Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 178 (2008) 2105–2127
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Using cooperative mobile agents to monitor distributed and dynamic environments ☆

Sergio Ilarri [a,*], Eduardo Mena [a], Arantza Illarramendi [b]

[a] IIS Department, University of Zaragoza, María de Luna 1, Zaragoza 50018, Spain
[b] LSI Department, University of the Basque Country, Apdo. 649, 20080 San Sebastián, Spain

Received 9 May 2007; received in revised form 19 November 2007; accepted 23 December 2007

Abstract

Monitoring the changes in data values obtained from the environment (e.g., locations of moving objects) is a primary concern in many fields, as for example in the pervasive computing environment. The monitoring task is challenging from a double perspective. First and foremost, the environment can be highly dynamic in terms of the rate of data changes. Second, the monitored data are often not available from a single computer/device but are distributed; moreover, the set of data providers can change along the course of time. Therefore, obtaining a global snapshot of the environment and keeping it up-to-date is not easy, especially if the conditions (e.g., network delays) change.

In this article, a decentralized, loose, and fault-tolerant monitoring approach based on the use of mobile agents is described. Mobile agents allow easy tracking of the involved computers, carrying the monitoring tasks to wherever they are needed. A deadline-based mechanism is used to coordinate the cooperative agents, which strive to perform their continuous tasks in time while considering data as recent as possible, constantly adapting themselves to new environmental conditions (changing communication and processing delays). This approach has been successfully used in a real environment and experiments were carried out to prove its feasibility and benefits.

Keywords: Distributed monitoring; Distributed systems; Mobile agents; Cooperative agents; Coordination

1. Introduction

With the upcoming arrival of pervasive computing, environments where many distributed heterogeneous entities (e.g., sensors, computing devices, portable computers, etc.) must cooperate and interchange information to perform a task have become commonplace. Specifically, obtaining global snapshots of data in wireless

☆ Supported by the CICYT project TIN2007-68091-C02. We would also like to thank the support of the Spanish Excellence Network of Agents (CICYT TIN2005-25869-E), and Edith Bosque for her help with some English doubts.
* Corresponding author. Tel.: +34 976 76 23 40; fax: +34 976 76 19 14.
E-mail addresses: silarri@unizar.es (S. Ilarri), emena@unizar.es (E. Mena), jipileca@si.ehu.es (A. Illarramendi).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.12.015

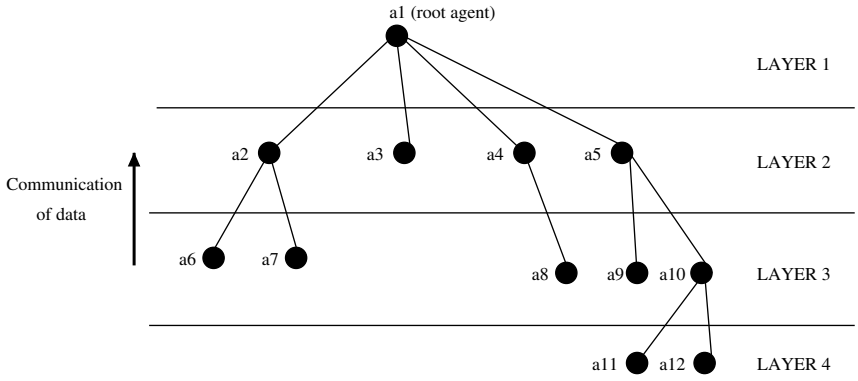

Fig. 1. Example of layered agent architecture.

environments and keeping the obtained global snapshots as up-to-date and consistent as possible is difficult due to the distributed and dynamic nature of such context.

In general, a monitoring task in the environments considered presents important challenges. As the involved data can change very frequently, a continuous and updated answer must be provided to the user by efficiently monitoring the environment (e.g., a change in the answer must be detected as soon as possible while minimizing the communication cost). On the contrary, the set of involved devices/computers can change dynamically over time, and so the distributed monitoring must adapt itself quickly to the new requirements. Moreover, every device/computer will be affected by different conditions in terms of its network resources (connectivity and available bandwidth), processing power, and current overload. The dynamic nature of the environment regarding the data that must be monitored, the devices/computers involved in the monitoring, and the environmental conditions is a challenge. So, the continuous distributed retrieval of highly dynamic data is of paramount importance.

A *divide-and-conquer approach* based on software agents [35] (autonomous software components) is a natural way to tackle the monitoring tasks in the aforementioned distributed environments. Thus, a monitoring application (represented by a *root agent*) can achieve its goal by cooperating with a set of remote helping agents, which could, in turn, require the collaboration of other remote helping agents, and so on, until the devices that obtain the relevant data can be accessed directly by agents that are leaf nodes in the agent architecture (see Fig. 1 for an example). As the involved data are constantly changing, the agents must perform their tasks with a certain *task frequency* (to limit the cost of monitoring), communicating new data recursively through the network of agents up to the monitoring application. The number of layers needed is a consequence of the divide-and-conquer strategy applied, which depends on the requirements of the monitoring application (e.g., the division into layers can be guided by geographic criteria and/or the existence of different types of roles/tasks). Some examples of this approach are provided as follows:

- To monitor the temperature of several cities in three states, an architecture composed of the following components could be considered: (1) a root agent (which provides the retrieved information to the user); (2) three helping agents at layer 1 (one for each state); and (3) for each agent at layer 1, several helping agents at layer 2 (one for each city, in the corresponding state, whose temperature must be monitored). The designer of the multiagent architecture could also decide to use just one layer of helping agents, i.e., one helping agent for each city (independent of the states). In this context, the values of the temperatures might be required to be updated every 30 min.
- The monitoring of the stock quotes of the New York Stock Exchange[1] could be considered with an architecture having: (1) a root agent providing the information to the user (layer 1); (2) several helping agents at layer 2 to monitor different industries (e.g., oil and gas, basic materials, consumer goods, etc.); (3) several helping

[1] And according to the information shown at http://www.nyse.com/about/listed/industry.shtml?ListedComp=All.

agents at layer 3 to monitor a specific sector for each industry (e.g., within consumer goods we can distinguish automobiles, food and beverages, etc.); and (4) several helping agents at layer 4 to monitor the quotes for different companies within that specific sector. This division into layers can facilitate different monitoring tasks; for example, if just the top three companies in each sector were to be monitored, the agents at the corresponding layer can filter the unnecessary results, minimizing the communications. Considering the task period, in this scenario the quotes may need to be updated, for example, every 5 min, or as soon as possible.

Regarding the process of continuous monitoring, it is proposed that each agent should follow a deadline-based approach to coordinate its set of helping agents: a *coordinator agent* communicates to its helping agents a *deadline* (the time instant by which it should have received new data from them because it needs to start its own tasks). The helping agents will do their best to try to meet that deadline.

In the following sections, a monitoring example and several application contexts where the proposed monitoring approach can be applied are initially introduced. Then, the advantages of mobile agents are highlighted [32], which is the agent technology that is needed in this context. Finally, the main advantages of our monitoring proposal are summarized.

1.1. Example and application contexts

As a concrete sample for a monitoring application, the problem of detecting all the Bluetooth devices on a university campus is considered. Bluetooth devices can only be detected by a nearby Bluetooth-enabled computer. Therefore, monitoring all the Bluetooth devices on the university campus from a single centralized computer is not even possible directly, and a distributed monitoring approach is required.

According to our proposal, the monitoring application would require a *monitoring agent* in each building on the campus to detect the Bluetooth devices inside that building. Then, each building-level monitoring agent would need other monitoring agents in every room in the building to detect the Bluetooth devices inside that room. These room-level monitors should communicate the Bluetooth devices in their range to their corresponding building-level monitoring agent, which would correlate the information received (e.g., by removing duplicates, in case agents in adjacent rooms detect the same device at the same time) and send the list of devices in the building to the user's monitoring application. This application would correlate the received data and present the result to the user. This data flow from room-level monitoring agents to the user's monitoring application must be continuous, as Bluetooth devices can be switched off/on and may enter/exit the campus at any time.

The geographic area/s under monitoring can also be changed dynamically. For example, the user may want to dynamically add/remove buildings to/from the set of buildings that are to be monitored. Another example can be provided by a mobile user carrying a PDA (Personal Digital Assistant) with Bluetooth, who may want to monitor the devices within 1000 m around his/her location, using the available computers/devices in the surroundings (the PDA cannot perform this task by itself, as the range of Bluetooth is not as large as 1000 m); the set of computers/devices needed will change as the user moves from one place to another. In these situations, shutting down and restarting the monitoring every time there is a change in the set of involved areas should be avoided; similarly, having a monitoring agent on all the computers/devices that could be possibly used in the future is impractical.

The proposed monitoring approach could be useful in scenarios of distributed sensor interpretation [2], data fusion [34], and distributed query processing of dynamic data [15], in which cases, a continuous monitoring of the environment must be carried out. All these applications (*monitoring applications* [25]) have in common a set of agents that is used to obtain data from different sources. In addition, these data must be integrated to get a global view of the situation (*answer*). As data can change along time, the obtained answer should be refreshed periodically. Moreover, all the agents involved should get their data within the same, small, time window so as to get a coherent answer (*consistent snapshot*). As this is the most challenging situation, obtaining complete and up-to-date snapshots of the relevant data has been focused on in this article.

1.2. Benefits of using mobile agents

Mobile agents [32,33] are *programs* that can move autonomously from one computer to another. They are very useful for monitoring purposes because mobility gives the agents the autonomy needed to monitor the

environment effectively, allowing them to keep track of the monitoring area/s easily. Thus, mobile agents offer several advantages:

- A solution based on agents that move and distribute themselves to perform their monitoring tasks leads to a *convenient and clean design*. With this approach, there is no need to keep track explicitly of the computers/devices involved in the monitoring, as this task is carried out distributively by the mobile agents themselves. Mobile agents can communicate among themselves independently of the computers where they are currently executing their tasks [16]: the communications are routed transparently by the *mobile agent platform*, which allows the mobile agents to execute their tasks and provides them with different services (such as migration, communication, and security).
- As opposed to an approach that is based on static agents (i.e., traditional software agents that do not move), mobile agents lead to an *adaptive approach*. Thus, without mobile agents, static monitoring agents that are ready on all the computers/devices would be needed (just in case they are needed for monitoring in the future), which introduces an unnecessary overhead. Moreover, if monitoring is carried out using ad hoc networks, it is not possible to foresee the computers/devices that will be available over time (because they enter/leave the network dynamically), and therefore the required monitoring agents cannot be "preinstalled". On the contrary, only a mobile agent platform is required on those computers/devices if an agent-based approach is adopted; for example, on a road, mobile agents could move from car to car to monitor the road ahead.
- A monitoring application that works on the basis of mobile agents is also a very *flexible approach*. Adding new monitoring functionalities is as easy as incorporating new types of mobile agents into the monitoring system. An instance of a new type of agent can create more agents that will move between the involved computers/devices as needed. This can be carried out dynamically, without disturbing other monitoring systems in operation. Moreover, mobile agents can move across, and interact with, heterogeneous computers and devices.
- By moving themselves "near" the environment they need to monitor, mobile agents can *reduce the network overload and latency*: they minimize the amount of communications that a remote monitoring would otherwise require. Furthermore, sometimes a remote monitoring is not even possible (e.g., in the example described in Section 1.1 only nearby devices can be detected directly).
- Mobile agents also exhibit a *good performance* compared to the traditional client/server approach for remote monitoring [10,31,41]. The importance of mobile agents for distributed, mobile, and pervasive computing has been highlighted, for example, in [4,36,38,39,41,45].

Therefore, mobile agents are very useful from a design point of view (they encapsulate the dynamics of the monitoring process), and they also offer other general benefits [23]. Thus, mobility is a desirable feature for agents that have to monitor distributed and wireless environments.

Mobile agent technology is advocated herein because it provides suitable mechanisms to solve the presented problem in a distributed, efficient, and convenient manner. A mechanism that works on the basis of remote procedure calls could have been used instead. However, this would lead to a nonflexible and more difficult-to-implement solution: agent migration would be replaced by remote invocations that create and destroy threads representing the behavior of the respective agents. Mobile agents bring the required functionality to any computer, thereby avoiding the installation and launch of specialized daemons/servers on each machine to provide the required services. In addition, mobile agents ease the addition of new services once the system is working: new agents with new functionalities could travel to the devices/computers without reinstalling anything there. Moreover, scalability, security, and fault tolerance for mobile agents must be provided by the mobile agent platform used (e.g., [9,16,37]).

1.3. Main benefits of our proposal

The monitoring approach proposed in this article presents the following main advantages: (1) *use of mobile agents* – agents can decide to move among computers to monitor the involved "areas" effectively and efficiently; (2) *adaptation to the environment* – agents adapt dynamically to the current conditions (by postpon-

ing/anticipating their tasks, as needed); (3) *data freshness* – agents try to return the most recent data that can be obtained from the environment, performing *just-in-time* refreshments; (4) *fault-tolerance* – agents are loosely coupled, which leads to a "graceful degrading" of performance when some of them fail to perform their tasks in time; (5) *adaptation to challenging contexts* – it deals with situations wherein some agents cannot perform their tasks at the required frequency, constrained by challenging environmental conditions. Moreover, an *unobtrusive monitoring* that does not overload existing wireless devices with monitoring tasks and instead uses fixed computers whenever it is possible is preferable; even if certain data were to be retrieved from a mobile device, sometimes the raw data can be obtained and processed on a fixed computer, communicating only the relevant data to the device, thereby alleviating its processing and wireless communication overload.

Thus far, the general monitoring approach alone has been described. In the rest of this article, how the agents coordinate to continuously monitor the environment is described in detail. In Section 2, some related works are considered. In Section 3, the process by which the mobile agents coordinate their tasks using *soft deadlines*, which indicate the time instants by which a monitoring agent should provide its results to its coordinator, is reconstructed. In Section 4, some strategies are proposed that deal with the problem arising when some agents cannot perform their tasks at the required frequency due to the challenging environmental conditions. In Section 5, the proposed approach is evaluated experimentally considering a real application. Finally, in Section 6, the conclusions of this study are summarized.

2. Related work

In this section, some related studies are reviewed. First, a description of some reports that used mobile agents to carry out monitoring tasks is provided. The second focus is on researches in the field of cooperative problem solving using software agents. Finally, a comparison of the proposed approach with an architecture for the coordination of just-in-time production and distribution is provided.

Several reports, such as [11,19,21,22,26,30,43], advocate the use of mobile agents to carry out monitoring tasks. However, this is the first effort at proposing a complete and general approach to continuously monitor highly dynamic, distributed environments using mobile agents which also includes a study and evaluation of the synchronization mechanisms needed. Thus, for example, in [22] the focus is on a specific context (traffic monitoring for navigation applications) and not the agent synchronization problem; in [19,21] the goal is to obtain information about processes in a distributed system, wherein neither a *continuous* monitoring nor obtaining global and consistent snapshots is required; a similar comment applies to [30] (although their goal is different: network monitoring); in [11,26] the aim is to quantify the benefits of a monitoring approach that works on the basis of mobile agents; and in [43] a mobile agent-based architecture is defined to monitor computational resources in grid computing, and coordination issues are not considered.

Many reports in the field of distributed artificial intelligence focus on *cooperative problem solving* [7] among groups of independent agents, which negotiate [42,47] the best plan to accomplish a given task. For example, inspired by market mechanisms, the *Contract Net Protocol* (CNP) [40] for decentralized allocation of tasks and many variants of increasing complexity have been proposed. Similarly, there are several coordination mechanisms, such as those based on *bidding* [6] and *Partial Global Planning* [8]. The general setting of these approaches differs from the proposed context, as they consider agents that do not necessarily share a common goal or know each other. Hence, these protocols may not be suited to dynamic contexts (wherein the relevant data, the set of involved devices/computers, and the environmental conditions can change frequently along time) that require agents to perform their tasks continuously, and therefore they cannot be easily applied when a continuous monitoring is required from each involved agent. Moreover, the mobility of agents could also make those protocols inappropriate. Finally, no timing guarantees are provided by these approaches. Despite the popularity of protocols such as the CNP, some experimental results suggest that it is ineffective for scenarios with a relatively high number of agents [18], and therefore, it should be avoided when a quick allocation of tasks is required. However, it could possibly be used to set up a cooperation structure among preexisting agents for monitoring purposes, as a previous step to the herein proposed monitoring approach.

Finally, in [5] an architecture for the coordination of just-in-time production and distribution is presented. Though this is a problem different from the one under study (a monitoring approach based on mobile agents is proposed in this article, whereas they propose a multiagent architecture to coordinate

industrial processes), there are some parallelisms that are worth mentioning. First, their goal is to achieve just-in-time production, in just the same way that the proposed monitoring agents strive to "schedule" their tasks to be carried out just-in-time: as late as possible (to consider the most recent data) but on time (to meet the deadline); however, it should be noted that the temporal scale is expected to be quite different! Second, in their problem there is a tradeoff between the denial of service (due to a shortage in the production) and the cost of surplus production, whereas in the proposed context sometimes a high level of freshness of the retrieved data (i.e., a high task frequency) cannot be achieved while maintaining its consistency (i.e., keeping the agents synchronized), as is explained in Section 4. And third, they need to estimate the consumer needs, whereas the method herein proposed uses estimations of the agents' (correlation and communication) delays.

3. Autosynchronizing monitoring agents: use of deadlines

In a dynamic environment that is constantly changing, agents must monitor the relevant data with a certain task frequency. Thus, the agents have to carry out certain tasks periodically, and the time instants when different agents in the monitoring application perform their tasks must be somehow temporally correlated to ensure some consistency in the results obtained (e.g., to compare the values of two temperature sensors, two temperatures measured at approximately the same time instant are needed). In other words, the agents need to synchronize/coordinate their tasks according to some basic policy. In this section, the basics of the proposed synchronization approach are first described. Then, an account of how deadlines are computed on the basis of delays experienced by the agents is provided. Finally, the automatic adjustment of the required deadlines by the agents concerned, when the environmental conditions change, is elaborated.

3.1. Basics of the synchronization approach

Every agent in the network is proposed to be associated with a certain *deadline*, which allows the agent to determine when it has to perform its tasks and communicate a new result (data needed by the coordinator) to its upper-level agent (the *coordinator agent*). Such a deadline must be computed by the coordinator on the basis of its own deadline, so as to ensure that it will receive the data it needs from its helping agents in time to perform its own tasks.

The use of deadlines is the only way to keep the agents synchronized with each other. An alternative to the deadline-based approach would be a waiting-based approach in which an agent waits until it has received all the necessary data that it needs to correlate (*correlation task*) so as to communicate its own results (*communication task*). However, in such a case, a single agent missing its deadline would prevent the final result of the task from being updated with the required frequency (its coordinator could wait *too much time* for the results from such an agent). Moreover, with such a strategy, there is no way to ensure that the agents will obtain new data *at the same time*, as every agent performs its tasks independently (i.e., without any knowledge about when the other agents will perform theirs).

A loose coupling among agents by using *soft deadlines* [12] is profitable, so that the results from an agent are always stored by its coordinator even if they do not arrive on time. Thus, at the expense of some uncertainty, a global result can be obtained despite partial failures in the system, such as an agent that fails to meet its deadline: its coordinator can still use the latest result received from such an agent. So, a global result can always be obtained (with some uncertainty in the case of missed deadlines). Every agent communicates (to its coordinator) not only its results but also a measure of the quality of the communicated results, which is computed taking into account the time elapsed since it received (from its underlying agents) the data used to obtain such results. In this manner, a global quality measure is eventually obtained by the root agent, which can be used to show the user some indications of the reliability of the relevant results. Two indicators for the quality of reliability are the *percentage of refreshed data* and the *average age of data*. Thus, in an ideal case, the percentage of refreshed data should be 100% (all the data are updated) and the average age of data should be as small as possible (and less than the required task period). The agent architecture can be configured to consider only those results having a certain minimum quality; for example, the user may not be interested in the answers in which less than 80% of the data have been refreshed.

It is not enough for an agent to meet its deadline, but it should also obtain the best results within the existing time constraints: in the context of this discussion, this implies minimizing the *uncertainty gap* (see Definition 1).

Definition 1. The uncertainty gap between an agent x and another agent y is the time elapsed since x finishes the communication of data to y until those data are eventually processed by y (to get its own results). For example, x and y can be a helping agent and its coordinator, respectively, in which case the uncertainty gap is a measure of the synchronization degree between an agent and its coordinator. The larger the uncertainty gap, the older the data that are considered by y. The ideal situation is that y processes x's data as soon as it receives them; the existence of a time interval between the reception of data by y and its correlation task implies that if x had communicated its data later, then y would have correlated more recent data. Therefore, to minimize the uncertainty gap, every agent should perform its tasks as late as possible (to capture the most recent data from the environment) while meeting its deadline.

In the remainder of this section, two alternative synchronization strategies are explained and then a description of how different agents interact across the various layers is described.

3.1.1. Synchronizing readings vs. communications

Synchronizing the agents implies the determination of the time when they must perform their tasks, and thus be able to cooperate among themselves. Between the following two possible synchronization strategies (summarized in Table 1, with $\surd$ indicating a positive feature and χ indicating a negative one) one could be theoretically chosen:

- *Synchronizing the readings* – with this approach, the helping agents created by a certain coordinator agent would synchronize the time instant at which they obtain new values from the environment (the *starting deadlines*), as shown in Fig. 2a. In the figure, two helping agents and their coordinator agent are shown, and the starting deadline of each agent is indicated. The coordinator agent sets the time instant at which its helping agents should start their tasks by taking into account the different delays of its helping agents. This approach aims at achieving *consistent snapshots* of the environment, as all the data are obtained at approximately the same time. This is particularly important, for example, in applications relying on data fusion techniques [34], as an inconsistent snapshot would lead to erroneous deductions. Though this approach provides reading consistency, *it increases the uncertainty gap* and, therefore, the *age of the data* returned to the coordinator agent. Moreover, the coordinator agent must keep track of *the delays of all its helping agents* and *adjust their deadlines* when *any delay* changes. Finally, even if a coordinator is able to synchronize the readings of all its helping agents, to assume that it will be possible to coordinate the readings of all the agents in the network (or even just of all the agents in a layer) is very challenging (a global coordination mechanism would be needed, which should be very sensitive to the changing delays of any agent). Therefore, full data consistency cannot be achieved.
- *Synchronizing the communications* – with this approach, agents synchronize the time instant at which they finish communicating their results to their coordinator agent (the *ending deadlines*), as shown in Fig. 2b. In the figure, two helping agents and their coordinator agent are depicted, and it can be seen how the ending deadline of each helping agent must be computed taking into account the starting deadline of the coordinator. The coordinator agent sets the time instant at which it should have received the results from its

Table 1
Synchronizing readings vs. synchronizing communications: a summary

Synchronize	Readings	Communications
Comply with	Starting deadlines	Ending deadlines
Benefit	$\surd$ Reading consistency...	$\surd$ Maximum freshness
	χ ...but only under the same coordinator	
Keep track of	χ Delays of all helping agents	$\surd$ Only its own delays
Adjustment triggered by	χ Any helping agent	$\surd$ Only the coordinator

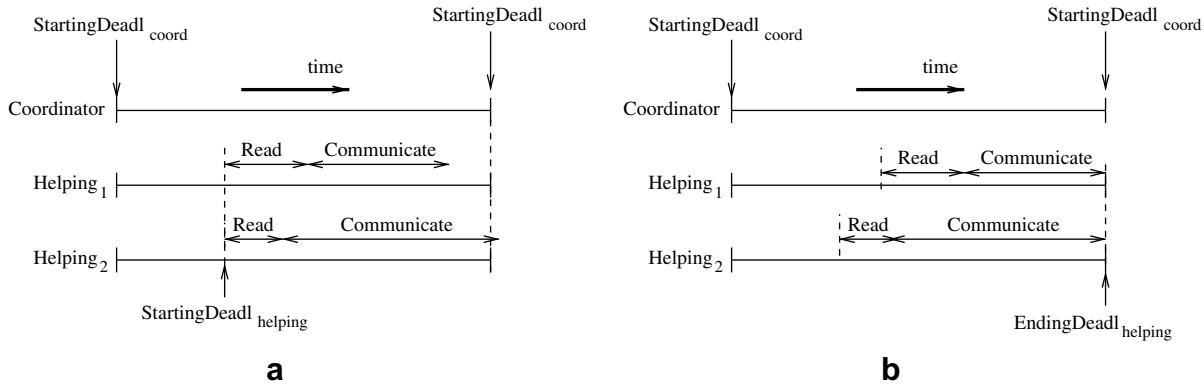

Fig. 2. Synchronization reference: (a) synchronizing the readings or (b) synchronizing the communications.

helping agents by taking into account its own deadline and delays. This approach aims at *minimizing the uncertainty gap* of the received data: agents read values from the environment as late as possible while meeting their deadlines. Another advantage is that the coordinator only needs to communicate a new deadline to its helping agents if there is a significant change in *its own delays*, as each helping agent adjusts itself to finish its tasks in time. The disadvantage is that the results returned to the coordinator can be based on *data read at different time instants*. As the tasks are started as late as possible, this approach could be more sensitive to unexpected delays; to avoid this, agents can start their tasks slightly before the time instant when they estimate they should (see Section 3.2.3).

Therefore, a synchronization based on the ending of tasks (synchronizing the communications) is preferable because it minimizes the average uncertainty gap (i.e., it maximizes the data freshness), and the alternate strategy does not ensure full data consistency either. However, it should be stressed that *the proposed synchronization approach is valid for both cases with minor modifications*: only the interpretation of deadlines would change (starting vs. ending time).

3.1.2. Synchronization across agent layers

To ensure that all the agents perform their tasks in time, agents at a certain layer should have communicated their results to their coordinator before a certain time instant (deadline) assigned to them by the coordinator. Such a coordinator will similarly correlate the received data and communicate its results to its coordinator agent according to its own deadline. Therefore, agents interact in two possible ways:

- *By communicating new results* to their coordinator agent, which will process the data received from its helping agents and obtain a result to communicate to its own coordinator.
- *By communicating new deadlines* to its helping agents. In this manner, an agent establishes the deadline by which time it should have received new results from its helping agents. An agent may need to readjust the deadlines of its helping agents when the delays change (see Section 3.3).

In Fig. 3, the interaction of agents across the different layers is shown. A notation similar to the one used in the Unified Modeling Language (UML) sequence diagrams is used in this article: each vertical line represents the life of a different object (in this case, a different agent) and the exchange of messages is represented herein. It should, however, be noted that the UML notation is not followed strictly, as for example, a change in the slope of arrows indicates the communication delay between the agents. In the following, the interactions regarding an intermediate agent at layer i, $Agent_i$, created by a certain $Agent_{(i-1)}$ at layer $(i-1)$, are described. The main steps to consider while managing deadlines are as follows: (1) $Agent_{(i-1)}$ communicates to $Agent_i$ the deadline for layer i; (2) $Agent_i$ receives its deadline and calculates the deadline of agents at layer $(i+1)$ by considering its own deadline and its estimated correlation and communication delays; (3) $Agent_i$ communicates to

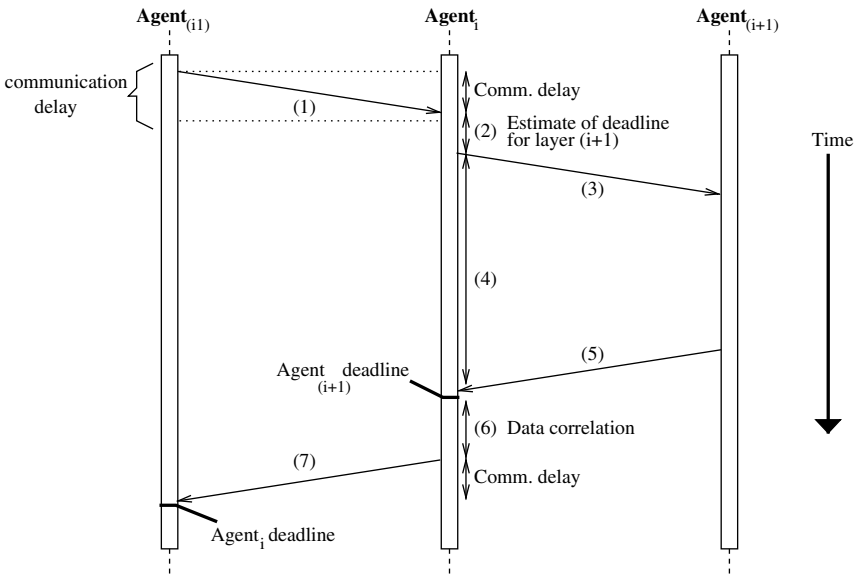

Fig. 3. Interactions among agent layers along time.

agents at layer $(i+1)$ their deadlines; (4) $Agent_i$ waits until the deadline of layer $(i+1)$ arrives as, to meet its own deadline, $Agent_i$ must begin correlating the received results right after they are obtained from layer $(i+1)$; (5) each $Agent_{(i+1)}$ sends its result to $Agent_i$; (6) $Agent_i$ correlates the results received from its helping agents, and (7) $Agent_i$ sends its result to $Agent_{(i-1)}$. Agents at the bottom layer of the hierarchy do not receive results from any agent; instead, they obtain data directly from the environment. Thus, the proposal described in this article relies on agents that behave responsibly (they comply with their deadlines) but do their tasks as late as possible (so as to consider the most recent data). As a result of all the agents in the network behaving in this manner, an emergent property is observed: the result of the monitoring (obtained by the root agent) is based on the latest data that can be obtained from the environment. To obtain even more recent data, an agent should start its tasks later: this would lead the agent to miss its deadline.

3.2. Computation of deadlines

In this section, the details about how deadlines are computed and handled by the agents are discussed. The traditional concept of absolute deadline is first considered. Then, the concept of relative deadlines that avoid the need for having the internal clocks of the involved computers and devices synchronized is introduced. Finally, how an agent determines, from the computed deadlines, the precise time to start its correlation and communication tasks is determined. In this section, it is assumed that the required task period is not smaller than the task delays experienced by the agents, and in Section 4, the strategies that the system can adopt when this assumption does not hold good are explained. The notation that will be used throughout the section is summarized in Table 2; the layer where an agent is executing is indicated only when it is relevant.

3.2.1. Absolute deadlines

Helping agents must finish their tasks at a certain time instant set by their coordinator so that it can meet its own deadline. This time instant is called an *absolute deadline* (see Definition 2). Absolute deadlines are coded as $\langle date, hour \rangle$. To compute an absolute deadline, the coordinator agent must be aware of the time it needs to perform its tasks (see Definition 3).

Definition 2. The absolute deadline of an agent y at layer $(i+1)$ is the time instant when it should have made its result available to its coordinator x at layer i. It is given by the following equation:

$$absDeadl_{y@i+1} = absDeadl_{x@i} - \widehat{taskD}_{x@i} - securityMargin,$$

Table 2
Summary of the notations used

Notation	Meaning
$absDeadl_x$ ($absDeadl_{x@i}$)	Absolute deadline of agent x (x at layer i: $x@i$)
$taskD_x$ ($taskD_{x@i}$), $\widehat{taskD}_x$ ($\widehat{taskD}_{x@i}$)	Task delay of agent x: real and estimated (x at layer i: $x@i$)
$securityMargin$	Security margin for the computation of the spare time
$correlD_x$	Correlation delay of agent x
$resComD_x$	Result communication delay of agent x
$relDeadl_x(t)$, $\widehat{relDeadl}_x(t)$	Relative deadline of agent x, relative to time t: real and estimated
$tDeadlEst_x$	Time instant when x starts estimating the relative deadline of a helping agent
$deadlCommD_x$, $\widehat{deadlCommD}_x$	Time needed by x to estimate and comm. a new deadline: real and estimated
$startTimeTravel_x$	Time instant when agent x starts a trip
$travelDelay$, $\widehat{travelDelay}$	Time needed for an agent to travel to another computer/device: real and estimated
t, $\hat{t}$	Time instant when a relative deadline is received: real and estimated
$spareT_x(t)$	Spare time of agent x at time t
T_{req}	Required task period

where $taskD_{x@i}$ denotes the task delay of an agent x at layer i (see Definition 3), and the symbol $\widehat{\ }$ is used to denote the estimates of values. A value $securityMargin$ is subtracted because agent x could start its task slightly earlier than necessary (see Section 3.2.3).

The absolute deadline of an agent is computed from the absolute deadline of its coordinator. The root agent, which does not have a coordinator, initially sets its absolute deadline by just considering the current time instant and an estimate of the time needed to get the first result from its helping agents. It should be stressed at this point that assuming a hierarchical cooperation structure is not a limitation of this approach. Thus, in case an agent has to send its results to several receiving agents (i.e., there are several coordinators), the minimum of the deadlines of all the coordinators should be taken into account, which would lead to an unavoidable unsynchronization between the sender agent and the other receiving agents. However, a better approach would be for the sender agent to have several threads for these tasks, which is equivalent to having several sender agents. This hierarchical structure would lead to a better and easier design.

Definition 3. The task delay of an agent x is the time it spends performing the tasks of correlating the results received from its helping agents (maybe using also other data that it obtains directly from the environment) and communicating its own result to its coordinator agent:

$$taskD_x = correlD_x + resComD_x.$$

Agents at the same layer can experience different delays: they could execute their tasks on different computers and/or perform different correlation tasks. In addition, the delays can change along time, so the agents keep track of the delays which they incur to obtain estimates of any future delay (see Section 3.3).

3.2.2. Relative deadlines

As the agents in the monitoring system could execute their tasks on different computers and move from one computer to another, dealing with absolute deadlines would require that the internal clocks of these computers be perfectly synchronized. Therefore, in this approach, deadlines communicated among layers/computers are relative (e.g., "I want your result in ten seconds") rather than absolute (e.g., "I want your result ready at 13:05:20"). Relative deadlines are modeled as long values, indicating the milliseconds until the deadline. Definition 4 shows how *relative deadlines* are computed.

Definition 4. The relative deadline of a certain agent x, relative to a time instant t, is the time interval that remains at t before the next absolute deadline of x comes:

$$relDeadl_x(t) = absDeadl_x - t$$

A relative deadline is a way of representing an absolute deadline that does not depend on the internal clock of the computer where the agent executes. Thus, an agent needs to convert an absolute deadline into a relative deadline whenever the deadline must be transmitted to another computer, which occurs in any one of these two situations:

(1) *When an agent x needs to communicate a new deadline to its helping agents*: In this case, x will compute the relative deadline of its helping agents, which will be relative to the time instant $\hat{t}$ at which each helping agent is expected to receive such a relative deadline. The value of $\hat{t}$ is given by the equation:

$$\hat{t} = tDeadlEst_x + \widehat{deadlCommD_x},$$

where $tDeadlEst_x$ denotes the time instant when agent x starts estimating the relative deadline of the helping agent and $deadlCommD_x$ is the time needed by agent x in estimating and communicating the new deadline.

(2) *When an agent x is going to travel to another computer*: In this case, the agent will transform its absolute deadline into a relative deadline before traveling. The relative deadline obtained will be relative to the time instant $\hat{t}$ at which the agent expects to finish its journey:

$$\hat{t} = startTimeTravel_x + \widehat{travelDelay},$$

where $startTimeTravel_x$ denotes the time instant when agent x starts its trip and *travelDelay* is the travel delay. The actual *travelDelay* is computed as the difference between the time instants at the origin and destination, using in both cases the clock of the same computer as a reference.

An error in the estimation of *travelDelay* or $deadlCommD_x$ can be detected by the agent once the trip or the communication finishes, respectively, so that the agent can correct its mistake by adjusting the computed relative deadline (and, in the second case, communicating a *relative deadline correcting offset* to its helping agents); therefore, it can be assumed that $\hat{t} = t$. In any case, once the relative deadline has been transmitted to the target computer at time t, the corresponding agent x transforms such a relative deadline back into an absolute deadline:

$$absDeadl_x = relDeadl_x(t) + t$$

In this manner, by comparing the current time instant with the corresponding absolute deadline, the agent is able to detect the amount of time left before its deadline (see Section 3.2.3). In Section 5.2, it is shown experimentally that by dealing with relative deadlines, the imprecision caused by having the clocks of the computers/devices involved unsynchronized can be avoided.

3.2.3. Spare time

An agent must decide when it should perform its correlation and communication tasks. If it starts too late, it will miss its deadline; if it starts too early, it could process data older than necessary. The amount of time the agent should wait is called the *spare time* (see Definition 5).

Definition 5. The spare time of an agent x at time instant t is defined as the amount of time that the agent should wait before performing its tasks, so as to correlate data as recent as possible. It is given by the following equation:

$$spareT_x(t) = absDeadl_x - t - \widehat{taskD_x} - securityMargin,$$

where *securityMargin* is a time interval that can be additionally subtracted to prevent agents from missing their deadlines due to either (unexpected) slight delay increases or being awoken a little bit late by the scheduler of the operating system.[2]

Once agent x has performed its task for its deadline $absDeadl_x$, it computes its next deadline by adding the required *task period* T_{req} (time interval between two monitoring tasks) to its previous deadline. The required task period may be set by the end user depending on the monitoring requirements and the wireless costs he/she

[2] The synchronization approach described herein could benefit from the existence of a real-time operating system (and/or a real-time programming language, such as *Java Real-Time*), which ensures that agents awake in time. Several policies have been proposed to schedule tasks based on their deadlines, such as *Earliest Deadline first*, *Least Slack first*, or *Rate Monotonic Scheduling* (e.g., see [27]); however, the monitoring agents must not perform their tasks very early, as in that case they would retrieve information that could become obsolete more easily.

might want to assume; alternatively, the agent network can compute the maximum task period supported by the network of agents and adjust dynamically to that period (using an approach similar to that explained in Section 4.1). In Fig. 4, the basic algorithm for the agents in the monitoring application is depicted.

3.3. Dynamic deadline adjustment

As explained in the previous section, the estimate of deadlines is based on parameters that change over time: the correlation and communication delays. So, the correlation tasks may become more or less time-expensive, as the amount of data obtained from the helping agents will change and the computer load can vary. Similarly, communication delays among agents can change due to many reasons (the available bandwidth, network failures, the amount of data sent, agents that move to different computers, etc.).

Therefore, after communicating the respective results to the upper layer, each agent recalculates the deadline of its lower layer taking into consideration its new task-delay estimate. Whenever a significant change happens, the new deadline will be communicated to agents at the lower layer to make data from the lower layer be obtained later/earlier and be able to process them later/earlier, according to the new delay estimate. The agent whose deadline is adjusted could also need to communicate new deadlines to its own lower layer recursively. In this manner, *the network of agents dynamically adapts its behavior to fulfill the requested task frequency by considering the current environmental conditions*.

In Fig. 5, an example about how deadlines are automatically adjusted is represented. In (1), the communication of the results from $Agent_i$ to $Agent_{(i-1)}$ is delayed (due to a lower network speed or network instability that implies communication retries), and thus $Agent_i$ misses the deadline (2). At time instant (3), $Agent_i$ realizes that it cannot fulfill the next deadline either (4), as the result communication should have begun before the new delays were estimated (dotted arrow in the figure). As the next deadline that can be achieved is (5), a new deadline for layer $(i+1)$ is computed (considering the new delays) and communicated to $Agent_{(i+1)}$ (6), which for-

```
Algorithm MonitoringAgent
Require: initialDeadline is the initial deadline of the agent, T_req is the required task
  period, and estDelay is the estimated delay of the agent's tasks
Ensure: a task will be performed every T_req, providing results as up-to-date
  as possible within the given time constraints (required task frequency)
 1 : securityMargin ⇐ ...; {For example, 100 ms}
 2 : deadline ⇐ initialDeadline;
 3 : setTaskPeriod(T_req);
 4 : while ¬ end do
 5 :    spareTime ⇐ getSpareTime(); {See Section 3.2.3}
 6 :    if spareTime < 0 then
 7 :       deadline ⇐ deadline + getTaskPeriod();
 8 :    else
 9 :       sleep(spareTime);
10 :       correlationAndCommunicationTask();
11 :       prevEstDelay ⇐ estDelay;
12 :       estDelay ⇐ estimateTaskDelay();
13 :       deadline ⇐ adjustDeadlIfNeeded(deadline, estDelay, prevEstDelay); {See Section 3.3}
14 :       deadline ⇐ deadline + getTaskPeriod();
15 :    end if
16: end while
```

Fig. 4. Algorithm for the agents in the monitoring application.

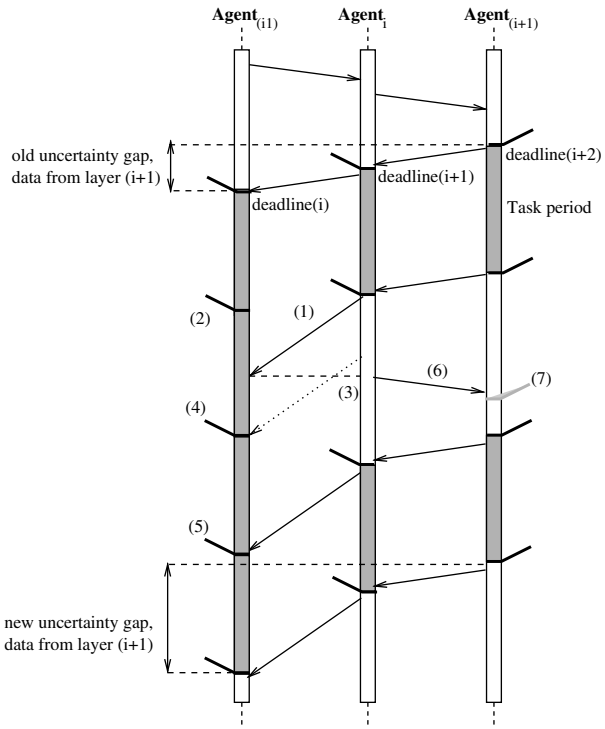

Fig. 5. Dynamic adaptation to new delays.

gets its old deadline (7). Since then, everything works fine as the system has adapted to the new delays. The requested task frequency is supported again at the expense of a longer uncertainty gap, which is unavoidable due to the slower communication between layer i and $(i-1)$.

Several techniques can be used to predict future delays, such as the method proposed in [46], linear extrapolation, double exponential smoothing [24], artificial neural networks [3], (simple or exponential) moving averages [29], the Kalman filter [17], or the bandwidth-estimate method in [20] (for network delays). In different situations, different choices could be preferable so as to react faster/slower to the delay changes. The adaptability of mobile agents to the changing network conditions has also been studied in the context of mobile agent (dynamic) planning for distributed information retrieval [1], in which situation an agent must choose the most efficient itinerary and adapt it dynamically if the conditions change.

4. Dealing with too high delays

The environmental conditions could become so difficult that some agents could be unable to return new data to their coordinator at the requested frequency. This happens when the time needed by an agent to perform its correlation and communication tasks is greater than the task period because either the required task frequency or the agent's task delay is too high. Thus, an agent cannot provide data faster than it can obtain them: the maximum frequency at which an agent can operate is limited by its task delay. Different agents are subject to different delays and therefore have different maximum frequencies. If the helping agents return data at different frequencies (they are not synchronized), then the data correlated by its coordinator will be a mixture of data obtained at different moments, which can lead to a loss of quality in the data presented to the user: it could be a snapshot that did not happen in the past. So, the most desirable situation would be to provide data at a frequency as high as possible with a minimum unsynchronization. In the following sections several approaches to deal with this problem are described.

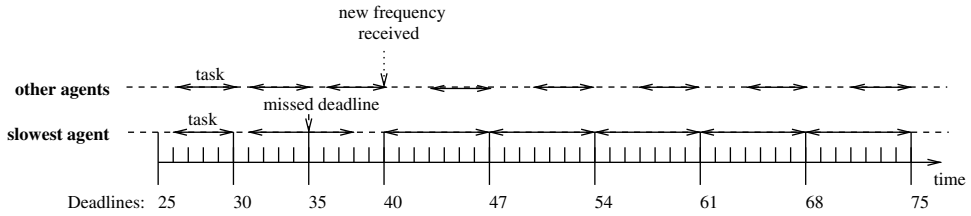

Fig. 6. Too high delays: synchronization with the slowest agent.

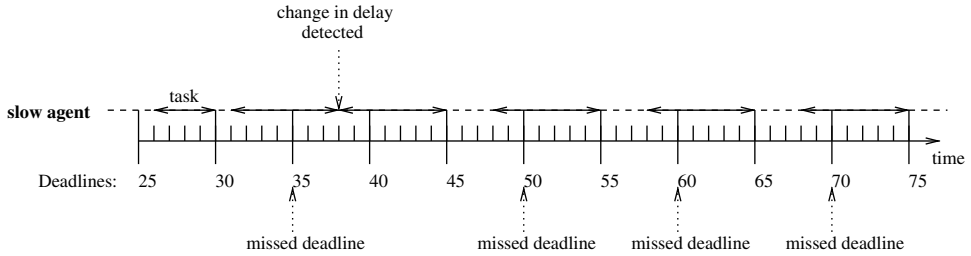

Fig. 7. Too high delays: ignoring the slowest agent.

4.1. First approach: synchronization with the slowest agent

This strategy adjusts the frequency of all the agents in the network to the maximum frequency supported by the slowest agent. With that purpose, every agent reports to its coordinator about the highest frequency supported by all its helping agents and itself (to save costs, along with the results). Thus, the root agent obtains the lowest frequency of the whole agent network and adjusts its helping agents to that frequency, which will in turn propagate the new frequency down the network of agents recursively.[3]

The behavior of the agents when this strategy is applied in a scenario with a task period of 5 s is shown in Fig. 6; deadlines are labeled with time annotations (in seconds) and an increase in delays occurs at time instant 31. The slowest agent misses one deadline, and then the period is adjusted to 7 s, so that all the agents meet their deadlines after the synchronization.

With this approach, it is necessary to keep track of the maximum task frequencies supported by the agents (each agent obtains the maximum frequency for its helping agents and propagates this information upwards). If the root agent detects that all the agents can again support the requested frequency (i.e., if the required task frequency is not higher than the maximum task frequency supported by the helping agents), then all the agents adjust themselves again (under request from the root agent) to that frequency. A similar adjustment occurs if, despite the existence of *slow agents*, the network is now able to support a frequency higher than before.

4.2. Second approach: to ignore the slowest agent

Another solution to the problem of too high delays is just to ignore that one of the agents cannot support the required frequency. That particular agent is allowed to run *unsynchronizely* with respect to the remaining agents, although it will miss some of its deadlines (see Fig. 7).

In this approach an agent does not try to meet deadlines that it is going to miss (e.g., in the example, it does not start a task at time instant 45). The percentage of missed deadlines is given by $(1 - 1/\lceil \Delta/T_{\text{req}} \rceil) \times 100$, where Δ is the task delay and T_{req} is the required task period.

[3] This could also happen in other circumstances. For example, the user could increase/decrease the frequency of the monitoring task at any time.

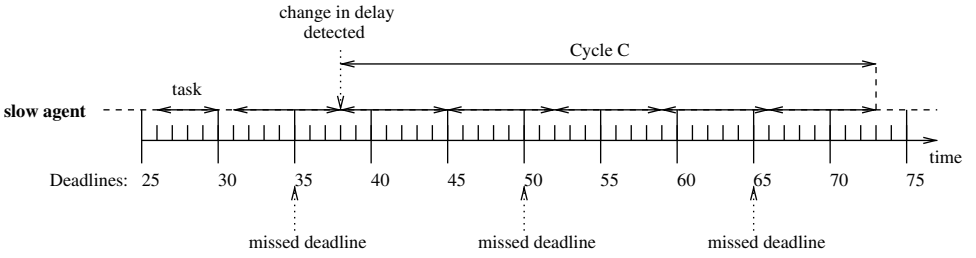

Fig. 8. Too high delays: no spare time.

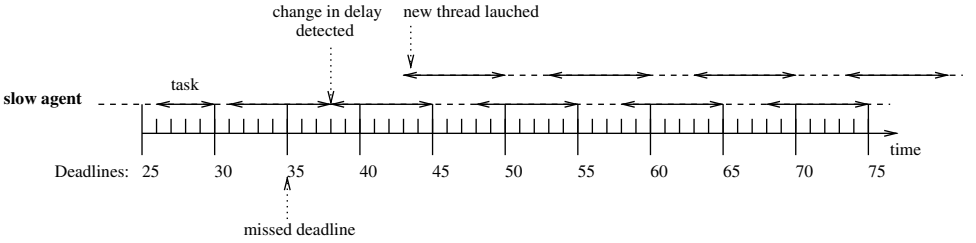

Fig. 9. Too high delays: multithreading approach.

4.3. Third approach: no spare time

This strategy assumes that it makes no sense to allow spare time when an agent cannot support the required frequency. So, after detecting that the new delays are greater than the task period, the agent could start the tasks for its next deadline immediately, as shown in Fig. 8. With this approach, the average percentage of deadlines missed by an agent during a time interval T' is $\frac{100\times N_{\text{missed}}}{\lfloor \frac{T'}{T_{\text{req}}} \rfloor}$, where $N_{\text{missed}} = \lfloor \frac{T'}{T_{\text{req}}} \rfloor - \lfloor \frac{T'}{\Delta} \rfloor$. It should be noted that the uncertainty gap changes continually for every task period; therefore, T' in the previous formulae must be selected to cover a whole cycle of behavior C (e.g., $T' = \Delta \times T_{\text{req}}$).

4.4. Fourth approach: multithreading

This strategy considers that, if needed, an agent can perform several tasks concurrently so that it can meet its deadlines, by using several threads of execution (see Fig. 9). The number of threads used by an agent with this strategy is $\lceil \Delta / T_{\text{req}} \rceil$.

Thus, in theory, the multithreading approach is the ideal solution as it allows any frequency to be supported by using additional threads, no matter the delays.[4] However, some limitations arise when the number of threads increases, as this strategy can overload the computer if *many* threads are needed.

4.5. Summary of the approaches considered for dealing with too high delays

The algorithm in Fig. 4 should be therefore modified to add extra behavior to deal with too high delays, depending on the approach used:

- *To ignore the slowest agent* – In Section 3, where the algorithm in Fig. 4 appears, the possibility of too high delays is ignored. Therefore, no extra code is needed to implement this strategy.

[4] A parallelism can be found in nature: the Hubble telescope allows us to see *continuously* the light of the stars, although with unavoidable delays of up to 10 million years!

Table 3
Advantages and disadvantages of the strategies to deal with too high delays

Strategy	Advantage	Disadvantage
Synchronization with the slowest agent	Consistent snapshots	Poor data refreshment frequency
To ignore the slowest agent	Quick agents are not slowed down	Inconsistent snapshots Missed deadlines
No spare time	Quick agents are not slowed down Less missed deadlines	Inconsistent snapshots Variable uncertainty gap
Multithreading	Required task frequency Consistent snapshots	Overloading in critical situations

Table 4
Selection of the strategy to deal with too high delays

Strategy	Too high delays?	Computer overloaded?	Consistency preferred over data refreshment rate?
To ignore the slowest agent	No		
No spare time	Yes	Yes	No
Synchronization with the slowest agent	Yes	Yes	Yes
Multithreading	Yes	No	

- *No spare time* – With this strategy, the call to *getSpareTime*( ) in line 5 of the algorithm in Fig. 4 must return 0 if the agent is *slow* (i.e., if its task delay is greater than the required task period), which implies that no sleep will be performed in line 9.
- *Synchronization with the slowest agent* – In the communication task that takes place when calling *correlationAndCommunicationTask* in line 10 of the algorithm in Fig. 4, the agent communicates not only its results but also the minimum task period supported by all its helping agents and itself (the agent is informed of the period supported by a certain helping agent when it receives the results from it). In this manner, the root agent obtains the minimum task period supported by all the agents and (when it is necessary) propagates this task period down the system, which is adopted by all the agents (a call to *setTaskPeriod*( ) is performed with the received period as argument and so the call to *getTaskPeriod*( ) in lines 7 and 14 of the algorithm in Fig. 4 will return the new task period).
- *Multithreading* – In this case, if an agent is *slow* then it performs the correlation and communication tasks (line 10 of the algorithm in Fig. 4) using an auxiliary thread.

Each of the four approaches that have been described have different advantages and disadvantages, which are summarized in Table 3. According to these properties, in Table 4 are shown the conditions under which an agent dynamically selects the strategy to apply for dealing with too high delays. If the required task frequency can be supported, there is nothing particular to do, and so the strategy selected is to ignore the slowest agent. In case there are too high delays, the multithreading strategy is initially chosen. However, if the agent detects that the computer is overloaded, then it will switch to another strategy: *synchronization with the slowest agent* or *to ignore the slowest agent*, depending on whether the monitoring application prefers data consistency (data comparable, obtained at approximately the same time) or a low task period (data refreshed at the highest rate), respectively. If the computer stops being overloaded, then the multithreading strategy will be used again. Different agents could experience too high delays and yet decide on different strategies. It is not possible that one agent selects the *no spare time* strategy, and another agent, the *synchronization with the slowest agent* strategy, as both are exclusive (the choice depends, as has been explained, on the requirements of the monitoring application). Moreover, there is no problem if an agent selects the *no spare time* strategy and another agent selects the *multithreading* strategy, as both are strategies that only affect the behavior of the agent with too high delays. The *synchronization with the slowest agent* strategy triggers a change in the task frequency of all the agents in the network. There is no conflict, however, if another agent selects *multithreading*: the *synchronization with the slowest agent* will make the root agent to propagate a new task frequency and that agent will probably stop needing auxiliary threads. In this case, if the agent that selected the *synchronization with the*

slowest agent strategy can switch later to *multithreading*, it will report to its coordinator that it can again support the required task frequency and switch to *multithreading* if this report causes the root agent to propagate a new frequency. The whole network of agents adapts to any new situation dynamically.

5. Experimental evaluation

In this section, some tests that were carried out to evaluate the feasibility and suitability of the above synchronization approach are described, considering the design and implementation of a real multiagent system that follows the proposed guidelines to build monitoring systems. First, the architecture of the multiagent system that is considered is described. Second, the use of relative deadlines to effectively deal with unsynchronized computer clocks is evaluated. Third, the convenience of adjusting the deadlines of agents in the network when the respective conditions (i.e., delays) change is evaluated. Finally, the multithreading strategy that has been proposed to deal with situations where some agents cannot provide results with the requested task frequency is evaluated. To predict delays, a Kalman filter with the parameters described in [20] is used, although any other alternative could have also been logically considered.

5.1. A multiagent system for the processing of continuous location-dependent queries

For experimental evaluation, LOQOMOTION [15], a system for the processing of continuous location-dependent queries issued by mobile devices (e.g., portable computers with wireless connection) is considered. As an example of *location-dependent query*, consider the example of retrieving the police units (police stations, policemen, and police cars) within seven miles around *car38* (a stolen car) and the police cars within five miles from *policeCar5* (the current chasing police car). The goal of the query processing is twofold: (1) to continuously update on the user device the set of moving objects that satisfy the query constraints, and (2) to show the current location of each retrieved moving object on a map. The query processing is performed over a set of computers (*proxies*) which manage the location data of objects moving within different geographic areas.

In the architecture of LOQOMOTION, based on mobile agents, there is a *QueryMonitor* agent on the user device, which is in charge of presenting the answer to the user. This agent delegates the continuous processing of the query to a *MonitorTracker* agent. The MonitorTracker, which represents the QueryMonitor on the fixed network, creates a network of *Tracker* agents for tracking the location of the objects that are referenced in the query (called the *reference objects*, car38 and policeCar5 in the sample query). Each Tracker tracks a reference object and creates a network of *Updater* agents in charge of tracking the location of moving objects that are within a certain geographical area around that reference object. The MonitorTracker and the Trackers are mobile agents which travel among computers to optimize the performance (they keep themselves "close" to the user and the data they monitor, respectively). Periodically, each agent in the network correlates the results received from the agents that it created, and returns the results to its coordinator agent, until the final answer is obtained by the QueryMonitor.

According to this monitoring approach, LOQOMOTION is based on a distributed and divide-and-conquer cooperation structure with four layers of agents (QueryMonitor, MonitorTracker, Trackers, and Updaters), which cooperate to obtain the answer to a continuous location-dependent query. Moreover, the answer depends on location data, that change constantly, and therefore must be refreshed efficiently with a certain frequency. The experiments that follow were carried out considering a scenario with several moving objects and six proxies that were simulated in four computers using the approach presented in [14]. In this scenario, the sample query presented at the beginning of this section was issued.

5.2. Dealing with clock unsynchronization

In this section, an analysis of how the use of relative deadlines (see Section 3.2.2) overcomes the inconvenience of clock unsynchronization is dealt with. In Fig. 10, a strategy based on the absolute vs. the relative deadlines is compared by showing the average location error for different values of the task period and varying the clock unsynchronization (the objects in this test move at 40 mph). In the figure, there are three axes: (1) the percentage of unsynchronization between the computer clock of the user device and the computer clocks of the

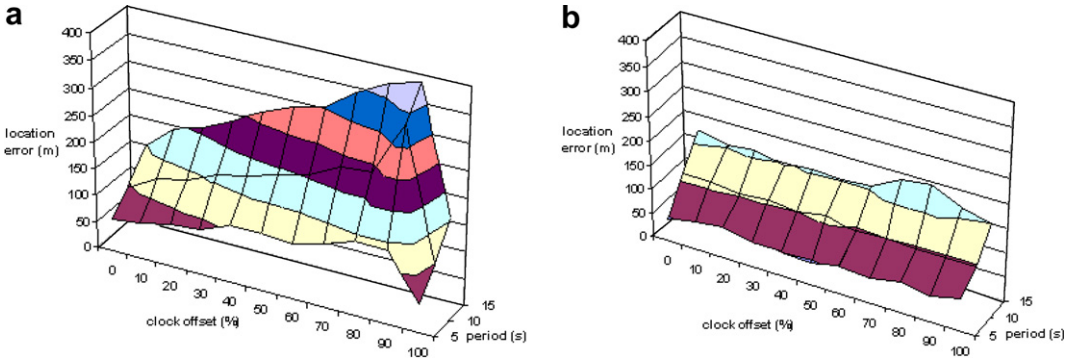

Fig. 10. Clock unsynchronization: (a) absolute deadlines or (b) relative deadlines.

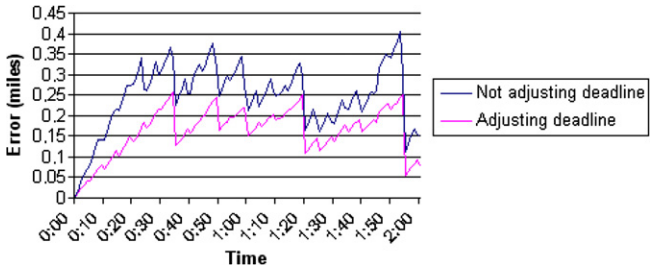

Fig. 11. Location error: convenience of adjusting deadlines.

proxies; (2) the task period (in seconds); and (3) the *average location error* (i.e., the average difference between the real locations of the objects and the locations shown to the user during the monitoring), which is the measured value. The clock offset is expressed as a percentage of the task period. For example, a clock offset of 50% with a 5-s period implies that the clock of the proxies will be 2.5 s ahead of the clock on the user device. This implies that the agents at the proxies will perform their tasks further in advance and, therefore, the age of the data returned will increase (they wrongly believe that it is later, so they start their tasks earlier).

As expected, the average error is greater as the task period increases, because the answer is updated at a lower rate. When there is no clock unsynchronization, the error committed is approximately the same, independent of whether absolute or relative deadlines are used. Furthermore, the error is not affected by the clock offset if relative deadlines are used. However, the error increases significantly with the clock offset in case the deadlines are absolute. It should be noted that the error decreases for a clock offset of 100% when absolute deadlines are considered: this is because an offset of 100% leads to the same unsynchronization as an offset of 0% with respect to its previous deadline.

5.3. Dynamic deadline adjustment

In this experiment, the convenience of adjusting the deadlines of the agents when the delays change (see Section 3.3) was evaluated quantitatively. It is simulated that Trackers experience an increase of 10 s in their communication delays and that the objects move at 60 mph. In Fig. 11 the average location error at each time instant is shown, both in the case of adjusting and not adjusting the deadlines. Minimum errors are obtained when the data at the user device are updated, and maximum errors, right before the update.[5] By adjusting the deadlines, the network of agents resynchronizes itself and the freshness of the data is maximized, which decreases the error.

[5] At 60 mph, an object travels 0.17 miles in 10 s, which explains the high location error just before the data are refreshed.

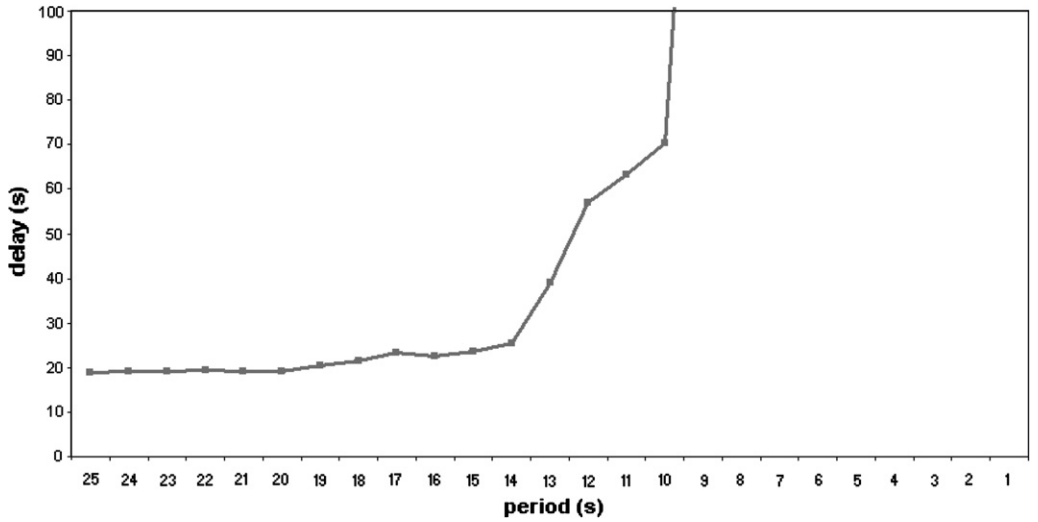

Fig. 12. Delays with the multithreading approach.

5.4. Strategies to deal with too high delays

In [13] the advantages and disadvantages of the four approaches proposed in Section 4 to deal with too high delays were compared experimentally, and it was shown that the multithreading approach achieves the best results. However, in those experiments the delays were simulated and, therefore, the *real* limitations of the multithreading approach could not be detected. To account for this situation, the tests that are explained in the following were carried out, which show that *the multithreading approach can help use the available bandwidth in a real environment*.

In this test, the QueryMonitor (which is the root agent in the monitoring architecture of LOQOMOTION) is placed on a Pentium III, 350 MHz, 128 MB RAM, with Windows 98, connected to the Internet through a 56Kbps modem. The remaining agents deploy themselves over the set of proxies available in the Local Area Network of our university, as in the previous experiments. A *real* bottleneck in this scenario is the modem communication between the QueryMonitor and the remaining agents in the network. It is simulated herein that the answer obtained by the MonitorTracker (and communicated to the QueryMonitor) takes up 200 KB.

In Figs. 12–14, a 3-min test to evaluate the impact of decreasing the task period (i.e., requiring a higher task frequency) is shown. As can be seen in Fig. 12, the delay increases as the task period decreases, which leads to a higher number of missed deadlines (see Fig. 13). This is due to a higher number of threads (see Fig. 14) competing for the network bandwidth. Hence, it is observed that the multithreading approach is considerably harmful for periods smaller than 9 s.[6] However, the *multithreading approach* is beneficial for a required period no smaller than 13 s, wherein the agent misses a 35.71% of the deadlines; applying the formulae appearing in Sections 4.2 and 4.3 for that period, an average of 50% missed deadlines with the *ignoring approach* and 31.58% missed deadlines with the *no spare time approach* is computed, although the no spare time approach cannot minimize the uncertainty gap. The approach that adjusts to the slowest agent does not miss deadlines, but the task period is increased (and so the required monitoring frequency is neglected).

In Fig. 15, a task period of 14 s is focused on and the test is run for the duration of 1 h, to show how the multithreading approach evolves over time (higher delays lead to more threads that could again increase the delays). The fluctuations in the figure are due to the variability of the performance of the modem, and the changes in the percentage of missed deadlines at the beginning occur because only a small number of deadlines are available to compute the average and, therefore, any missed/met deadline has a great impact on the

[6] The experienced delays for these periods are represented in Fig. 12 as points that do not fit the *Y*-scale: the delays of the tasks cannot be precisely measured because they take too long!

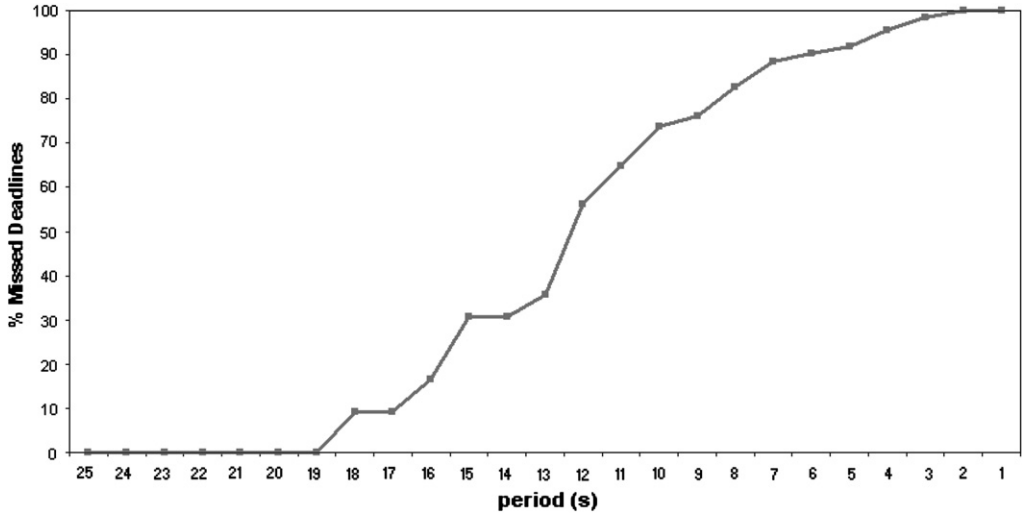

Fig. 13. Percentage of missed deadlines with the multithreading approach.

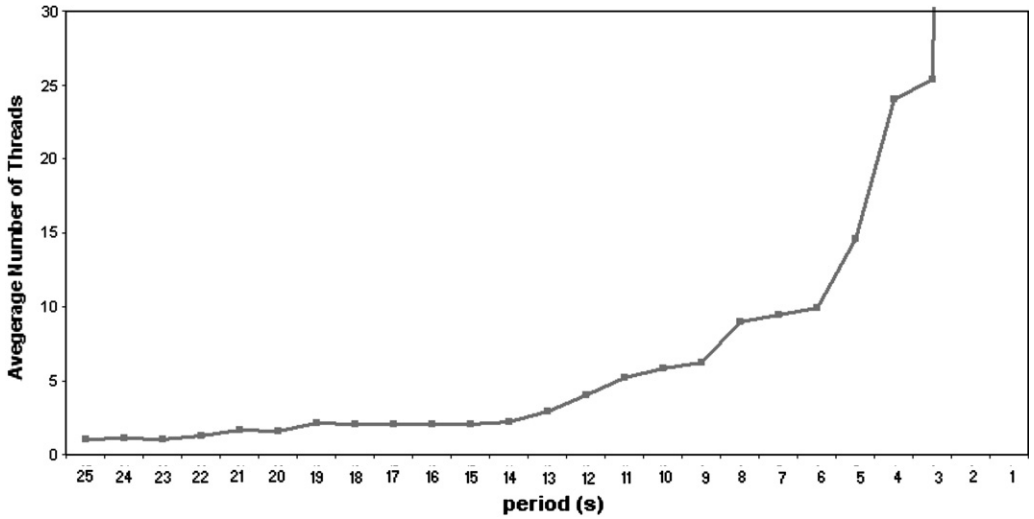

Fig. 14. Number of threads with the multithreading approach.

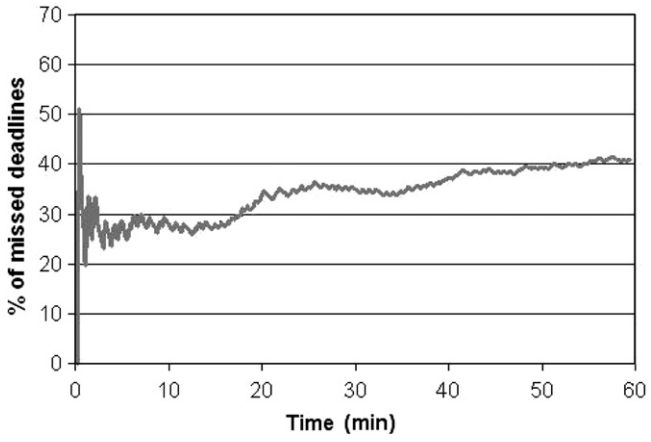

Fig. 15. Multithreading approach: transmitting 200 KB, 14-s task period.

average thus far. Experiments with other task periods were also carried out and it was concluded that the performance degrades slowly unless the required task period is already too challenging from the start.

Though the exact quantitative results must be considered with caution (modem communications are unreliable), similar behaviors with other data sizes and task frequencies have been observed. Hence, it can be concluded that by using multiple threads, frequencies higher than those allowed by the network delays can be supported (some other reports about concurrent communications also confirm this conclusion [28]). The multithreading approach is adequate in most cases, although launching new threads can also increase the delays.

6. Conclusions

In this article, an approach for continuously monitoring highly dynamic distributed environments using mobile agents has been presented. Our proposal considers a divide-and-conquer cooperation structure with agents that take into account the delays they experience and the environmental delays (i.e., processing and network delays). The main features of this approach are:

- *Mobile agents are used to perform the monitoring tasks efficiently and effectively*, thus reducing the network traffic and carrying the tasks to wherever the required data can be obtained with a good performance.
- *Agents adapt to the current situation* by taking into account the delays experienced by their cooperative agents and the environmental delays (i.e., correlation and network delays) for deciding when they should perform their tasks.
- The tasks are performed according to the *task frequency required by the user*. In addition, the agents try to finish their tasks in time but as late as possible, so as *to act upon the most recent data*.
- There is a loose coupling among the agents (based on *soft commitments*), which increases the *fault tolerance* of the system, as it leads to a "graceful degrading" of performance when some of the agents fail to perform their tasks in time.
- Relative deadlines are used, which ensures that this approach works well even if the *internal clocks of the computers/devices involved in the monitoring are unsynchronized* (as expected when many computers are involved).
- Several approaches can be used to deal with situations where some agents cannot perform their tasks at the required frequency, due to *challenging environmental conditions* (e.g., too high communication delays and/or required frequency).
- *Monitoring tasks are performed in wired computers* whenever it is possible. Thus, wireless devices are not overloaded with monitoring and communication tasks.

Moreover, the feasibility and practical interest of the presented approach in a real multiagent application have been detailed. An interesting line of future work would be to analyze the benefits of applying techniques of multiagent learning [44] in this context.

References

[1] J.-W. Baek, H.Y. Yeom, A timed mobile agent planning approach for distributed information retrieval in dynamic network environments, Information Sciences 176 (22) (2006) 3347–3378.
[2] N. Carver, A new framework for inference in distributed Bayesian networks for multi-agent sensor interpretation, in: B. Gupta (Ed.), 22nd Int. Conf. on Computers and Their Applications, CATA'07, ISCA, 2007, pp. 101–106.
[3] S. Crone, Stepwise selection of artificial neural network models for time series prediction, Journal of Intelligent Systems 14 (2–3) (2005) 99–122.
[4] P. Davidsson, M. Boman, Distributed monitoring and control of office buildings by embedded agents, Information Sciences 171 (4) (2005) 293–307.
[5] P. Davidsson, F. Wernstedt, A multi-agent system architecture for coordination of just-in-time production and distribution, in: ACM symposium on Applied computing, SAC'02, ACM Press, New York, NY, 2002, pp. 294–299.
[6] R. Davis, R.G. Smith, Negotiation as a metaphor for distributed problem solving, Artificial Intelligence 20 (1) (1983) 63–109.
[7] E.H. Durfee, V.R. Lesser, D.D. Corkill, Trends in cooperative distributed problem solving, IEEE Transactions on Knowledge and Data Engineering 1 (1) (1989) 63–83.

[8] E.H. Durfee, V.R. Lesser, Partial global planning: a coordination framework for distributed hypothesis formation, IEEE Transactions on Systems, Man, and Cybernetics 21 (5) (1991) 1167–1183.
[9] G. Vigna (Ed.), Mobile Agents and Security, Springer-Verlag, London, UK, 1999.
[10] R.S. Gray et al., in: G.P. Picco (Ed.), 5th IEEE Int. Conf. on Mobile Agents, MA'01, LNCS, vol. 2240, Springer-Verlag, London, UK, 2001, pp. 229–243.
[11] D. Hart, M. Tudoreanu, E. Kraemer, Mobile agents for monitoring distributed systems, in: P. Stone, G. Weiss (Eds.), 5th Int. Conf. on Autonomous agents, AGENTS'01, ACM Press, New York, NY, 2001, pp. 232–233.
[12] B. Horling, V.R. Lesser, R. Vincent, T. Wagner, The soft real-time agent control architecture, Autonomous Agents and Multi-Agent Systems 12 (1) (2006) 35–92.
[13] S. Ilarri, E. Mena, A. Illarramendi, Dealing with continuous location-dependent queries: just-in-time data refreshment, in: 1st IEEE Int. Conf. on Pervasive Computing and Communications, PerCom'03, IEEE Computer Society, Los Alamitos, CA, 2003, pp. 279–286.
[14] S. Ilarri, E. Mena, A. Illarramendi, Testing agent-based mobile computing applications using distributed simulations, in: 7th Int. DEXA Workshop on Mobility in Databases and Distributed Systems, MDDS'04, IEEE Computer Society, Los Alamitos, CA, 2004, pp. 652–656.
[15] S. Ilarri, E. Mena, A. Illarramendi, Location-dependent queries in mobile contexts: distributed processing using mobile agents, IEEE Transactions on Mobile Computing 5 (8) (2006) 1029–1043.
[16] S. Ilarri, R. Trillo, E. Mena, SPRINGS: a scalable platform for highly mobile agents in distributed computing environments, in: 4th Int. WoWMoM 2006 Workshop on Mobile Distributed Computing, MDC'06, IEEE Computer Society, Los Alamitos, CA, 2006, pp. 633–637.
[17] A. Jain, E.Y. Change, Y. Wang, Adaptive stream resource management using Kalman filters, in: G. Weikum, A.C. König, S. Dessloch (Eds.), ACM SIGMOD Int. Conf. on Management of Data, SIGMOD'04, ACM Press, New York, NY, 2004, pp. 11–22.
[18] Z. Juhasz, P. Paul, Scalability analysis of the contract net protocol, in: 2nd IEEE/ACM Int. Symposium on Cluster Computing and the Grid, CCGRID'02, IEEE Computer Society, Los Alamitos, CA, 2002, pp. 346–347.
[19] R.P. Kennedy, Monitoring of distributed processes with mobile agents, in: 7th IEEE Int. Conf. and Workshop on Engineering of Computer-Based Systems, ECBS'00, IEEE Computer Society, Los Alamitos, CA, 2000, pp. 205–210.
[20] M. Kim, B. Noble, Mobile network estimation, in: 7th Int. Conf. on Mobile Computing and Networking, MobiCom'01, ACM Press, New York, NY, 2001, pp. 298–309.
[21] F.J. Kurfess, D.P. Shah, K. Holthaus, F. Miralles, Monitoring distributed processes with intelligent agents, in: 6th IEEE Int. Conf. and Workshop on Engineering of Computer-Based Systems, ECBS'99, IEEE Computer Society, Los Alamitos, CA, 1999, pp. 196–202.
[22] K. Lam, A. Kwan, K. Ramamritham, RTMonitor: real-time data monitoring using mobile agent technologies, in: P.A. Bernstein, Y.E. Ioannidis, R. Ramakrishnan, D. Papadias (Eds.), 28th Int. Conf. on Very Large Data Bases, VLDB'02, Morgan Kaufmann, St. Louis, MO, 2002, pp. 1063–1066.
[23] D.B. Lange, M. Oshima, Seven good reasons for mobile agents, Communications of the ACM 42 (3) (1999) 88–89.
[24] J.J. LaViola, Double exponential smoothing: an alternative to Kalman filter-based predictive tracking, in: Workshop on Virtual environments 2003, EGVE'03, ACM Press, New York, NY, 2003, pp. 199–206.
[25] V.R. Lesser, Cooperative multiagent systems: a personal view of the state of the art, IEEE Transactions on Knowledge and Data Engineering 11 (1) (1999) 133–142.
[26] A. Liotta, G. Knight, G. Pavlou, On the performance and scalability of decentralized monitoring using mobile agents, in: Rolf Stadler, Burkhard Stiller (Eds.), 10th IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management, DSOM'99, Springer-Verlag, London, UK, 1999, pp. 3–18.
[27] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Readings in Hardware/Software Co-design (2001) 179–194.
[28] Y. Liu, W. Gong, P. Shenoy, On the impact of concurrent downloads, in: 33rd Winter Simulation Conference, WSC'01, IEEE Computer Society, Los Alamitos, CA, 2001, pp. 1300–1305.
[29] S. Lowry, The Magic of Moving Averages, Traders Press, Greenville, SC, 1998.
[30] S.S. Manvi, P. Venkataram, A method of network monitoring by mobile agents, in: A. Kumar, V.U. Reddy (Eds.), Int. Conf. on Communications, Control, and Signal Processing, CCSP'00, Viva Books, India, 2000, pp. 214–218.
[31] E. Mena, J.A. Royo, A. Illarramendi, A. Goñi, Adaptable software retrieval service for wireless environments based on mobile agents, in: C.-H. Yeh, S. Tekinay (Eds.), Int. Conf. on Wireless Networks, ICWN'02, CSREA Press, USA, 2002, pp. 116–124.
[32] D. Milojicic, F. Douglis, R. Wheeler, Mobility: Processes, Computers, and Agents, ACM Press, New York, NY, 1999.
[33] D. Milojicic et al., MASIF, the OMG mobile agent system interoperability facility, in: K. Rothermel, F. Hohl (Eds.), 2nd Int. Workshop on Mobile Agents, MA'98, LNCS, vol. 1477, Springer-Verlag, London, UK, 1999, pp. 50–67.
[34] H.B. Mitchell, Multi-Sensor Data Fusion: An Introduction, Springer-Verlag, 2007.
[35] M.J. O'Grady, G.M.P. O'Hare, Mobile devices and intelligent agents – towards a new generation of applications and services, Information Sciences 171 (4) (2005) 335–353.
[36] S. Rahimi, J. Bjursell, M. Paprzycki, M. Cobb, D. Ali, Performance evaluation of SDIAGENT, a multi-agent system for distributed fuzzy geospatial data conflation, Information Sciences 176 (9) (2006) 1175–1189.
[37] V. Roth, M. Jalali-Sohi, Concepts and architecture of a security-centric mobile agent server, in: 5th Int. Symposium on Autonomous Decentralized Systems, ISADS'01, IEEE Computer Society, Los Alamitos, CA, 2001, pp. 435–442.

[38] I. Satoh, Mobile agents for ambient intelligence, in: First Int. Workshop Massively Multi-Agent Systems I, MMAS'04, LNCS, vol. 3446, Springer-Verlag, London, UK, 2005, pp. 187–201.
[39] A. Selamat, H. Selamat, Analysis on the performance of mobile agents for query retrieval, Information Sciences 172 (3–4) (2005) 281–307.
[40] R.G. Smith, The contract net protocol: high-level communication and control in a distributed problem solver, Distributed Artificial Intelligence (1988) 357–366.
[41] C. Spyrou, G. Samaras, E. Pitoura, P. Evripidou, Mobile agents for wireless computing: the convergence of wireless computational models with mobile-agent technologies, Mobile Networks and Applications 9 (5) (2004) 517–528.
[42] K.P. Sycara, Multiagent compromise via negotiation, Distributed Artificial Intelligence 2 (1989) 119–137.
[43] O. Tomarchio, L. Vita, A. Puliafito, Active monitoring in grid environments using mobile agent technology, in: S. Hariri, C.A. Lee, C.S. Raghavendra (Eds.), 2nd Workshop on Active Middleware Services, AMS'00, Kluwer Academic Publishers, USA, 2000, pp. 57–66.
[44] K. Tuyls, P.J. Hoen, K. Verbeeck, S. Sen (Eds.), Learning and Adaption in Multi-Agent Systems: 1st Int. Workshop on Learning and Adaption in Multi-Agent Systems, LAMAS'05, LNCS, vol. 3898, Springer-Verlag, 2006.
[45] S.M.T. Yau, H.V. Leong, A. Si, Distributed agent environment: application and performance, Information Sciences 154 (1–2) (2003) 5–21.
[46] I. Yeom, A.L.N. Reddy, ENDE: an end-to-end network delay emulator tool for multimedia protocol development, Multimedia Tools and Applications 14 (3) (2001) 269–296.
[47] X. Zhang, V. Lesser, Meta-level coordination for solving negotiation chains in semi-cooperative multi-agent systems, in: 6th Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, AAMAS'07, ACM Press, New York, NY, 2007, pp. 50–57.