Abstract

In presence of reconfigurable environments, such as AmI systems, the orchestration and coordination of more devices involved in the execution of services requested by users is of paramount prominence. Effective solutions are based on planning techniques and multiagent technologies. Planning techniques find plans of actions to address users’ goals and allow the produced plans to be flexibly tailored to exploit the capabilities of the devices currently available. Literature focuses on the analysis of the most appropriate planner for AmI applications and leaves open the technological issues. In this paper we show the limitations due to the multiagent state-of-the-art technologies and we design a multiagent system for planning in AmI applications.

1 Introduction

Ambient Intelligence (AmI) “naturally” captures all the situations wherein computationally environments are populated by devices that exhibit some forms of intelligence. Many different forms of intelligence can be found in real-world applications. For instance, information concerning the status and the activities of the user can be inferred through sensors spread in the environment or the functioning of the devices can be orchestrated to provide compositions of services. Although the AmI paradigm is intrinsically heterogeneous, one principal aim in AmI is to make the interaction between users and smart environment easy. This is particularly prominent when the services requested by the users need the coordinated execution of several devices and these are mobile. An AmI system should be able to autonomously find what devices are to coordinate and how they can be coordinated to address the users’ requests. This problem can be reduced to a planning problem in multiagent domain, wherein a reconfigurable network of devices, each one with specific capabilities, can be employed firstly to build a plan of actions that address the user’s request and subsequently to execute such a plan by coordinating the devices. In [2] the authors study what is the planner most appropriate for AmI applications, proposing the D-HTN (Distributed Hierarchical Task Network) planner. Its peculiarities are: (a) the planning activity is carried on in centralized way by a single agent (since devices usually have low computational capabilities), (b) the knowledge concerning the task decompositions is distributed in the agents (since the network of agents is dynamical and a centralized knowledge cannot capture this dynamism).

The present paper aims at studying the employment of the multiagent state-of-the-art technologies to develop AmI applications based on D-HTN planners. Related works are the following. In [3, 4] the authors propose ad hoc middlewares for AmI systems able to take into account context awareness, whereas in [8] the authors employ standard multiagent technologies (e.g., FIPA protocols [6], RDF based ontologies [9], and JADE [7]) for the integration of the devices. Although these works make use of state-of-art technologies, they do not support planning activities. To the best of our knowledge, the unique multiagent system for AmI applications that supports planning activities was presented in [2]. However, the implementation discussed in [2] is not based on multiagent state-of-the-art technologies.

The original contributions we provide are: we show that the multiagent state-of-the-art is not fully satisfactory to support planning in AmI applications (particularly with mobile devices), we develop a multiagent system employing the state-of-the-art technologies, where they are suitable, and designing ad hoc solutions, otherwise. Finally, we provide a preliminary experimental evaluation of our proposal.

The paper is organized as follows. The next section discusses D-HTN planning. Section 3 presents our multiagent system. Section 4 experimentally evaluate our proposed system. Section 5 concludes the paper.

2 Planning in AmI Applications

2.1 D-HTN Planner

D-HTN [2] is an extension of HTN [5] planner to AmI. As is HTN, D-HTN is based on task networks, e.g. $[(n_1 : \alpha_1), (n_2 : \alpha_2), \ldots, (n_m : \alpha_m), \phi]$ where:
$\alpha_i$ are tasks, either primitive (that can be directly executed by an agent) or non-primitive (that must be further decomposed); $n_i$ are labels to distinguish different occurrences of the same task; $\phi$ is a Boolean formula representing the constraints on the tasks, such as variable bindings constraints and ordering constraints. An example can be: $[(n_1 : \text{CreateRequestText}(g_1, t_1)), (n_3 : \text{SendEmail}(t_1, a_1)), (n_1 \prec n_3) \land (n_2 \prec n_3) \land (n_1, \text{RequestText}(t_1), n_3) \land (n_2, \text{EmailAddress}(a_1), n_3)]$. The intended meaning is that, in order to request a good $g_1$ by email, we have to first create the RequestText $t_1$ and look for the EmailAddress $a_1$ of a supplier of $g_1$, then we have to SendEmail with content $t_1$ to $a_1$.

The functioning of a HTN planner is simple: it starts with an initial task network $D$ representing the problem (the goal) and with a set $M$ of methods or decompositions. Each decomposition is a pair $m = (t, d)$, where $t$ is a non-primitive task and $d$ is a task network; $m$ says that a way to achieve $t$ is to perform the tasks in $d$. Then, HTN planning proceeds by finding a non-primitive task $t$ from the current task network $D$ and a method $m = (t', d')$ in $M$ such that $t'$ unifies with $t$ and by replacing $t$ with $d'$ in $D$ (after appropriate substitution, see [5]). When only primitive tasks are left in $D$, a plan for the original problem can be found. A plan is a sequence of ground primitive tasks.

The D-HTN planner extends HTN as follows. 1) Each agent keeps a local data structure, called plan library, that stores all the decompositions it knows. The decompositions in the plan library of an agent have been defined by the designer during the installation of the agent and are peculiar for each agent. 2) The planner is the only entity supposed to have the computational power to generate a plan, the other agents are only requested to communicate decompositions (and to execute primitive tasks). 3) By means of a communication mechanism based on message passing: the planner can ask the currently connected agents to send their available decompositions for a given task, and the agents can send to the planner the requested decompositions.

Furthermore, the D-HTN planner allows one to design conditional plans, supporting the if-then and while constructs during the execution of the tasks, and characterizes plans according to three numerical indexes (i.e., performance, the cost, and the success probability). When more alternative plans are available for a given goal, the planner chooses the plan that maximizes the performance indexes.

### 2.2 Device Programming Paradigm

An effective architecture for devices was proposed in [2]: each device is controlled by a dedicated agent and this agent is splitted in a pair of semiagents: the cooperative semiagent (CO) and the operative semiagent (OP).

The CO accomplishes cooperative tasks, providing the connection to and the disconnection from the AmI system and the communication ability to exchange messages. These abilities are exploited in event management (the agent can generate an event, register itself to particular events, and receive the notification that a registered-to event happened) and in the goal management (the agent can generate goals, propose decompositions for a task, carry out an action to execute a plan, and update its plan library).

The OP is the operative and specific part of the agent. Basically, the OPs are the devices scattered in the environment. Their specific functions can be computational or interactive with the physical world. The functions performed by the OP constitute the bases for the primitive tasks of the agent. An OP can be implemented with different architectures and by different programming languages. By using the wrapper, CO can be executed remotely with respect to the OP since they communicate via remote method invocation. This is useful because usually the CO requires more computational power than the OP.

### 3 A System for Planning in AmI

#### 3.1 System Architecture and Solutions for Mobile Devices

The proposed system, named “domotic agency”, is based on LEAP [7] and implements the CO/OP paradigm.

In our mind, each environment (e.g., a house) is equipped by a domotic agency and each domotic agency runs on a different LEAP platform. Domotic agency’s architecture is simple: it is composed of an agent called major-domo that deals with the planning and execution activities, of one agent called device agent (split in CO and OP semiagents) for each device present in the environment, and of the directory facilitator (DF) of LEAP wherein the agents register the services they provide. The majordomo provides the services for planning and executing plans, while the device agents provide the services concerning the capabilities of the corresponding devices (e.g., a cellular phone provides the SMS service). Each device agent takes part to the planning activity providing the decompositions known by the device to the majordomo through the CO semiagent and to the execution activity regulating and monitoring the functioning of the device through the OP semiagent. In addition to the agents previously cited, the majordomo exploits some agents to handle the planning and execution activities. These agents are transient in the platform and are not visible

---

1. Domotic agency is available at HTTP://HOME.DE1.POLMI.IT/NGATTI/DA2008.RAR.
2. LEAP is a light version of JADE [7], i.e. the principal middleware for multiagent systems. LEAP and JADE are developed in JAVA.
to the device agents; we discuss details in Section 3.3. Once plans have been produced, they are stored in a library within the majordomo agents. This allows one to reuse such plans if the devices required to achieve them keep to be present in the environments. On the contrary, in presence of highly dynamic network of devices, planning will be necessary.

The main technological issues to address in the domotic agency architecture concern the design and the interaction of the CO/OP semiagents.

We consider at first the OP semiagents. They cannot be implemented in JAVA for a wide class of devices. This happens for all the embedded systems with low computational and memory capabilities that do not support any JVM, e.g. wireless sensors. Moreover, for a large class of devices supporting JAVA, e.g. palmtops, cellular phones, and smart phones, the use of LEAP for implementing the OP semiagent is not satisfactory due to some technical limitations. The device agents must be able to work also when they are not connected to the agency. To this end, it is necessary that a LEAP platform runs on each single device and that the device agents are launched in a container linked to the platform related to the own device. In this way the whole system, composed of the agency and of the devices, is split over several LEAP platforms and communication between different platforms is required. This form of communication is not currently satisfactory in LEAP and it is supported only for TCP/IP. This prevents the employment in the agency of all the devices whose communication technologies are not based on TCP/IP, e.g. Bluetooth.

We consider now the CO semiagents. With fixed devices, once these have been connected to the wired network, the corresponding CO semiagents can be easily launched in LEAP by creating a remote container linked to the main platform (i.e., the one with the majordomo agent). Instead, with mobile devices, LEAP presents some complications. A CO semiagent must be able to move from the own device to the main platform or must be directly launched on such platform. This is essentially needed to reduce possible delays in the planning activities due to the devices’ low computational and communication capabilities. However, both mobility between different platforms is not supported by LEAP and the employment of LEAP to launch the CO semiagent in the main platform is not feasible in concrete applications. Indeed, in order for an agent to be launched in a remote container, either the classes of the CO semiagent must be already present on the main platform before the device connects to the agency or the classes must move from the device to the main platform. The first situation is not reasonable in real-world applications, the main platform having not the CO semiagent classes of all the possible devices. The second situation is not technically supported. Indeed, classes’ mobility requires JAVA reflection, but several JVMs for mobile devices do not implement it, e.g. JVMs for cellular phones. Moreover, LEAP does not provide any support for the automatic discovery of platforms.

In order to assure an effective functioning of the domotic agency, we need to address the following issues. 1) The mobile devices must be able to dynamically and automatically connect to the domotic agency. 2) The agency must know at each time point what devices are currently connected. 3) The device agents must be able to move into the platforms on which the domotic agency runs.

In our proposal OP semiagents can be implemented by employing different programming languages, whereas every CO semiagent is a LEAP agent. Our solution introduce three components to cope at best with the considered problem. We briefly introduce them and subsequently we discuss their functioning in detail.

- A Bluetooth module called Bluetooth Access Point (BAP) to bridge the agency with the mobile devices (the proposed system supports also different technologies, e.g. WiFi, employing modules defined similarly to BAP).

- An agent called Mobility Server Agent (MSA) running on the main platform, that allows the devices to connect to the agency via BAP and allows the devices to move their CO semiagents into the main platform. MSA is a LEAP agent.

- An application called LifeCycle Manager (LCM) running on each mobile device, that continuously searches for the domotic agency, and that, once it has discovered the agency, connects to it and transmits the classes of the corresponding CO semiagent to the agency; LCM is not a LEAP agent, but it must implement an ad hoc protocol we defined (we omit the protocol, being out of the scope of the paper).

The BAP module publishes the service ‘DOMOTIC AGENCY AP’ by using a Bluetooth interface. The LCM of each mobile device non-connected to an agency is in the discovery phase searching continuously for a BAP. Once the LCM has found a BAP, it accesses to the service ‘DOMOTIC AGENCY AP’ through RFCOMM protocol and subsequently the BAP module enables the communication between the LCM and the MSA. In this situation the LCM transits from the discovery phase to the initialization phase. In this phase the LCM accomplishes two operations: it transmits the classes of the own CO semiagent to the agency through a class loader service provided by MSA and concurrently it launches an agentLet on the device. This is the counterpart of the CO semiagent on the device. The agentLet enables the communication between the CO semiagent and the OP semiagent. If devices have very low computational capabilities, agentLet can be included in the OP semiagent. The present implementation of the LCM is opaque both to the classes of the CO semiagent to transmit to the agency and to the classes of the specific agentLet to launch. Precisely, the agentLet can be implemented by using different programming languages and can be easily set as parameter of the LCM. Once the MSA has received the classes
of a CO semiagent, it launches such an agent. Once both the CO semiagent and the agent are active, the LCM suspends itself until the device is connected to the agency. When the device disconnects the agency, the LCM activates again. The MSA grants a CO semiagent to communicate with the related device. In other words, the MSA works as a bridge between the network on which LEAP platform runs and the mobile devices. Moreover, the MSA continuously makes polling to check the presence of the devices that are currently registered in the agency and, if a device is not present, the MSA kills the corresponding CO semiagent. Fig. 1 depicts the architecture of the domotic agency.

3.2 Ontologies for Domotic Agency

We need to provide an ontology to describe the services the agents can provide and the content of the messages exchanged by them. We use the FIPA [6] ontology to describe the agents’ services. The ontology for the content of the messages instead is an original contribution of us.

The content of the messages exchanged by agents during their activities is composed of tasks and decompositions for D-HTN plans. We codify a D-HTN plan by an ad-hoc ontology based on RDF [9]. We call it COM.DOMOTIC-AGENCY.ONTOLOGY.D-HTN. The base element of the ontology is the task. We recall that a task can be primitive or non-primitive (a primitive task is a method executable by the agent). In order to provide a satisfactory ontology, we distinguish final decompositions from middle decompositions. In a final decomposition \( m = (t, d) \), \( t \) is a primitive task and \( d \) is a method executable by the agent. In a middle decomposition \( m = (t, d) \), \( t \) is a task to decompose and \( d \) is a task network. We define the class MIDDLEDECOMPOSITION for middle decompositions and the class FINALDECOMPOSITION for final decompositions. A task can be of two possible classes: COMPOUNDTASK if it is the task \( t \) to decompose within a middle decomposition, and TASK if it is one of the tasks that compose a task network \( d \) within a middle decomposition or it is the task \( t \) of a final decomposition. Given a middle decomposition \( m = (t, d) \), \( t \) is associated to \( m \) through the property HAS-COMPOUNDTASK, whereas the tasks composing the task network \( d \) are associated to \( m \) through the property HAS-TASK. Given a final decomposition \( m = (t, d) \), \( t \) is associated to \( m \) through the property D-HTNVOC:HAS-TASK, whereas the method is identified by using a string and it associated to \( m \) through the property ASSOCIATED-METHOD.

Both TASK and COMPOUNDTASK have two properties. The first one is DHTN-VOC:HAS-INPUT whose argument, of class rdfs:SEQ, is a container whose arguments must be of class INPUT. The second one is HAS-OUTPUT whose argument, of class rdfs:SEQ, is a container whose arguments must be of class OUTPUT. Both INPUT and OUTPUT are defined, through the property rdfs:SUBCLASSOF, as subclasses of ARGUMENT. The inputs and outputs are typed through the property ARGUMENTTYPE whose argument is a rdfs:DATATYPE property.

We need now to define how the inputs of the task \( t \) to decompose are employed by the task network \( d \) and, vice versa, how the outputs of the task network \( d \) are employed by the task to decompose \( t \). We define the two following properties: we use INPUT-MAPPED-TO to map an input of \( t \) to a specific input of a specific task in \( d \), and, analogously, we use OUTPUT-MAPPED-FROM to map a specific output of a specific task in \( d \) to an output of \( t \). For instance, consider a task FILE:WORD:COUNT that receives a file as input and

![Figure 1. Domotic agency’s architecture with three mobile devices and one fixed device.](image-url)
produces an integer as output. Suppose that such a task can be decomposed in a task network composed of a couple of primitive tasks: FILE_READ that receives a file as input and produces a text as output, and WORD_COUNT that receives a text as input and produces an integer as output. We must map the input of the non-primitive task FILE_WORD_COUNT to the input of the primitive task FILE_READ and we must map the output of the primitive task WORD_COUNT to the output of the non-primitive task FILE_WORD_COUNT.

Moreover, we need to specify how the outputs produced by the tasks belonging to a task network \( d \) can be used as inputs by other tasks belonging to the same task network. We use the property WAIT-FOR-OUTPUT to maps inputs of tasks belonging to \( d \) to outputs of tasks belonging the same task network. Notice that this property implicitly induces an ordering over the execution of the tasks. Consider the example we have reported above: a task FILE_WORD_COUNT to decompose in a task network composed of task FILE_READ and task WORD_COUNT. To assure that the task WORD_COUNT is executed after the task FILE_READ and the output produced by FILE_READ is passed to the task WORD_COUNT, we employ the property WAIT-FOR-OUTPUT between the output of FILE_READ and the input of WORD_COUNT. By introducing fictitious inputs and outputs, we can assure a task \( t_2 \) to be executed after a task \( t_2 \) also when \( t_2 \) does not need any input from \( t_1 \).

Each decomposition, both MIDDLE-DECOMPOSITION and FINAL-DECOMPOSITION, is characterized by three performance indexes through the following properties: PERFORMANCE-INDEX, COST-INDEX, and SUCC-INDEX.

We now consider conditional operators if-then and while. We recall that these operators can be applied only to the components of a task network \( d \) and therefore, in our ontology, they are defined only in middle decompositions. We initially consider the operator if-then. We describe it by using an additional task \( t_c \) (of class TASK) whose output (of class CONDITIONAL-TYPE) drives the conditional execution and by two additional task networks \( d_l \) and \( d_f \) to activate when the condition is true and false, respectively. The two additional tasks networks \( d_l \) and \( d_f \) are bound to \( t_c \) through two properties: IF-CONDITIONAL-IS-TRUE maps the output of \( t_c \) to all the tasks composing \( d_l \) and, analogously, IF-CONDITIONAL-IS-FALSE maps the output of \( t_c \) to all the tasks composing \( d_f \). The definition of the operator while is similar to one of the operator if-then: a task \( t_c \) and a task network \( d_w \) are introduced and the output of \( t_c \) drives the execution of the task composing \( d_w \).

Finally, we characterize each decomposition with its application domain (e.g., “communications”, “health-care”) through the property HAS-DOMAIN. We report in Figure 2 an example of instance produced according to our ontology: the decomposition of the task FILE_WORD_COUNT in FILE_READ and WORD_COUNT.

![Figure 2. A decomposition in COM.DOMOTIC-AGENCY.ONTOLOGY.D-HTN.](image-url)

### 3.3 Planning and Execution by Agents

In domotic agency the planning and execution activities are addressed by employing three services: TASK_RESOLUTION, TASK_DECOMPOSITION, and TASK_EXECUTION. We discuss these services in detail. When CO semiagents are launched, they connect JADE DF to be registered. In the registration they specify the application domain related to the own device.

The service TASK_RESOLUTION is provided by the majordomo of the agency and its aim is to build an executable plan, to execute it, and finally to inform the requester of the service about the results of the planning and execution activities. Obviously, the initial task of the plan is specified by the agent that has requested the service. This service is implemented by employing FIPA-REQUEST protocol [6].

In order to assure an efficient planning activity, once the majordomo has received a request for TASK_RESOLUTION service, it assigns the request an identifier \( i \) and launches an agent named goal agent to manage the request. In this way, the majordomo can wait for new requests, while the goal agents solve the received requests. The planning algorithm executed by goal agents is based on breadth-first search. Breadth-first search can be easily implemented by using iterative algorithms. However, in presence of behav-
ior based agents its implementation is involved. To this end, we developed a finite-state machine behavior (see Fig. 3).

![Finite-state machine behavior of a goal agent.](image)

The behavior starts from *level creation* state where a level is added to the plan. The first time the behavior is in this state the zero level composed of the initial task is added to the plan. If all the tasks composing the last level of the plan are primitive, then the behavior terminates. Otherwise, the behavior transits to the *task expansion* state where each task composing the last level of the plan is singularly analyzed. For each task to analyze the behavior transits to the *decomposition request* state where the goal agent asks for decompositions to the CO semiagents present in the agency with the same application domain of the task to decompose through the service TASK\_DECOMPOSITION and subsequently waits for such decompositions.

The service TASK\_DECOMPOSITION is provided by the CO semiagents and its aim is to provide decompositions for a given non-primitive task. This service is employed by the majordomo during the planning activity and is implemented by employing FIPA-QUERY protocol [6].

Once the goal agent has received all the requested decompositions or a temporal deadline has been expired, the behavior transits to the *decomposition processing* state. In this state, if the goal agent has not received any decomposition, then the behavior transits to the *backtracking* state in which the task currently analyzed is marked as ‘unsolvable’. Moreover, in the *backtracking* state the behavior removes all the tasks that compose the decomposition wherein the task currently analyzed is, and it sets the next level to analyze equal to the previous level. In this way, alternative plans can be found. If instead the goal agent has received decompositions, the behavior of the agents accomplishes the following steps: 1) it checks the effectiveness of the decomposition, verifying the presence of cycle among the tasks composing the task network and the task to decompose, 2) it removes duplicates of the same decomposition, 3) it removes all the decompositions wherein at least a task is marked as unsolvable, 4) it orders the decompositions according to the performance policies. Once, all these steps have been accomplished the behavior transits to the *task expansion* state.

The service TASK\_EXECUTION is provided by the CO semiagent and its aim is to execute a given primitive task. This service is employed by the goal agents during the execution activity and is based on FIPA-REQUEST protocol [6]. The execution activity is based on a finite-state machine behavior in order to effectively address the situations where the execution of a task fails and recovery planning is needed. For the sake of brevity, we omit this description.

4 Experimental Evaluation

4.1 Case Study and Experimental Setting

In order to evaluate the feasibility of multiagent technologies in AmI applications, we focus on the response time. We analyze the time spent by the agency to produce plans in presence of different configurations of the devices’ network. Properties such as agency reconfigurability and interoperability between heterogeneous systems are granted by the multiagent technologies we have employed; the soundness of the planner has been already proved.

For reasons of space, we limit our analysis to a simple case study where there are \( m \) device agents which provide the same following decompositions (results with more complex settings are similar):

\[
\begin{align*}
(T, ([n_1 : A(a))], [n_2 : B(b)]), (n_3 : C(a, b, b)), (n_4 : D(c)),
(n_1 \succ n_3) \wedge (n_2 \succ n_3) \wedge (n_3 \succ n_4)),
(A(o), ([n_1 : E(o))], (n_2 : F(a, o)), (n_3 \succ n_2)),
(B(o), ([n_1 : G(a))], (n_2 : H(a, o)), (n_1 \succ n_2)),
(D(i), ([n_1 : I(i, a))], (n_2 : J(a)), (n_1 \succ n_2)).
\end{align*}
\]

In words, the task \( T \) can be decomposed in four tasks \( A, B, C, D \) such that \( A \) and \( B \) must be executed before \( C \) and \( D \) must be executed before \( E \). The tasks \( A, B, C, D \) can be decomposed in a pair of tasks to execute sequentially, e.g. \( A \) is decomposed in \( E \) and \( F \) and \( E \) must executed before \( F \). The tasks \( C, D, E, F, G, H, I, J \) are primitive. The situation where all the devices provide the same decompositions represents the worst case for D-HTN planner, since all the devices are queried by the majordomo for each decomposition request and the number of decompositions to the task \( T \) is \( m^k \), where \( k \) is the number of primitive task (in the considered case study \( k = 7 \)).

We consider different experimental settings in each one we consider situations characterized by different values of \( m \). **S1:** all the \( m \) CO semiagents and the majordomo run on the same PC; the CO semiagents are not connected to the corresponding OP semiagents. **S2:** all the \( m \) CO semiagents run on the same PC, whereas the majordomo run on a different PC; the CO semiagents are not connected to the corresponding OP semiagents. **S3:** all the \( m \) CO semiagents and the majordomo run on a single PC; the CO semiagents
are connected to the corresponding OP semiagents that run on a single palmtop. S4: all the m CO semiagents and corresponding OP semiagents run on a palmtop, whereas the majordomo run on a PC.3 We implement domotic agency by employing JRE 1.5.0, JADE 3.3, and JENA 2.1 [1].

4.2 Experimental Results

For each setting S1-S4 we have evaluated the time spent for solving a single task T with different values of m; we denote the average response time by \( \overline{R} \). We report these values in Table 1. In S3 and S4 the notation \( n_1/n_2 \) denotes that the agency is populated by \( n_2 \) device agents and, among these, \( n_1 \) run on the palmtop.

We make some observations. Consider S1. Although the task T is simple to solve, \( \overline{R} \) is not negligible also in presence of \( m = 1 \) (i.e., \( \overline{R} = 0.83s \)). This is mainly due to overheads in the planning and execution activities. The value of the overheads can be easily derived by considering \( m > 1 \). Precisely, the overhead time is about 0.71s per request. Although the number of produced plans increases as \( O(m^2) \), the increasing of \( \overline{R} \) is approximately linear in \( m \). This result shows that the proposed solution is not feasible when the user’s service requires very low response time, whereas it can be employed when the service requires a response time of the order of some seconds also in presence of a large amount of redundant devices. Typically, the services provided in AmI applications requires response time of the order of some seconds. Consider S1 and S2. The values of \( \overline{R} \) are similar and therefore the planning activity is not significantly delayed when CO agents runs on different servers linked through high speed connections. Consider S1 and S3. Also in this comparison the values of \( \overline{R} \) are similar. The short delay present in S3 is due to the communication between CO semiagents to the OP semiagents. Consider S1 and S4. The values of \( \overline{R} \) significantly increase because of intensive communications through Bluetooth. This result underlines the effectiveness of the proposed solution also when the device’s computational capabilities are not low (i.e., the device is palmtop).

5 Conclusions and Future Works

In this paper we have considered the problem of developing a multiagent system based on the state-of-the-art technologies to support planning in AmI applications. Planning is of prominence importance in AmI, since it allows the dynamical decomposition of users’ requests in a course of actions executable by devices that populate the AmI system. Although literature discuss what is the best planner for AmI applications, it leaves open technological issues. In particular, it does not analyze the feasibility of the technologies currently available in the state of the art in developing planning systems for AmI applications.

In this paper we present a multiagent system to support planning based on standard technologies, where it is possible, and developing ad hoc solutions otherwise. We have employed LEAP, RDF ontologies, and FIPA protocols. Moreover, we show that the management of mobile devices into the AmI system requires ad hoc solutions for the remote launching of the device agents. Finally, we preliminarily evaluate the time spent by our system to solve a goal risen by an user.

In future, our intention is to improve the performance of the planner, to include scheduling algorithms to manage situations where there are concurrent requests, and to provide experimental evaluations in concrete case studies.

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>R</td>
<td>m</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>0.83s</td>
<td>1</td>
<td>0.92s</td>
</tr>
<tr>
<td>10</td>
<td>2.35s</td>
<td>10</td>
<td>2.39s</td>
</tr>
<tr>
<td>20</td>
<td>3.12s</td>
<td>20</td>
<td>3.12s</td>
</tr>
<tr>
<td>30</td>
<td>4.21s</td>
<td>30</td>
<td>4.43s</td>
</tr>
<tr>
<td>50</td>
<td>7.39s</td>
<td>50</td>
<td>7.62s</td>
</tr>
</tbody>
</table>

Table 1. Average response time \( \overline{R} \) in different experimental settings in function of the number \( m \) of device agents.

References


---

3We used two PCs with 1.4GHz CPU and 1GB RAM connected through Ethernet and a palmtop with 624MHz CPU and 64MB RAM connected to the PCs through Bluetooth v.2.