring

As of today, numerous platforms [10–12] have been developed for environmental measurement and processing applications. We have chosen to elucidate the requirements of such modern sensor networks by presenting an application in this domain, focusing on monitoring, and analysing data related to noise and air pollution in an urban environment. As will be shown, these networks have stringent requirements in terms of bandwidth and deadline-sensitive data processing.

In this section, we describe the actual use case, leading to a summary of more generic challenges and possible solution approaches for these demanding sensor applications.

3.1. Use Case. Urban environments are typically characterized by busy motorized traffic, leading to elevated noise levels and high concentrations of airborne pollutants. As of today, there is a specific interest in ultrafine particles (UFPs) that could potentially be more harmful to health than the coarser fraction more regularly measured (PM10, PM2.5).

Both noise and airborne pollutants share the same dominant source in an urban environment, namely road traffic. It was shown in [13] that UFP and specific noise indicators can be reasonably well correlated and further improved by including some basic meteorological observations. As a result, one of the approaches followed is using a large number of microphones as proxies for UFP sensors. As discussed in [13], a few expensive UFP monitoring stations might be needed in the network, to further adapt such correlations, depending on specific conditions. However, deploying a dense urban network of high-cost UFP sensor devices and high-quality microphones is often beyond the budget of city authorities. A solution to this problem is to deploy a limited number of expensive (and high-precision) sensors together with a large number of low-cost devices (offering lower data quality). The high-precision nodes then serve to calibrate the less expensive devices, while the latter are able to monitor spatial variation. It was shown in [14] that some low-cost microphones (used in consumer electronics) can be accurate for environmental noise monitoring.

Obtaining high-quality data from this type of sensor network is more complex than in the case where a dense network of expensive devices were available. Typically, advanced calibration techniques and interpolation schemes (e.g., for translating noise data to air pollutant concentrations) are required, leading to the need for complex data processing in the sensor network itself. Furthermore, centralized control in such a dense and complex network becomes unfeasible. The systems must become autonomous, having the possibility of automatic self-adaptation to changes in the environment.

3.2. Design Principles. One of the goals of our middleware platform is to ease the construction of environmental monitoring applications by hiding as much as possible the low-level complexities commonly found in modern sensor networks. Therefore, we designed a component-based service-oriented architecture allowing the dynamic rewiring and reconfiguration of its features and functioning.

3.2.1. Extensible Software Architecture. Due to the hardware restrictions and the heterogeneity typically encountered in sensor devices, deployment and configuration of sensor nodes is a challenge. We have developed a component-based service-oriented architecture where the functionality is contained in pluggable modules, as presented in Figure 1. Using this approach, the sensing application can be easily adapted to the underlying hardware. Furthermore, components can be deployed and removed at run time. By taking intelligent decisions about the deployment of modules, resource usage can be optimized (e.g., replacement of preprocessing logic).

The main functions of each layer can be summarized as follows.

(i) The *sensor layer* consists of sensor nodes and sensors. This layer is responsible for controlling the sensors, gathering sensory information, performing data preprocessing, and data communicating with the back-end infrastructure.

(ii) The *virtualization layer* hides the heterogeneity of the actual hardware, providing a uniform interface to the upper layers. This layer is also responsible for checking the reliability of the sensor data. Special validation agents compare the received measurements with historical data and measurements from neighbor sensors. Another responsibility is management of sensor assets. To detect problems related to sensor nodes, control agents continuously monitor the state of a sensor node (i.e., heartbeat information).

(iii) The *task distribution layer* manages the data processing functions. Jobs are planned on nodes in the network and executed by software agents.

(iv) Various types of sensors exist, which can deliver different types of information, such as chemical analyses of gases, weather data, images, and audio files, and so forth. Furthermore, data can be generated through algorithms or models. The purpose of the *storage layer* is to hide the details of where and how the data is stored.

(v) The *interface layer* enables the web-based discovery, exchange, and processing of sensor data, as well as the tasking of sensor systems in a standardized manner. This layer provides the interoperability with third-party systems.

3.2.2. Self-Configuring. An important characteristic of wireless sensor networks is that they are highly dynamic. As a consequence, measurement and processing nodes should discover each other and work together automatically. Therefore, we implemented a discovery mechanism in the several layers of the sensing application. On the sensor node, a module can check the availability of other components and adapt accordingly. The virtualization layer on the other hand holds a registry containing all the dynamic information (e.g., IP address) needed to discover and contact sensor nodes.

3.2.3. Adaptability. In a traditional client/server-based computing paradigm, sensor nodes send raw sensory data to a back-end processing center for data fusion. In the case of high-bandwidth sensors, the amount of data is extremely high, and a local processing has to be performed before transmission. The data fusion executed on the individual sensor node results in a lower communication bandwidth, increased scalability, and reliability.

We propose an agent-based control of the data fusion and transmission, allowing an autoadaptation of each sensor node to changes and failures. Several fault scenarios were identified with possible solution approaches. These mechanisms were translated to agent logic, helping to remove possible bottlenecks and enhancing the systems efficiency. Examples include failing communication links, special events in the measured phenomenon, and failing of sensors and intermediary nodes.

3.2.4. Autonomic Distributed Data Fusion. We designed an architecture for an agent-based task distribution system, which is presented in Figure 2. The idea is that each task is assigned to a task force, which is a group of agents, logically grouped for executing a job. This group can be spread over multiple physical machines and has one agent that acts as the manager. A task is assigned to a task force by the task handler; this component has also the responsibility of deciding which agents will be responsible for executing a job and where in the network they will run. The task handler should not be regarded as a central component; there exist multiple instances of the component within the network, and decisions are made on the basis of information available in the registries, which are described in this section.

The responsibility of the *task handler* is accepting job requests and assigning them to agents. When this component receives a task, it first looks up which agents can handle this task. Next step is the resource allocation, that is, assigning agents to a way network utilization is minimized and the computational resources are efficiently used. The *task handler* relies on the resource, agent and task registries for its execution. Multiple *task handler* instances exist in the system to ensure scalability and reliability.

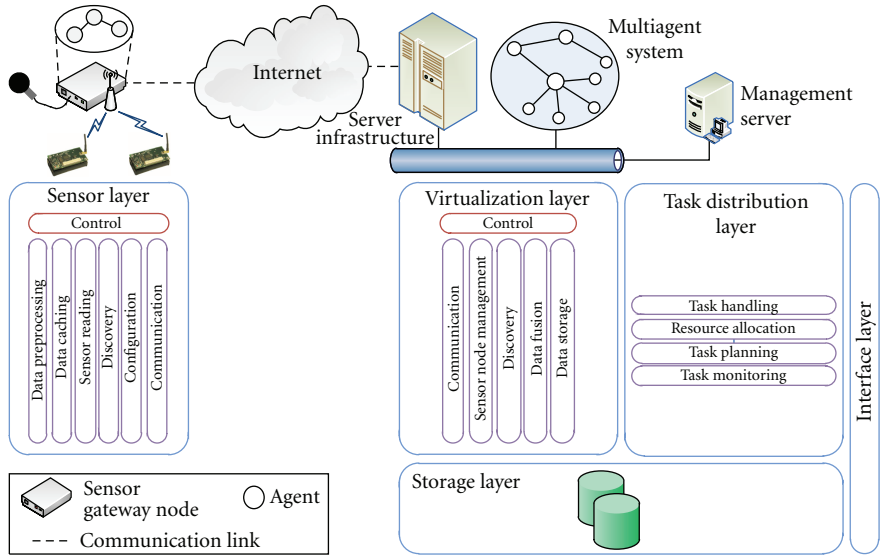

Figure 1: Overview of the different layers in the environmental monitoring platform. High-level components can be grouped in these layers according to their functionality, such as the sensing, transmitting, storage, and processing of sensor data. Each layer provides a uniform interface, hiding the complexity.

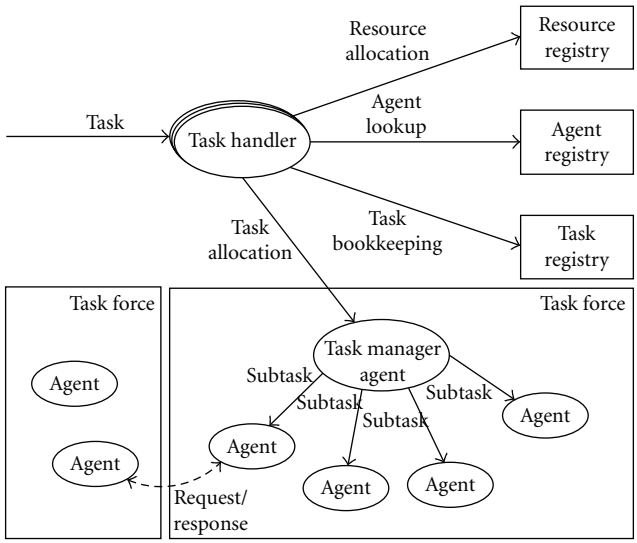

Figure 2: Architecture of the agent-based task distribution system used in the context of long-running data-driven applications. A submitted job is handled by the *task handler* which allocates the task to a *task force* based on the information available in the registry databases.

Processing nodes can range from mobile devices (e.g., smartphones) to powerful back-end servers. As a consequence, the *task handler* needs an overview of all nodes where agents can be allocated to. This application, in contrast to general theory, has a very intensive interaction with the data, for example, calculation of a one-hour mean involves only a condition (is the quality of the data good) and a simple (multiply and) accumulate for each second of data that is retrieved from the database. Thus, there is a strong urge to have the agents work on a resource that has a short distance in time to the data.

The *resource registry* contains two types of information. The first type is the static information which does not have to be updated often. Examples are operating system, installed libraries, total amount of main memory, the number of CPUs, maximum bandwidth, location, IP address, and so forth. The second type of information is updated frequently and represents the current state of the node, for example, comprising average CPU load, idle time, free memory size, free disk space, and available bandwidth. During the resource allocation phase, the task handler checks if the requirements of an agent can be satisfied. Such constraints are specified using description files. If a node does not fulfil all requirements (e.g., size of main memory or free disk space), it will not be used.

The *agent registry* acts as an agent information store. It contains information like the resource requirements and file locations (e.g., scripts, executables). It also provides a mapping from task name to agent identifiers. When the task handler receives a job, it first looks up the task identifier in this registry to check if it is possible to handle the request or not. Only registered jobs with registered job identifiers are accepted for allocation.

When a task is accepted, planned and resources are allocated, information has to be stored giving a consistent and global overview of the multiagent systems. First of all, we have to keep track of all running jobs, parameters, their start dates, end dates, current state, and assigned resources. This state is important to detect failures; if a task stopped unexpectedly or missed its deadline, it has to be deleted and rescheduled. The *task registry* also gives an overview of all deployed agents. It holds the state of an agent (e.g., running or waiting for input), but also the resources used by the agent. This last type of information is especially useful for detecting

bottlenecks or performance problems; it also can trigger a relocation process from an agent to another.

The *task force* is a logical group of cooperating agents responsible for executing a job, consisting of a *task manager Agent* and multiple slave agents. The *task manager agent* acts as a mediator for the group; it receives the tasks, splits it up in subtasks, creates a workflow, and assigns subtasks to the subordinates. This entity is also responsible for monitoring the task progression and is responsible for resolving issues locally. A possible scenario is the removal and recreation of an agent that is blocked for a longer time. When no suitable solution can be found, the *agent manager agent* can decide to remove the *task force* from the system and resubmit the assigned task to the *task handler Agent*.

4. Database Model and Technology Choices

4.1. Open Standards. Due to the large number of sensor manufacturers and differing accompanying protocols, integration and discovery of sensor data is not straightforward. Lee et al. [15] argue the need for open, standardized sensor interfaces and sensor data formats to effectively integrate, access, fuse, and use sensor-derived data.

OGC's Sensor Web Enablement (SWE) [16] initiative aims at standardizing the entire sensor web process to bring sensor resources on the web. In order to manage the heterogeneity of the sensor resources and make them available on the application level, our monitoring platform has been made compliant with the OGC's SWE standards for web-based discovery, subscription, publishing, and alerting.

4.2. Database Model. The OGC observations and measurements standard (O&M) [17] defines a domain-independent, conceptual model for the representation of (spatiotemporal) measurement data. The open-source initiative 52 North [18] has provided a reference implementation for the sensor observation service (SOS) [19], using O&M to deal with measurements in a standardized way. At the database level, 22 tables are used with a lot of constraints and triggers. In our prototype, we implemented the subset shown in Figure 3. According to the OGC O&M standard, the elements in the database model are described as follows.

(i) *Feature of interest* refers to the real-world object to which the observation belongs (e.g., City of Ghent). In the context of our use case, we regard sensor nodes as features of interest.

(ii) *Procedure* points to the method, algorithm, or instrument used to generate the result.

(iii) *Phenomenon* is a property of a feature of interest (e.g., temperature).

(iv) *Observation* is defined as an act associated with a discrete time instant or period through which a number, term, or other symbol is assigned to a phenomenon.

The sensor alert service (SAS) [20] plays an important role for sending event-triggered alerts, for example, in case of threshold transgression. To be able to store SAS-compliant alerts, we extended the database model was extended with an alert table.

4.3. Open Services Gateway Initiative. The OSGi (Open Services Gateway initiative) component model provides a secure execution environment, support for run time reconfiguration, lifecycle, management, and various system services. OSGi targets powerful embedded devices such as smart phones and network gateways along with desktop and enterprise computers.

The OSGi Framework was chosen as dynamic module system and service platform for the Java programming language. OSGi enables the creation of highly cohesive, loosely coupled modules (also known as OSGi bundles) that can be composed into larger applications and managed remotely. Furthermore, each module can be individually developed, tested, deployed, updated, and managed with minimal or no impact to the other modules.

4.4. Agent Framework. JADE (Java Agent DEvelopment Framework) [21] was chosen as the software framework for developing interoperable intelligent multiagent systems. The JADE run time provides the basic services (i.e., dynamic discovery, asynchronous peer-to-peer communication, naming, etc.) for distributed peer-to-peer applications in both wired and mobile environments. Furthermore, JADE is compliant with the FIPA [22] specifications that enable end-to-end interoperability between agents of different agent platforms.

The JADE architecture is completely modular and, by activating certain modules, it is possible to execute the run time on a wide class of devices ranging from complex server-side infrastructures to mobile devices. An example of such a module is LEAP, which enables the division of the JADE container into a front-end running on the mobile device and a back-end running in the wired network. As a consequence, the front end becomes lightweight because part of the functionality of the container is delegated to the back end. Furthermore, JADE provides a homogeneous set of APIs that are independent from the underlying network and Java version.

4.5. Target Computing Platforms. OSGi is best suited for powerful embedded devices, because the smallest standard implementation, Concierge [23], consumes more than 80 kB. An OSGi Profile for Embedded Devices (OSGi ME) [24] exists, but the minimal memory usage is approximately 40 kB. The LooCI middleware on the other hand is designed for more constrained devices supporting Java ME, such as the Sun SPOT hardware [25]. The LooCI component model has a memory footprint of just 20.8 kB [6].

As of today, considerable success has been encountered in porting multiagent technology on mobile devices such as smart phones and battery-powered wireless sensor nodes. Recent developments of mobile multiagent systems include AFME [26], JADE-LEAP [27], micro-JIAC [28], and MobileC [29].

With the choice of OSGi as component model, and the JADE run time for multi-agent support, we target powerful

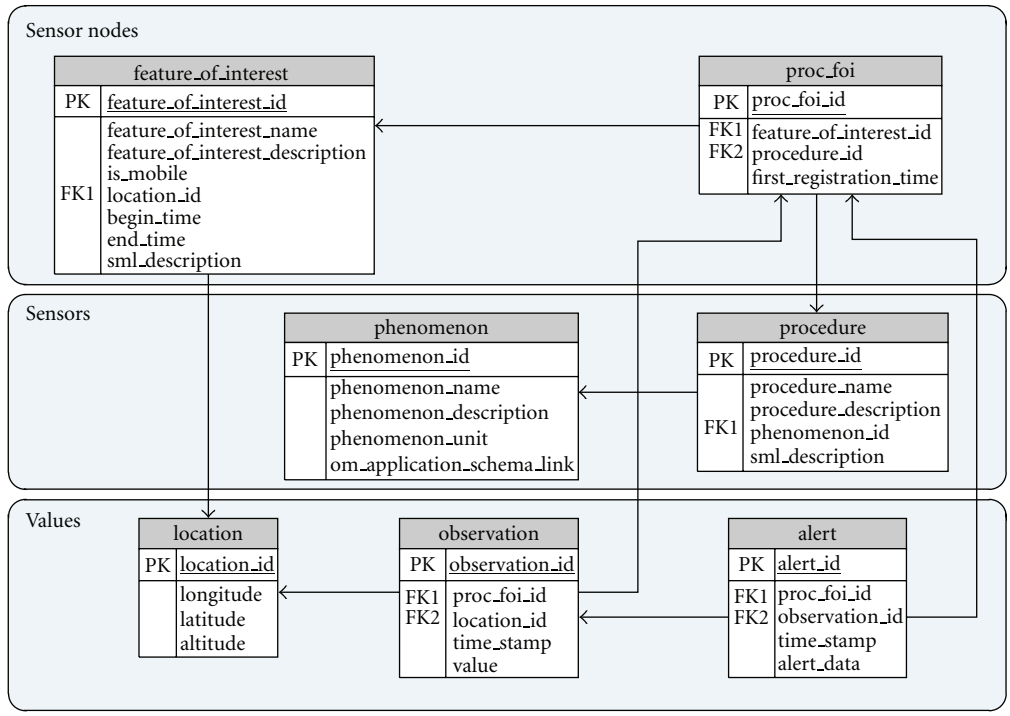

Figure 3: Relational schema of the database structure for storing sensor measurements and alerts.

sensor gateway nodes. However, with these choices, there is still a possibility to integrate sensor nodes that have strict constraints in terms of processing capabilities, memory, and communication abilities. In our proposed middleware architecture as presented in Section 3.2, the sensor node gateway can act as a data sink and controller to one or more resource-constrained sensor nodes. A similar approach was presented in [30] where a network of ad hoc ZigBee-enabled devices are dynamically integrated into an OSGi-based home gateway.

5. Autonomic Sensor Data Processing

Autonomous agents may consume and share resources in an unpredictable manner. Therefore, there is a need for dynamic and adaptive load-balancing strategies which allow to respond to changes (i.e., resource consumption behaviour of single agents). Possible mechanisms to deal with this decreased performance include agent cloning, migration, and task passing. Such load-balancing processes could either be handled by each individual agent or by special manager agents.

In the past, several approaches [31, 32] have been studied to use mobile agents for dynamic resource sharing. Jan Stender et al. [31] argue the need to introduce a load balancing agent (LBA) which decides when load-balancing is necessary and coordinates the process. In the presented paper, only agent migration was considered. We improved the algorithms to make use of agent cloning and taskdivision. In our approach, a load balancing procedure is started when one of the following preconditions is met:

(1) the execution time of a task is above a certain predefined threshold determined from user demands (e.g., based on desired task periodicity);

(2) the CPU usage and memory consumption of the agent are above a threshold;

(3) the machine hosting the agent is overloaded;

(4) an external event occurs, for instance, another agent needs to execute a task with high priority, as a result the load of the other tasks on the machine should be reduced.

Using the approach with dedicated load-balancing agents, the logic of the agents responsible for processing is kept simple and more maintainable. Each such LBA runs on the nodes which are part of the multiagent processing infrastructure and periodically executes a procedure as presented in Algorithm 1. Based on the current level of load l_c on the current host, the agent decides to optimize the task throughput (Algorithm 2) or to reduce the resource usage (CPU and memory) by migrating agents (Algorithm 4). Over time, it is possible that agents are alive but have little or no work, due to the unavailability of sensors. This problem is addressed in Algorithm 3. In this section, we present a three-step procedure to balance the load across machines used for autonomic sensor data processing.

5.1. Task Throughput Optimization. The approach taken to deal with a high agent workload is splitting tasks in subtasks

```
1: Procedure LOADBALANCE()
2:     A[] ← discoverMobileAgents()
3:     if (l_c < l_th) then
4:         optimizeTaskThroughput(A[])              ▷ Algorithm 2
5:         optimizeNumberOfAgents(A[])              ▷ Algorithm 3
6:     else
7:         optimizeResourceUtilization(A[])         ▷ Algorithm 4
8:     end if
9: end procedure
```

Algorithm 1: Load-balancing procedure to reduce the work load of an agent. $A[]$: list containing mobile agent identifiers, l_c: CPU load on the current host, and l_{th}: predefined CPU threshold.

```
1: Procedure optimizeTaskThroughput(A: Set of mobile agents)
2:     l_max ← 0
3:     a_o ← null
4:     for all a ∈ A do                              ▷ Find max. loaded agent
5:         l_a ← getTaskLoad(a)
6:         if l_a > l_th,max then
7:             l_max ← l_a
8:             a_o ← a
9:         end if
10:    enf for
11:    if a_o not null then                          ▷ Split task load
12:        sids_ao[] ←getSensorIDs(a_o)
13:        splitsids_ao[][] ←split(sids_ao[], 2)
14:        n ←length(splitsids_ao[])
15:        if n = 2 then
16:            n_ao ← numberOfClones(a_o)
17:            n_A ← length(A)
18:            if (n_ao < n_th,ao) ∧ (n_A < n_th,A) then   ▷ Local cloning
19:                a_c,o ← clone(a_o)
20:                reAssignSensors(a_c,o, splitsids_ao[0])
21:                reAssignSensors(a_o, splitsids_ao[1])
22:            else                                  ▷ Remote cloning
23:                lba[] ← discoverLBAs()
24:                lba_u ← findSuitableHost(lba[], a_o)
25:                if lba_u not null then
26:                    a_c,o ← lba_u.clone(a_o)
27:                    lba_u.assignSensors(a_c,o, splitsids_ao[0])
28:                    reAssignSensors(a_o, splitsids_ao[1])
29:                end if
30:            end if
31:        end if
32:    end if
33: end procedure
```

Algorithm 2: Algorithm to optimize the task throughput by assigning subtasks to clones. l_{max}: maximum task load, $l_{\text{th,max}}$: upper threshold task load, l_a: agent task load, $n_{\text{th},A}$: mobile agent instances threshold, a_o: overload agent identifier, a_c: agent clone identifier, $sids[]$: list with sensor identifiers, and $n_{\text{th},a}$: agent instances threshold.

and to do the processing in parallel. In sensor networks, a task can be split according to the sensor identifier and distributed across other agents. This concept is illustrated in Algorithm 2.

The first step is to search for overloaded agents (lines 22–10). An agent is considered overloaded if the average task load l_a exceeds a limit $l_{\text{th,max}}$. Only the agent with the maximum load will be selected for task splitting. The next step is to divide the sensor list in two (lines 12–14) and assign one part locally and the other to a clone. To avoid the presence of too many agents on one machine, there is first a check (line 18) on the number of clones n_{a_o} and total agents n_A. If local cloning is possible, a copy is made of the overloaded agent and the split task distributed over

```
1: Procedure optimizeNumberOfAgents(A: Set of mobile agents)
2:    l_min ← 0
3:    a_u,1 ← null
4:    for all a ∈ A do                                  ▷ Find max. loaded agent
5:       l_a ← getTaskLoad(a)
6:       if l_a < l_th,min then
7:          l_min ← l_a
8:          a_u,1 ← a
9:       end if
10:   end for
11:   if a_u,1 not null then
12:      sids_a_u,1 [] ←getSensorIDs(a_u,1)
13:      a_u,2 ← findSuitableAgent(sids_a_u,1 [])
14:      if a_u,2 not null then                         ▷ Local task passing
15:         assignSensors(a_u,2, sids_a_u [])
16:         kill(a_u,1)
17:      else
18:         lba[] ← discoverLBAs()
19:         lba_u ← findSuitableAgent(lba[], sids_a_u,1 [])
20:         if lba_u not null then                      ▷ Remote task passing
21:            a_u,2 ← lba_u.findSuitableAgent(sids_a_u,1 [])
22:            lba_u.assignSensors(a_u,2, sids_a_u [])
23:            kill(a_u,1)
24:         end if
25:      end if
26:   end if
27: end procedure
```

Algorithm 3: Algorithm to reduce the number of underloaded agents in the multi-agent system by using task merging. l_{min}: minimum task load, $l_{\text{th,min}}$: lower threshold task load, l_a: agent task load, $a_{u,1}$: most underloaded agent identifier, $a_{u,2}$: second most underloaded agent identifier, and $sids[]$: list with sensor identifiers.

```
1: Procedure optimizeResourceUtilization(A: Set of mobile agents)
2:    l_max ← 0
3:    a_o ← null
4:    for all a ∈ A do                                  ▷ Find max. loaded agent
5:       l_a ← getTaskLoad(a)
6:       if l_a > l_th,max then
7:          l_max ← l_a
8:          a_o ← a
9:       end if
10:   end for
11:   if a_o not null then
12:      lba[] ← discoverLBAs()
13:      lba_u ← findSuitableHost(lba[], a_o)
14:      if lba_u not null then                         ▷ Agent migration
15:         a_c,o ← lba_u.clone(a_o)
16:         sids_a_o [] ← getSensorIDs(a_o)
17:         lba_u.assignSensors(a_c,o, sids_a_o [])
18:         kill(a_o)
19:      else                                           ▷ Agent priority reduction
20:         increasePeriodicity(a_o)
21:      end if
22:   end if
23: End procedure
```

Algorithm 4: Algorithm to reduce CPU and memory usage by migrating agent across physical machines and task periodicity adaptation. l_{max}: maximum task load, $l_{\text{th,max}}$: upper threshold task load, l_a: agent task load, a_o: overload agent identifier, a_c: agent clone identifier, and $sids[]$: list with sensor identifiers.

the two entities. If a local clone cannot be made due to the constraints, an attempt is made to create a clone on another machine (lines 22–28). Therefore, a discovery is needed of all LBAs in the processing infrastructure. The local LBA has to communicate to find out if a clone can be made on the remote machine. Based on the free resources (e.g., average CPU load, current amount of agents on the node), an LBA will be selected for remote cloning.

5.2. Number of Agents Optimization. Because the number of active sensors in the network is dynamic, it is difficult to determine in advance the optimal number of processing agents. Over time, it is possible that an agent is inactive most of the time due to an inactive sensor. An approach to deal with this problem is presented in Algorithm 3 and Figure 4 and entails task merging.

Firstly, there is a search for underloaded agents (lines 2–10). If the number of tasks over time l_a is below a threshold $l_{\text{th,min}}$, it will be considered as an underloaded agent. When such an entity is found, there will be a task-merging attempt with another local (lines 12–16) or remote underloaded agent (lines 18–24). After task-merging, one underloaded agent will execute the combination of the two tasks, and the other agent will be removed from the system.

5.3. Resource Usage Optimization. In order to avoid high CPU loads and excessive memory usage, it should be possible to relocate a complete agent instead of a part of the workload. Agent migration is a solution to free local resources. A major challenge is to find the agent that is responsible for the high load or memory usage on the local host. As illustrated in Algorithm 4, we assume that the agent with the most tasks will be likely using the most resources and is a candidate for migration. Another approach is to use monitoring information. The LBA checks the evolution of the CPU load after a migration. When the relocation has no significant influence on the resource usage, the LBA could ignore this agent for future migration decisions. A fallback step is to increase the periodicity (e.g., from each second to every 5 seconds) of the task to decrease the CPU load.

In a first step, an agent will be searched with an average number of tasks l_a which is greater than a threshold $l_{\text{th,max}}$. If such an entity is found, there is a discovery phase to find all LBAs in the infrastructure. If a host with sufficient resources can be found, the agent will be migrated. Selection of the suitable host will be based on the following parameters: average CPU load, available memory, and number of agents present on the node. The host selection and agent migration procedure is presented in lines 12–16. When no migration is possible, the fallback step is executed which entails increasing of the task periodicity (line 18).

6. Experimental Results

6.1. Test Setup. In order to evaluate the data collection and processing functions, we created a test setup in a lab environment. The setup is illustrated in Figure 5 and consists of Alix 3C3 [33] single-board computers running the Voyage linux [34] operating system responsible for measuring functions. These devices are also handling the data preprocessing and caching into files. At predefined intervals, measurements are uploaded to a server. A processing server is responsible for storing the data into a datastore. Table 1 summarizes the hardware specifications and software versions of the devices in the test setup.

6.2. Data Preprocessing. On-board preprocessing data is an approach to limit the amount of data to be stored. By carrying out as much as possible processing on the node itself (as limited the resources may be), the required storage capacity on the node and the needed bandwidth are reduced. This approach has been successfully applied to the noise sensors. The audio card on the Alix single-board computer (SBC) is used to sample the input signal at 48000 Hz in mono, and each sample has a size of 2 bytes. The required storage to buffer one second of audio is 93.75 KiB. Storing and sending this data can become problematic, due to the storage and connectivity constraints of the sensor node. Instead of storing and sending the raw audio signal, the data is preprocessed into a format which still contains the characteristics of the audio signal. A typical preprocessing step is the calculation of the 1/3 octave band levels, which represents the spectral content of the signal. As a result, it is possible to store one second of sound in a format that needs limited amount of storage and still contains enough information to be meaningful for postprocessing and analysis. We have evaluated the CPU load, memory consumption, and required network bandwidth of the preprocessing process on the sensor node. One of the most important parameters that influence the system usage is the FFT input length, which we varied from 1 to 0.125 s of audio samples. The results are presented in Table 2.

In order to evaluate the importance of the on-board preprocessing we implemented a test setup where the Alix SBC is only used for recording audio files covering 1 second and sending the data to the server for processing. The server CPU load, memory consumption, and minimum network bandwidth are shown in Table 3. We noted a huge increase in average CPU load compared to the sensor node side processing. This can be explained by the availability of the audio samples. On the SBC, the calculation process blocks until the FFT buffer is full, and the 1/3 octave band levels are subsequently calculated. On the server side, this process does not have to wait, resulting in a higher CPU load.

Tests showed that the Alix SBC is 15 times slower in calculating the 1/3 octave band levels compared to the server. Despite the speed difference, the server has an additional overhead of reading the received audio files into memory before the processing, which is not negligible. When processing the audio file with an FFT input length of 1 s, it takes 5.45 ms to read the file and 5.23 ms to calculate the 1/3 octave band levels. In this configuration, the server is capable of processing the data from only 9 audio sensors in real time.

6.3. Data Transfer Strategy. The used sensor nodes in the test setup are all equipped with a 4 GB compact flash card. Despite the large storage capacity, data needs to be

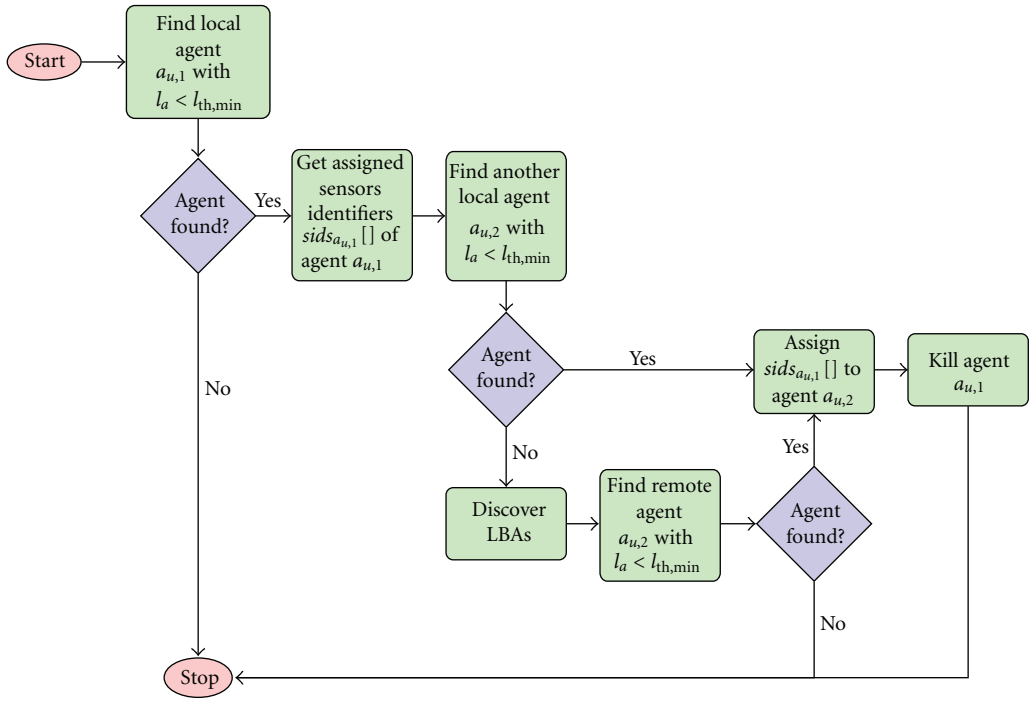

Figure 4: Flow chart illustrating the agent merge principle. The algorithm starts with searching for an underloaded agent ($l_a < l_{th,min}$). When such an entity can be found, there is a second search with possible inclusion of other machines. As soon as two underloaded agents can be found, tasks are merged and assigned to one entity.

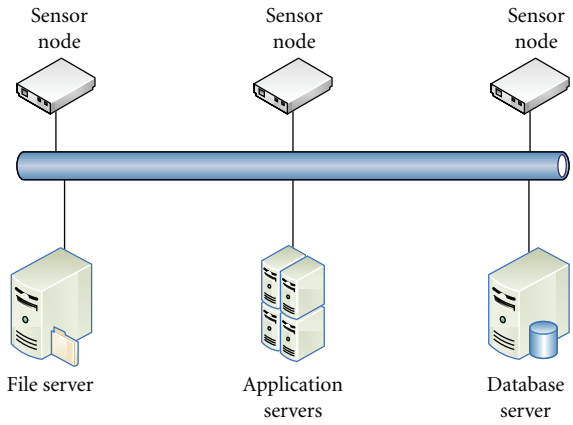

Figure 5: Overview of the devices in the test setup.

transferred to the server regularly in an efficient and reliable manner. We investigated two approaches: streaming of the data and file-based uploads.

In the streaming approach, measurements are pushed near real time (every second) to the server over a TCP socket. For reliability reasons, the server sends back a 32-bit checksum of the data. Only if the checksums match, the data is deleted from the send buffer on the sensor node.

In the file-based approach, measurements are stored in files which are uploaded at regular intervals (i.e., every minute). When the file server confirms the successful upload, data is removed from the sensor node.

We evaluated both data transfer strategies using data from the high-bandwidth audio sensors for different output rates. In order to make efficient use of the available network bandwidth, we also applied GZIP compression on the data. The results are shown in Table 4. In the file-based approach, a higher compression ratio can be achieved resulting in a more efficient use of the available network bandwidth.

We also evaluated the scalability of the server responsible for receiving and caching the measurements. The tests were executed in a configuration where one client machine sends data to a server using a varying number of concurrent connections. The data was transmitted over a 1 Gbps Ethernet connection. In both transfer strategies, the server can easily handle 100 concurrent connections without degrading throughput. Because the file-based transfer strategy uses the available network bandwidth more efficiently (as illustrated in Table 4), we adopted such an approach. The throughput results for this scenario are presented in Figure 6.

6.4. Agent-Based Processing Scalability. In order to evaluate the effectiveness of the load balancing algorithm as described in Section 5, a distributed multiagent processing infrastructure was implemented.

Three types of agents were used, and their functionalities can be described as follows.

(i) The *load balancing agent* (LBA) executes the load balancing algorithm and is deployed on each physical

Table 1: Overview of the hardware specifications and software versions.

Name	Sensor node	Application servers	Database server
CPU	Geode(TM) Integrated	AMD Athlon (64) X2 Dual core processor	Dual-core AMD Processor 2212
CPU speed	498.095 MHz	1000 MHz	2010.325 MHz
RAM	256 MB	2048 MB	4096 MB
Linux Kernel version	2.6.30-voyage	2.6.38-11-generic	2.6.18-6-amd64
Java version	1.6.0u21-b06	1.6.0u21	1.6.0u16-b01
OSGi version	Apache felix 3.0.1	Apache felix 3.0.1	Apache felix 3.0.1

Table 2: Overview of the average CPU load, memory usage, and required network bandwidth of the Alix single-board computer when audio is preprocessed on the sensornode side.

FFT input length (s)	CPU load (%)	Memory usage (MiB)	Network bandwidth (Bytes/s)
0.125	57.5	1.8	355
0.25	42.8	1.8	177
0.5	16.3	2.0	88
1	2.1	2.2	45

Table 3: Overview of the average CPU load, memory usage, and minimum network bandwidth of the server when the audio is not preprocessed on the sensornode side.

FFT input length (s)	CPU load (%)	Memory usage (MiB)	Network bandwidth (Bytes/s)
0.125	98	2.5	96000
0.25	95	2.6	96000
0.5	92	2.6	96000
1	90	2.7	96000

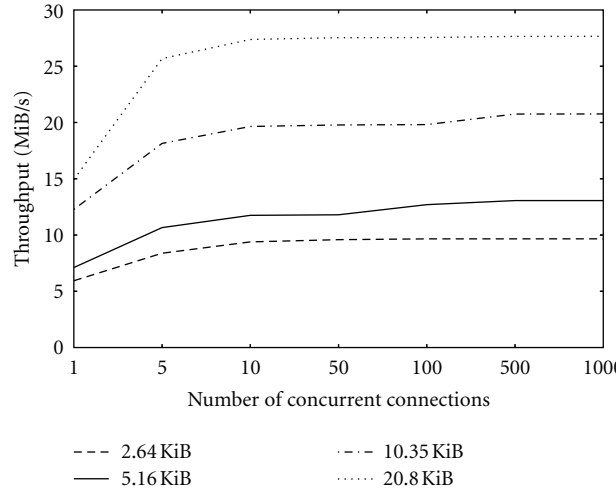

Figure 6: Overview of the server throughput as a function of the number of concurrent connections and varying file sizes. Each file contains 60 seconds of logging data with a different amount of samples per second.

machine in the multi-agent processing infrastructure. Each of such LBAs decides when load balancing is necessary and coordinates the process. Because the amount of processing nodes in the network can be dynamic, the agent should be capable of discovering other LBAs at run time. As a consequence, an LBA heavily relies on the discovery service provided by the agent middleware. In order to select the most suitable host for task allocation or agent migration, each LBA should be able to communicate with other load-balancing agents. The messaging services provided by the agent framework are used for the interaction.

(ii) As an example of a very simple task that a data analysis agent could perform, we consider an agent that calculates a 10- or 15-minute average of the incoming data. For this example, we also assume that all measurement data are correct and all sensors are reliable, so no prior check on validity is needed. The *task manager agent* is responsible for the management of average tasks in the processing infrastructure. When a measurement is received from the sensor network, the task manager agent calculates the interval start time (e.g., 10 min or 15 min) and stores this timestamp with the sensor identifier in the task list. An average task can be seen as an object that contains an interval start timestamp, length, and a sensor identifier.

(iii) *Average calculator agent* has the responsibility for calculating the average value of all measurements within a certain interval for one or multiple sensors. The processing is periodic, and when the agent wakes up, it fetches all average tasks from the task manager agent. When there are tasks available, measurements within the interval are retrieved from the datastore, the average value is calculated and stored. The tasks are stored in the internal queue of each agent. When a task is finished, the task manager agent is notified who deletes the task. When all tasks are processed, the agent suspends. The logic of the average calculator agent and the interaction with the task manager agent is illustrated in Figure 7.

6.4.1. Influence of Local Agent Cloning. In a first test, we evaluated the influence of local cloning on the task throughput. We monitored the CPU load and used network bandwidth

Table 4: Comparison table of the average used bandwidth when preprocessed audio data is streamed to the server near-real time (every second), and logging files holding 60 seconds of data are transmitted every minute.

FFT input length (s)	Streaming		File-based upload	
	Uncompressed (Bytes/s)	Compressed (Bytes/s)	Uncompressed (Bytes/s)	Compressed (Bytes/s)
0.125	1691	524	1558	355
0.25	863	312	797	177
0.5	449	214	409	88
1	242	156	210	45

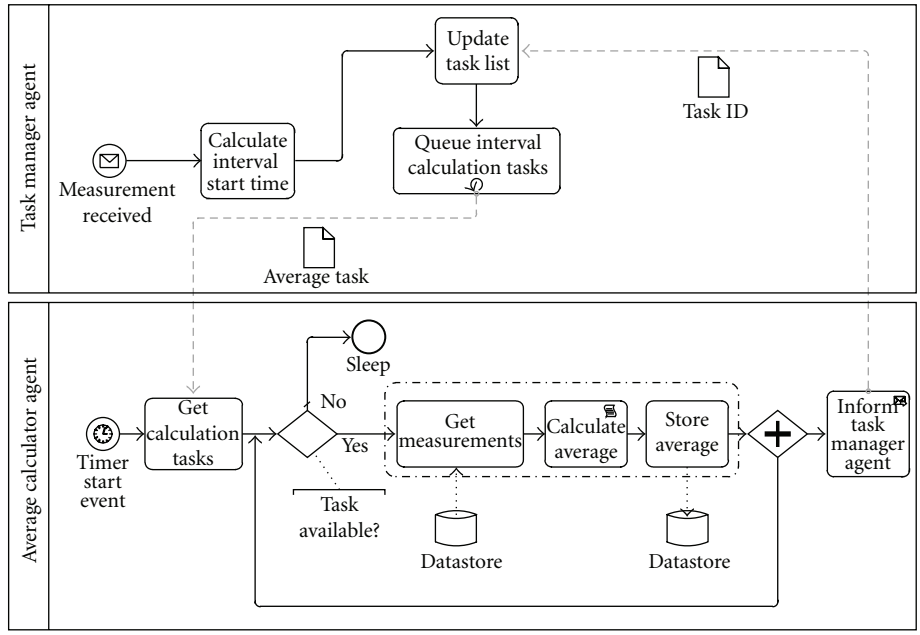

Figure 7: BPMN business process diagram illustrating the logic of the tasks manager agent and average calculator agent.

and task throughput in a configuration where a task queue is processed in parallel by a varying number of agents. In this test setup, the load-balancing algorithm was disabled, and the number of agents was configured in advance. All agents were located on one machine, and the measurement data needed for executing the task was fetched from a database server over a 100 Mbps network link. The test revealed that an increased task throughput can be achieved by hosting multiple agents on one machine, as illustrated in Figure 8. During the processing of a task, 60% of the time is spent on the calculation of the average audio levels and 40% on database communication. In this configuration, we noticed a decrease in task throughput when the task queue is processed in parallel by five agents. The main reason for this is that the processing is data driven and the communication with the database server becomes the bottleneck.

6.4.2. Influence of Data Locality. In order to minimize network communication with the datastore, we modified the logic of the LBA, to also allow cloning to the database

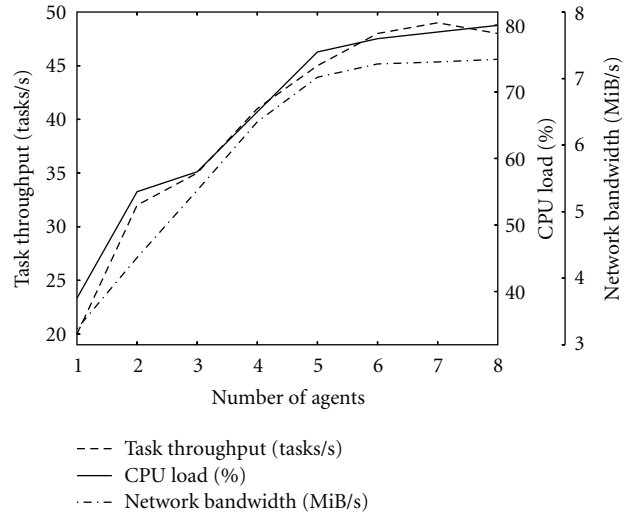

Figure 8: Overview of the task throughput, CPU load, and network bandwidth when a task queue is processed by a varying number of agents residing on one machine.

server. The load-balancing algorithms were tested in a four-machine processing infrastructure in two cases: with and without using the data locality optimisation. On each node, there was an LBA deployed communicating over 100 Mbps network links. The test results are presented in Table 5. We can conclude that by making use of data locality a huge improvement in task throughput can be achieved.

6.4.3. Data Processing Scalability. In order to evaluate the level of improvement using the load-balancing algorithms, we first tested the task throughput when tasks are processed sequentially by one agent with no load-balancing. A single process is able to complete twenty tasks in one second, which is illustrated in Figure 9.

Next, we tested our load-balancing algorithm with a varying number of processing machines and virtual sensors, also allowing cloning on the database server. The tests were executed in four-machine infrastructure connected by a 100 Mbps network. On each node, there was LBA deployed, with equal configuration parameters. During the tests, we used the following parameters for each LBA:

(i) maximum number of tasks before an agent is marked as overloaded ($l_{\text{th,max}}$): 50,

(ii) minimum number of tasks before an agent is marked as underloaded ($l_{\text{th,min}}$): 5,

(iii) CPU load (l_{th}) limit: 70%,

(iv) memory usage limit: 80%,

(v) maximum number of agent clones ($n_{\text{th},a}$): 9,

(vi) maximum number of agents ($n_{\text{th},A}$): 50.

The test results showed that a significant increase in task throughput can be achieved by balancing the load across agents and machines in the processing infrastructure. Making use of data locality results also in a significant performance increase. The $l_{\text{th,max}}$ parameter is of great importance, because it influences how fast local and remote clones will be created. As soon as the number of tasks exceeds $l_{\text{th,max}}$, a procedure is started to create local or remote clones. Setting this value too low will result in more unnecessary cloning operations, resulting in overhead which has an influence on the task throughput.

6.4.4. Influence of Agent Priodicity Adaptation. In the currently presented evaluation results, we assumed a uniform task load for all the sensors. In real-world scenarios, this may not be the case, for example, a sensor node can be disconnected from the network for several days, and when such a node is connected again, the task load for the attached sensors will be high. Our load-balancing algorithms respond to such a situation by moving the agents responsible for those sensors to machines with the most free resources. This can result in a situation where the agents are suboptimally distributed over the machines. To avoid such a situation, the load-balancing algorithms were improved by lowering

Table 5: Task throughput as a function of the number of machines evaluated in two cases. In a first scenario, no agents are allowed to run on the server hosting the database, resulting in a lower task throughput due to the network delays. In a second case, agents are given priority to clone on the database server and are exploiting the data locality.

Number of Processing Nodes	Maximum task throughput (tasks/s)	
	No data locality	Data locality
2	71	144
3	75	156
4	78	161

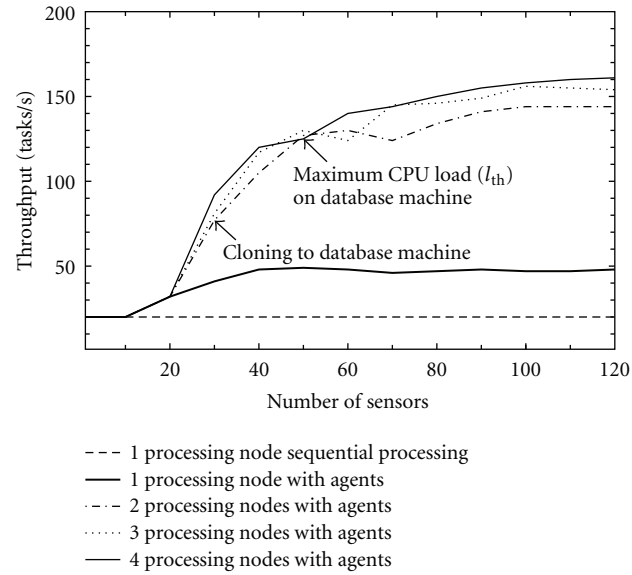

Figure 9: Task throughput as a function of the number of sensors in the network. All agents process one task per iteration.

the priority of the agents that meet all of the following preconditions:

(i) the CPU load l_c on the host where the agent resides is higher than a predefined threshold l_{th};

(ii) the task load l_a is above a threshold $l_{\text{th,max}}$;

(iii) no suitable machines can be found for agent migration.

We evaluated this approach by measuring the task throughput in two scenarios: with and without priority adaptation. In both cases, a task queue with data of fifty sensors was periodically processed by agents in a infrastructure consisting of four machines. At time t_0, we assumed a steady state of the load-balancing algorithms with a nearly constant task throughput. Next, we increased the task load for an increasing number of sensors from 1 task (15 minute period) to 96 tasks (24-hour period) at time t_1, t_2, and t_3. In both scenarios, we measured a decrease in overall task throughput. As illustrated in Figure 10, we noticed a higher performance in the case where priorities were adapted.

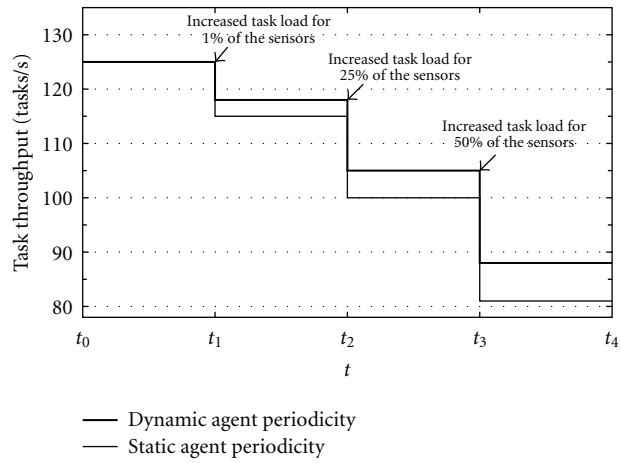

Figure 10: Overview of the influence of the increased task load for one or more sensors on the task throughput. The test was executed with fifty sensors on a distributed processing infrastructure consisting of four machines. In all cases, task load per iteration was increased from 1 task (15-minute period) to 96 tasks (24-hour period).

7. Conclusions and Future Work

In this paper, we have showed how intelligence can be brought to the several layers of the sensing application to achieve the required level of autonomy. By adopting the mobile agent approach, logic can be distributed into the measuring network. The intelligence from these agents can range from regulating the data preprocessing on the sensor node to the optimisation of the calculation tasks on the back-end infrastructure. Secondly, we have implemented load-balancing approaches to maintain the scalability of the fine-grained data processing tasks in a distributed environment. Therefore, this logic was incorporated into load-balancing agents that optimize the task throughput and resource utilisation on the current host and also collaborates with other agents for its operation. We have proven that using principles as cloning, task passing, and reallocation, a scalability processing infrastructure can be achieved with respect to the number of sensors. We also showed that the system is able to handle sudden task load increases by lowering the priority of those tasks, resulting in a fairer resource sharing. For completeness, we included a performance analysis of the current prototype, based on the design decisions.

In the future work, we plan to take advantage of the self-learning capabilities of an agent. As a result, the load-balancing agent will have the ability to take more intelligent decisions, which results in less unnecessary agent migrations. For example, by monitoring the resource utilisation after a migration/cloning, the LBA can assess the effect of the action and learn from it. This acquired knowledge can be used during future decisions. Secondly, we plan to investigate several database partitioning mechanisms. Data can be clustered according to parameters such as geographical region, time, and number of queries. The intelligent clustering of sensor data can further decrease the number of agent migrations and increase the overall performance.

Acknowledgment

This research is part of the IDEA (Intelligent, Distributed Environmental Assessment) project [35], a 4-year strategic basic research project, financially supported by the IWT-Vlaanderen (Flemish Agency for Innovation by Science and Technology).

References

[1] L. Mottola and G. P. Picco, "Programming wireless sensor networks: fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, pp. 19:1–19:51, 2011.

[2] K. Henricksen and R. Robinson, "A survey of middleware for sensor networks: state-of-the-art and future directions," in *Proceedings of the International Workshop on Middleware for Sensor Networks (MidSens '06)*, pp. 60–65, November 2006, Co-located with Middleware 2006.

[3] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[4] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, "Reconfigurable component-based middleware for networked embedded systems," *International Journal of Wireless Information Networks*, vol. 14, no. 2, pp. 149–162, 2007.

[5] The OSGi Alliance The OSGi framework, 2011, http://www.osgi.org.

[6] D. Hughes, K. Thoelen, W. Horré et al., "LooCI: a loosely-coupled component infrastructure for networked embedded systems," in *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM '09)*, pp. 195–203, December 2009.

[7] Java Platform, Micro Edition (Java ME), 2011, http://www.oracle.com/technetwork/java/javame/index.html.

[8] P. Boonma and J. Suzuki, "BiSNET: a biologically-inspired middleware architecture for self-managing wireless sensor networks," *Computer Networks*, vol. 51, no. 16, pp. 4599–4616, 2007.

[9] P. K. Biswas, H. Qi, and Y. Xu, "Mobile-agent-based collaborative sensor fusion," *Information Fusion*, vol. 9, no. 3, pp. 399–411, 2008.

[10] Y. Ma, M. Richards, M. Ghanem, Y. Guo, and J. Hassard, "Air pollution monitoring and mining based on sensor Grid in London," *Sensors*, vol. 8, no. 6, pp. 3601–3623, 2008.

[11] M. Bell and F. Galatioto, "Novel wireless pervasive sensors network to improve the understanding of noise across urban areas," in *Proceedings of the 8th European Conference on Noise Control (Euronoise '09)*, Edinburgh, UK, 2009.

[12] L. Luo, Q. Cao, C. Huang et al., "Design, implementation, and evaluation of EnviroMic: a storage-centric audio sensor network," *ACM Transactions on Sensor Networks*, vol. 5, no. 3, pp. 1–35, 2009.

[13] A. Can, M. Rademaker, T. Van Renterghem et al., "Correlation analysis of noise and ultrafine particle counts in a street canyon," *Science of the Total Environment*, vol. 409, no. 3, pp. 564–572, 2011.

[14] T. Van Renterghem, P. Thomas, F. Dominguez et al., "On the ability of consumer electronics microphones for environmental noise monitoring," *Journal of Environmental Monitoring*, vol. 13, no. 3, pp. 544–552, 2011.

[15] K. B. Lee and M. E. Reichardt, "Open standards for homeland security sensor networks—sensor interconnection and integration trough Web access," *IEEE Instrumentation and Measurement Magazine*, vol. 8, no. 5, pp. 14–21, 2005.

[16] M. Botts, G. Percivall, C. Reed, and J. Davidson, "OGC sensor web enablement: overview and high level architecture," *Lecture Notes in Computer Science*, vol. 4540, pp. 175–190, 2008.

[17] S. Cox, "Geographic information: observations and measurements OGC abstract specification topic 20," in *OpenGIS Abstract Specification 10-004r3*, 2010, Version: 2.0.0.

[18] 52North Initiative for Geospatial Open Source Software GmbH, 2011, http://52north.org/.

[19] A. Na and M. Priest, *OGC Implementation Specification 06-009r6: OpenGIS Sensor Observation Service (SOS)*, Open Geospatial Consortium, Wayland, Mass, USA, 2007.

[20] I. Simonis, *OGC Best Practices 06-028r3: OGC Sensor Alert Service Candidate Implementation Specification*, Open Geospatial Consortium, Wayland, Mass, USA, 2006.

[21] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "JADE: a software framework for developing multi-agent applications. Lessons learned," *Information and Software Technology*, vol. 50, no. 1-2, pp. 10–21, 2008.

[22] IEEE Foundation for Intelligent Physical Agents, 2011, http://www.fipa.org/.

[23] J. S. Rellermeyer and G. Alonso, "Concierge: a service platform for resource-constrained devices," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 245–258, 2007.

[24] A. Bottaro and F. Rivard, "RFP 126 OSGi ME—An OSGi Profille for Embedded Devices," OSGi Alliance, 2009.

[25] Sun SPOT World, 2011, http://www.sunspotworld.com/.

[26] C. Muldoon, G. M. P. O'hare, R. Collier, M. J. O'grady et al., "Agent factory micro edition: a framework for ambient applications," in *Proceedings of International Conference on Computational Science*, pp. 727–734, 2006.

[27] F. Bellifemine, G. Caire, and D. Greenwood, "Running JADE agents on mobile devices," in *Developing Multi-Agent Systems with JADE*, chapter 8, John Wiley & Sons, Chichester, UK, 2007.

[28] B. Chen, H. H. Cheng, and J. Palen, "Merging agents and services the JIAC agent platform," *Multi-Agent Programming*, vol. 1, pp. 159–185, 2009.

[29] H. Benjamin, K. Thomas, and H. Axel, "Mobile-C: a mobile agent platform for mobile C/C++ agents," *Software: Practice and Experience*, vol. 36, no. 15, pp. 1711–1733, 2006.

[30] Y. G. Ha, "Dynamic integration of Zigbee home networks into home gateways using OSGi service registry," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 2, pp. 470–476, 2009.

[31] J. Stender, S. Kaiser, and S. Albayrak, "Mobility-based runtime load balancing in multi-agent systems," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE '06)*, Knowledge Systems Institute, 2006.

[32] S. Eludiora, O. Abiona, G. Aderounmu, A. Oluwatope, C. Onime, and L. Kehinde, "A load balancing policy for distributed web service," *International Journal of Communications, Network and System Sciences*, vol. 3, no. 8, pp. 645–654, 2010.

[33] Alix 3C3, 2011, http://www.pcengines.ch/alix3c3.htm.

[34] Voyage Linux: x86 Embedded Linux = Green Computing, 2011, http://linux.voyage.hk/.

[35] IDEA (Intelligent, Distributed, Environmental Assessment) Project, 2011, http://idea.intec.ugent.be/en/.