Rainbow: an Intelligent Platform for Large-Scale Networked Cyber-Physical Systems

Andrea Giordano, Giandomenico Spezzano and Andrea Vinci

CNR – National Research Council of Italy
Istitute for High Performance Computing and Networking (ICAR)
Via P. Bucci 41C - 87036 Rende(CS), Italy
`{giordano,spezzano,vinci}@icar.cnr.it`

Abstract. Recent advancements in the fields of embedded systems, communication technologies and computer science, have laid the foundations for new kinds of applications in which a plethora of physical devices are interconnected and immersed in an environment together with human beings. These so-called Cyber-Physical Systems (CPS) issue a design challenge for new architecture in order to cope with problems such as the heterogeneity of devices, the intrinsically distributed nature of these systems, the lack of reliability in communications, etc. In this paper we introduce Rainbow, an architecture designed to address CPS issues. Rainbow hides heterogeneity by providing a Virtual Object (VO) concept, and addresses the distributed nature of CPS introducing a distributed multi-agent system on top of the physical part. Rainbow aims to get the computation close to the sources of information (i.e., the physical devices) and addresses the dynamic adaptivity requirements of CPS by using Swarm Intelligence algorithms.

1 Introduction

The increasing use of smart devices and appliances opens up new ways to build applications that integrate the physical and virtual world into consumer-oriented context-sensitive Cyber-Physical Systems (CPS) [9, 12, 8] enabling novel forms of interaction between people and computers. CPS are combinations of physical entities controlled by software systems to accomplish specified tasks under stringent real-time and physical constraints.

The emerging cyber-physical world interconnects a vast variety of static and mobile resources, including computing/medical/engineering devices, sensor/actuator networks, swarm of robots etcetera. Examples of CPS applications include [14] traffic control, power grid, smart structures, environmental control, critical infrastructure control, water resources and so on. These systems could be pervasively instrumented with sensors, actuators and computational elements to monitor and control the whole system. Furthermore, these devices should be interconnected so as to communicate and interact with each others and with people.

This scenario is supported by recent technology advancement in the fields of communication, embedded systems and computer science. On the communication side, new protocols like EPC TDS and IPv6 ensure unique addressability for all the elements involved in a CPS, while connectivity technologies like IEEE 802.11, ZigBee, Umts and ZTE, could ensure light and fast connection between the devices involved in the system and between the devices and the Internet. On the embedded systems side, the miniaturization and the constant improvement of energy efficiency of electronic components enables the environment to be easily instrumented with sensors, actuators and computing devices, while the presence on the market of cheap and general purpose single-board computers, like Raspberry PI [15] and BeagleBoard enables new approaches different from currently adopted ones. On the computer science side, the development of new techniques to analyse a massive volume of data, together with the advances in the fields of artificial and swarm intelligence, opens up new ways to exploit the data in order to coordinate the operations of the large number of devices involved. The networked cyber-physical world has a great potential for achieving tasks that are far beyond the capabilities of existing systems. However, the problem of effectively composing the services provided by cyber and physical entities to achieve specific goals remains a challenge [9, 12, 1]. Advanced models and architectures, autonomous resource management mechanisms, and intelligent techniques are needed for just-in-time assembly of resources into desired capabilities.

The complexity of a CPS, and the large number of elements involved, makes data analysis and operation planning a very difficult task. A currently used approach involves two layers: a *physical* layer and a *remote* (cloud) cyber layer. The physical layer sends sensed data to a remote server, which processes them and computes a suitable operation plan. Afterwards, the remote server sends the sequence of operations it must execute to each device on the physical layer. The reasoning is performed in the remote layer. This solution cannot be applied when there are constraints on *responsivity* time, that is, when a system needs to react fast to critical events that may overwhelm its integrity and functionality. Communication lag and remote processing can cause delays that a system simply cannot bear. A wide variety of applications means a wide variety of devices. Currently, there is a plethora of different devices, each with its own particular functionalities and capabilities. There are simple devices without any computational unit as well as "smarter" devices with high computation power inside. There are devices with no operating systems and devices with simple or complex operating systems, such as tinyOS or Android. Our framework is designed to cope with this inherent heterogeneity. To addressing the issues described above, our proposal moves on these main lines:

– Hiding the heterogeneity of CPS by introducing a *virtual object* layer.
– Moving the computation as close as possible to the physical resources in order to foster good performance and scalability.
– Introducing a distributed intelligence layer between the physical world and remote servers (cloud), which can execute complex tasks and horizontally/vertically coordinate the devices.

– Switching from a cloud-based model to a cloud-assisted one, where the intelligent intermediate level carries out almost all the real-time control tasks, whereas the remote cloud level remains in charge of non-real-time tasks such as offline data analysis or presentation. The information provided by the data analysis executed by the remote server are used by the intermediate level to optimize its operations and behaviour.

In this paper we propose a three-tier architecture (Rainbow) that uses single-board computers such as the Raspberry PI to connect massive-scale networks of sensors to the Cloud. This architecture is composed by the *Cloud* layer, the *Intermediate* layer and the *Physical* layer. Sensors are partitioned into groups, each of which is managed by a single computing node. These computing nodes host multi-agent applications designed to monitor multiple conditions or activities within a specific environment. Furthermore, agents can be intelligently assisted by cloud services, that support complex analytics, modeling, optimization and visualization tools, to make better operation decisions.

We present a new integrated vision that allows the designing of a large-scale networked CPS based on the decentralization of control functions and the assistance of cloud services to optimize their behaviour. Decentralization will be obtained using a distributed multi-agent system in which the execution of a CPS application is carried out through agents' cooperation [5, 10, 11, 2]. The distributed multi-agent system lays the foundations for properly exploiting swarm intelligence concepts. Swarm intelligence [3, 7] systems are typically multi-agent systems made up of a population of simple agents interacting locally with one another and with their environment. The agents follow very simple rules, and although there is no central control structure dictating how individual agents should behave, local and to a certain degree random, interactions among such agents lead to the emergence of "intelligent" global behaviour, unknown to the individual agents. Natural examples of swarm intelligence include ant colonies, bird flocking, animal herding, bacterial growth, and digital infochemicals. Agents interacting with cloud services can exploit the analysis, predicting, optimization and mining scalable capabilities on historical data allowing applications to adjust their behaviour to best optimize their performance.

The remainder of this paper is structured as follows: Section 2 is devoted to a description of the proposed Rainbow Architecture; Section 3 describes two example of use; finally, we draw conclusions and the future works.

2 Rainbow architecture

Rainbow is a three-layer architecture designed in order to bring the computation (i.e the controlling part) as close as possible to the physical part. Since CPS foresees that physical entities are spread across a large (even geographic) area, the previous assumption implies the controlling part to be intrinsically distributed.

Our proposal foresees the use of a *distributed agent-based* layer in order to address the aforementioned issues. The agent paradigm has several important characteristics:

Autonomy. Each agent is self-aware and has a self-behaviour. It perceives the environment, interacts with others and plans its execution autonomously.

Local views. No agent has a full global view of the whole environment but it behaves solely on the basis of local information.

Decentralization. There is no “master” agent controlling the others, but the system is made up of interacting “peer” agents.

Through these basic features, multi-agent systems make it possible to obtain complex *emergent* behaviours based on the interactions among agents that have a simple behaviour. Examples of emergent behaviour could refer to the properties of adaptivity, fault tolerance, self-reconfiguration, etcetera. In general, we could talk about *swarm-intelligence* when an “intelligent” behaviour emerges from interactions among simple entities. There are many swarm intelligence algorithms in the literature that could be properly adopted in the context of CPS. In 3.2 we show an example where Swarm Intelligence is used to map noise pollution inside a city area.

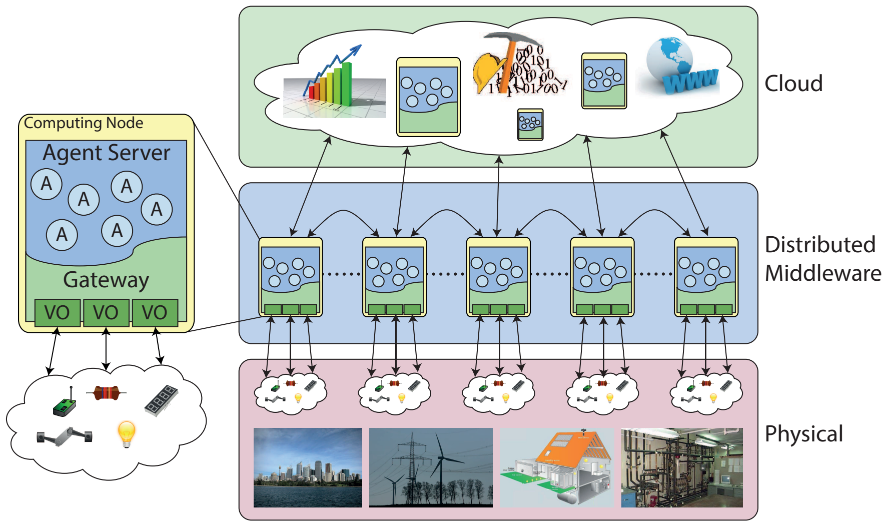


Fig. 1. Rainbow architecture.

Rainbow architecture is shown in Figure 1. As it can be seen, the architecture is structured into three layers. The bottom layer is the one that is devoted to the physical part. It encloses sensors and actuators, together with their relative computational capabilities, which are directly immersed in the physical environment.

In the Intermediate layer, sensors and actuators of the physical layer are represented as virtual objects (VOs). VOs offer to the agents a transparent and ubiquitous access to the physical part due to a well-established interface exposed as API. VO allows agents to connect directly to devices without care about proprietary drivers or addressing some kind of fine-grained technological issues. Each VO comprises "functionalities" directly provided by the physical part. Essentially, a VO exposes an abstract representation (i.e. *machine-readable description*) of the features and capabilities of physical objects spread in the environment. Functionalities exposed by different types of VOs can be combined in a more sophisticated way on the basis of event-driven rules which affect high-level applications and end-users.

In summary, as detailed in section 2.1, all the devices are properly wrapped in VOs which, in turn, are enclosed in distributed *gateway* containers. The computational nodes that host the gateways represent the middle layer of the Rainbow architecture. Each node also contains an agent server that permits agents to be executed properly. Gateways and agent servers are co-located in the same computing nodes in order to guarantee that agents exploit directly the physical part through VO abstraction. Instead of transferring data to a central processing unit, we actually transfer the process (i.e. fine-grain agent's execution) toward the data sources. As a consequence, less data needs to be transferred over a long distance (i.e. toward remote hosts) and local access and computation will be fostered in order to achieve good performance and scalability .

The upper layer of Rainbow architecture concerns the cloud part. This layer addresses all the activities that cannot be properly executed in the middle layer like, for instance, algorithms needing complete knowledge, tasks that require high computational resources or when a historical data storage is mandatory. On the contrary, all tasks where real time access to the physical part is required could be suitably executed in the middle layer.

2.1 Virtual Objects

We address issues about heterogeneity in CPS by introducing the Virtual Object (VO) concept. VO aims to hide heterogeneity by supplying a well-established interface permitting the physical parts to be suitably integrated with the rest of the system.

VO could be defined as a collection of physical entities like sensors and actuators, together with their computational abilities.

It can be composed of just a simple sensor or it can be a more complex object that includes many sensors, many actuators, computational resources like CPU or memory and so on.

In general, VO outputs can be represented by *punctual values* (e.g. the temperature at a given point of a room) or *aggregate values* (e.g. the average of moisture during the last 8 hours). Also, the values returned by VOs could be just the measurement of sensors or could be the result of complex computations (e.g. the temperature of a given point of space computed by means of interpolation of the values given by sensors spread across the environment).

Furthermore, a VO could supply actuation functionality by changing the environment on the basis of external triggers or internal calculus.

These different kinds of behaviour that VO can expose must be taken into account. VO is therefore conceived as a complex object that can read and write upon many simple physical resources. More in detail, we consider that each VO exposes different *functionalities*. Each functionality can be either sensing or actuating and can be refined by further parameters that dynamically configure it.

The previous assumption leads to the definition of *resource* as the following *triplet*:

$$[VOId, VOFunctionId, Params]$$

Where `VOId` uniquely identifies the VO, `VOFunctionId` identifies the specific functionality and `Params` is an ordered set of parameter values that configure the functionality.

For example let's consider a *Virtual Room* made of sensors for measuring different physical quantities inside a room such as *temperature*, *moisture*, *brightness* and so on. Suppose now you want to read from Smart Room the temperature in a given spatial point of the room. In that case the triplet could be:

$$[VirtualRoom, temperature, [x, y, z]]$$

Where `x`, `y` and `z` are the cartesian coordinate of the point of interest.

Using object oriented terminology, a Resource could be seen as a particular "instance" of a functionality of a given VO.

Besides read and write operations (i.e. sensing and actuation), it is provided for VOs to be able to manage events that occur in the physical part. To that scope, our proposed middleware includes a *publish/subscribe* component for managing events in each computing node. Each event is defined by a *logical rule* where one or more VOs could be involved.

Each rule is a *logical proposition* in which the *atomic predicates* can be of the following kinds:

- *resource < threshold (e.g. temperature < 300)*
- *resource > threshold*
- *boolean_resource (e.g. the_door_was_unlocked)*

Just an example of rule:

(temperature < 100 and brightness >500) or people > 3 or door_unlocked

All the physical things linked to a computing node together with relative VOs is enclosed in the *Gateway* container. The Gateway exposes an interface to interact directly with the VOs.

Each gateway represents the "entry point" that agents can use to exploit VOs of the relative computing node.

In the following is described the interface of Gateway that will be used by the overlying layer:

```
interface GatewayInterface {
    void resourceNaming(String name, VOId voId, VOFunctionId functionId,
        VOFunctionParams params);
    VOResult check(String name);
    VOResult check(String name, VOFunctionParams params);
    VOResult acting(String name);
    VOResult acting(String name, VOFunctionParams params);
    void setRule(Rule rule, String idRule);
    void subscribe(String idRule, EventHandler handler);
}
```

The method `resourceNaming` assigns an identification `name` to a given resource supplied by a given VO. A resource is a specific instance of a *functionality* of a VO refined by some *parameters*. In other word, a resource is the above-mentioned triplet: $[VOId, VOFunctionId, Params]$. The `name` assigned to a resource via `resourceNaming` can be used in the other methods in order to simply identify the resource. Furthermore, the identification `name` of a resource is useful to compose the rules in a more human-readable fashion.

The method `check` reads the current value of the resource identified by `name` whereas `acting` triggers the actuation operation upon the resource identified by `name`. Both `check` and `acting` methods are of two kinds: the first take only `name` as parameter and refers to the resource as it is previously defined in `resourceNaming`; the second kind, instead, permits the parameters of the referred resource to be refined dynamically.

The method `setRule` permits a complex rule to be published (e.g. *(temperature < 100 and brightness >500) or number_of_ person > 3 or door_unlocked*) and to assigns an id (i.e. `idRule`) useful for subscribing the rule afterwards.

The method `subscribe` permits a previously published rule (identified by `idRule`) to be subscribed. The occurrence of the event identified by `idRule` will be notified to the `handler` passed as a parameter to the method.

2.2 Rainbow Multi Agent system

The Multi Agent component of the Rainbow architecture is made up of the following entities: *Agents*, *Messages*, the *Agent Server* and the *Deployer*. Figure 2 shows these entities and how they interact among themselves and with the *Gateway*.

The *Agent Server* is the container for the execution of agents. It offers functionalities concerning the life cycle of the agents as well as functionalities for agents' communication. Agent servers are arranged in a peer-to-peer fashion where each agent server hosts a certain number of agents and permits them to execute and interact among themselves in a transparent way. In other words, when an agent requests the execution of a functionality, its host agent server is in charge of redirecting transparently the request to the suitable agent server. In the following are listed the main functionalities each agent server exposes:

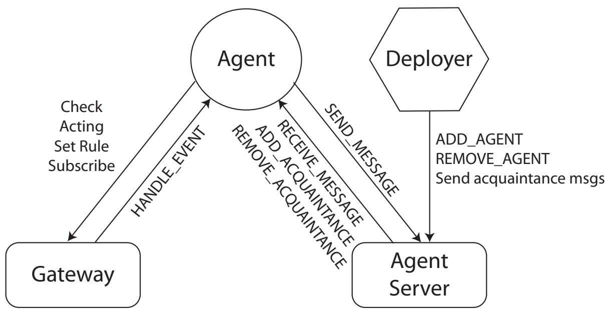
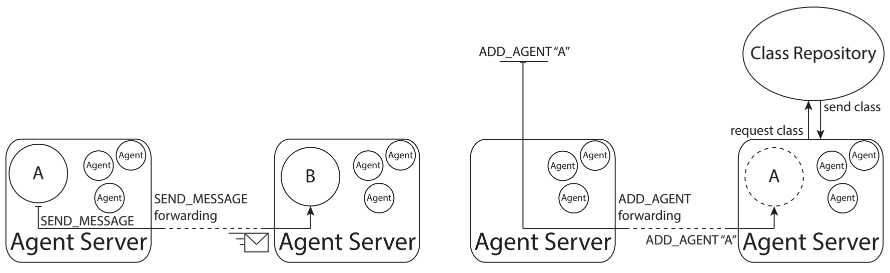


Fig. 2. Rainbow multi-agent entities.

`SEND_MSG`**.** Through this functionality, the communication between agents is performed. The Agent Server is responsible for correctly delivering messages from the sender agent to the receiver one. If the sender and the receiver do not belong to the same agent server, the message is forwarded to the suitable "peer" agent server which is, in turn, engaged finally to deliver the message. The latter mechanism is showed in figure 3(a).

`ADD_AGENT`**.** It instances an agent to an agent server. Rainbow Multi Agent system is designed to permit agents to be dynamically loaded to the agent server they have to belong to. As in SEND_MSG operation, agent servers are in charge for exchanging information among themselves in order to guarantee the ADD_AGENT request to be delivered to the correct agent server. This mechanism is shown in figure 3(b). The latter figure also shows how the code is dynamically loaded exploiting *class repository server*. More in detail, when an ADD_AGENT request reaches the suitable agent server, if the agent code is not already available, the agent server automatically downloads it from a class repository.

`REMOVE_AGENT`**.** It removes an instance of an agent hosted by an agent server. This operation also exploits the "forwarding" mechanism described above.



(a) SEND_MSG (b) ADD_AGENT

Fig. 3. Forwarding mechanism

A *Message* is the atomic element of communication between agents. It carries an application specific content together with informations about the sender agent and the receiver one.

Our architecture provides for specific kinds of message, that are the *acquaintance message*s. Those messages are used for establishing an acquaintance relationship among agents. The acquaintance message carries information about the location of a given agent (i.e. location of hosting agent server). The agent who receives the acquaintance message will use this information when it needs to send messages toward that destination. This kind of mechanism ensures agent behaviour to be completely independent w.r.t. the locations of agents it has to collaborate with.

For instance, let's consider that an agent is a computing node interconnected with others by means of a ring network. Each agent, therefore, can only interact with its previous agent nodes and its next one. Whenever further nodes must be connected to the ring network, only the acquaintance relationships have to be updated. In other words, a third entity can establish dynamically those acquaintance relationships without resorting to modifying, re-building or restarting any agent.

In Rainbow architecture the entity which is in charge of sending acquaintance messages in order to establish the acquaintance network is called *Deployer*. Deployer could be an external process as well as an agent, it can run during the configuration phase as well as during application execution. The Deployer concept will be described in details in section 2.2.

An *Agent* is an autonomous entity which executes its own behaviour interacting with other agents via Agent Server. In addition, each agent can interact with the physical part exploiting functionalities exposed by the Gateway (i.e. using the Virtual Object abstraction).

The functionalities of an agent are exposed to its own Agent Server and Gateway. As said before, Agent Servers are in charge of the "forwarding" mechanism that eventually ends with the calling of these functionalities, while the Gateway is in charge of notifying the events that occur in the physical part. In the following are listed the main functionalities of an agent:

`RECEIVE_MESSAGE`. It is called when there is a Message to be delivered for the agent.

`HANDLE_EVENT`. It is called by the Gateway to notify that an event is occurred in the physical part.

`ADD_ACQUAINTANCE`. It is called when there is an acquaintance message to be delivered to the agent. The implementation of this functionality concerns the store of the acquaintance relationship between the agent itself and the agent identified inside the message.

`REMOVE_ACQUAINTANCE`. It is called for removing a previously stored acquaintance relationship.

The specific behaviour of an Agent is realized through the implementation of `RECEIVE_MESSAGE` and `HANDLE_EVENT` functionalities.

Dynamic Deployment and Roles The deployment of the agents as well as the configuration of the acquaintance relationships and the start-up of the application are all actions performed by the so-called *Deployer*. An external process or even an agent can act as a Deployer. The deployment phase is typically executed just before the application can start properly; however, it is possible to act as Deployer even during application execution in order to update the configuration dynamically for hosting new features or adapting to foreseen and unforeseen changes in the environment. Deployer can be implemented centrally or in a distributed way. Basically, who acts as a Deployer operates using the `ADD_AGENT` functionality for deploying a new instance of an agent into an agent server, `REMOVE_AGENT` for removing a running agent from an agent server. Furthermore, Deployer is responsible for sending acquaintance messages that eventually end with calls to `ADD_ACQUAINTANCE` or `REMOVE_ACQUAINTANCE` on the specific agents. Finally, Deployer is also in charge of sending suitable "start" messages using `SEND_MSG` in order to start the application properly.

The acquaintance relationship is formally defined by a triplet: [A, B, R] where A and B are the agents involved in the relationship and R is a *Role* label. The triplet above means that agent A knows agent B and that B has the role R as acquaintance of A. During the execution, an agent exploits the Roles of its acquaintances to discriminate about how to interact with them.

As an instance, let's consider that each agent represents a physical person in a town. The relationship between two agents could have roles of neighbourhood and/or friendship. A deployer is in charge of configuring those relationships during the initial phase. In addition, as soon as a person changes home or starts a new friendship, the deployer has to re-arrange relationships dynamically among agents through sending acquaintance messages. During the execution of that system, each agent will use roles of neighbourhood and friendship to discriminate how to interact with other agents. For instance he/it can exchange information about its district with its neighbours while it invites its friends to a party.

3 Application Examples

In this section we introduced two examples of using the Rainbow architecture. The first one aims to show our architecture from a practical perspective in order to understand and better figure out all the system details. The second example is useful to understand how Rainbow can host suitable swarm intelligence strategies in order to realize CPS applications owning properties such as adaptivity, fault tolerance, self-reconfiguration, etcetera.

3.1 Floor control example

In this example we show an application for monitoring and controlling a floor of a building hosting offices. Each floor contains a certain number of room.

Figure 4(a) shows how a generic floor could be. In general, each room contains: doors, desks, chairs and adjustable brightness lights.

Each room is instrumented by some sensors and actuators listed below.

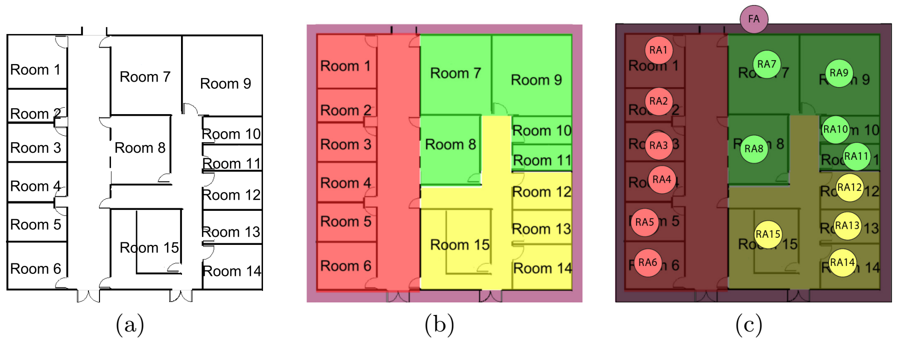


Fig. 4. (a) Floor topology (b) Rooms assignment to computational node. Each different color identify a different node. (c) Logical distribution of agents in the floor.

Sensors:

– sensors that detect the opening and closing of doors;
– sensors that detect when a person enters or leaves a room;
– proximity sensors detecting presence of the people in each zone of a room;
– a weight sensor for each chair in order to detect if the chair is currently used.

Actuators:

– adjustable brightness lights for all zones of a room;
– a display on each desk.

The use of the above described devices, for example, permits adjusting lights on the basis of people movements, writing informational messages on displays and so on.

Integration in Rainbow using Virtual Objects In order to develop the controlling part in a object-oriented fashion, it is required to integrate the above described physical things with Rainbow middleware defining the suitable Virtual Object(VO). Each VO abstracts and wraps a certain number of sensors as well as actuators. For the sake of simplicity, in this example we chose to design VOs in a human-readable fashion: *virtual desk*, *virtual chair*, *virtual door* and *virtual wall*.

The functionalities exposed by these VOs are listed in table 1, 2, 3, 4. It is worth to note that each functionality of the virtual wall is parametric: the *zone* parameter specifies which area of the room is referred.

Each VO is located on the same computing node where the sensors and actuators that VO encloses are connected to. A computational node can generally host VOs that may refer to more than one room. Assuming than we have only three computational nodes available to monitor and control the whole floor, we can assign rooms to nodes as in figure 4(b).

Functionality	**Type**	**Description**
lock	Sensing	Boolean, true if the door is closed
unlock	Sensing	Boolean, true if the door is open
entry	Sensing	Boolean, true when a person enter the room through the door
exit	Sensing	Boolean, true when a person exit the room through the door

Table 1. Virtual Door.

Functionality	**Type**	**Description**
proximity	Sensing	detects people near the chair
sitting	Sensing	Boolean: true when someone sits on the chair

Table 2. Virtual Chair.

Functionality	**Type**	**Description**
near people	Sensing	number of people in the zone (supplied by parameter)
add light	Acting	increase light brightness in the zone (supplied by parameter)
less_light	Acting	decrease light brightness in the zone (supplied by parameter)
light_off	Acting	set off light in the zone (supplied by parameter)

Table 3. Virtual Wall.

Functionality	**Type**	**Description**
proximity	Sensing	detects people near the desk
display	Acting	show a message supplied by parameter on the display

Table 4. Virtual Desk.

Multi-agent floor application The application is designed for managing the floor and its rooms. For each room a energy-saving light-management is developed which considers people presence for suitably adjusting the brightness of the various zones of a room. This control management will also consider if the chairs are utilized or not in order to better adjust the lights. In addition, it permits a message to be displayed on a certain desk when needed. All those features are implemented in the *RoomAgent*. The code inside the RoomAgent is a typical object-oriented code where VOs are exploited as simple objects. The code is omitted in this paper for sake of brevity.

Besides this room-wise features, the application is also designed for addressing issues concerning the entire floor (i.e. where more than one room is involved). For instance, it could be useful to know how many people are in the floor at a given time in order to properly manage the locking of the main door of the floor as well as to shut down all the lights where the floor is empty. In this example, instead, the knowledge of the number of people is used to notify a person when he is alone in the floor writing a message on the display of his desk.

The *FloorAgent* is designed for addressing the above mentioned issues. Summarizing, there is a *RoomAgent* per room and a unique *FloorAgent* as it is shown in figure 4(c).

Deployment of the application As mentioned before the *Deployer* is in charge to load the agents upon the agent servers, to establish acquaintance relationships among them and to start the application.

In our application, each *RoomAgent* must be located in the computing node where the VOs of the relative room belong to.

Conversely, the *FloorAgent* can be located everywhere in the system (it has not connection with any physical part), even in a remote cloud node. The process made by the *Deployer* is summarized in figure 5.

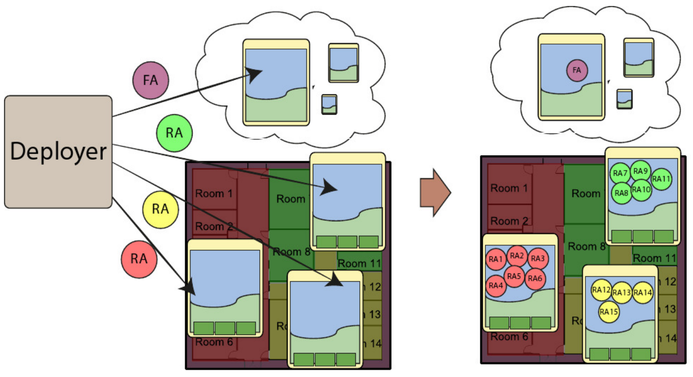


Fig. 5. Deployment of the agents and their physical distribution on the computing nodes.

Agent interaction and acquaintance relationships After loading each agent in the proper location, the *Deployer* sends acquaintance messages to each *RoomAgent* in order to let them know the *FloorAgent*. Afterwards, each agent sends an acquaintance message to the *FloorAgent* in order to be known by it. This is an example of an agent that acts as *Deployer*. Once the deployment phase is completed, the application execution can start. When a person leaves a room, *RoomAgent* will be notified by the gateway and, consequently, will send a message carrying the number of people currently inside the room to the *FloorAgent*. The latter will update its people counter on receiving such a message. When it verifies that there is only one person in the floor, it will send a message to the relative *RoomAgent* that, in turn, will write a message on the desk display.

3.2 Noise pollution mapping

Many environments, such as airports, road works, factories, construction sites, and other environments producing loud noises, require effective noise pollution monitoring systems. Noise pollution is a common environmental problem that affects people's health by increasing the risk of hypertension, ischemic heart disease, hearing loss, and sleep disorders, which also influence human productivity and behavior [13]. For this reason the European Community passed the directive 2002/49/EC [4], which declares noise protection as one necessary objective to achieve a high level of health and environmental conservation. The directive imposes several actions to be made upon member states, including the mapping of noise in larger cities via noise maps. On the basis of these maps, the countries can formulate plans to counter the threat that is noise pollution.

Noise maps are mostly based on numerical calculations that have shown to give good estimates of long term averaged noise levels. However, such maps does not take into account the real-time variation of the noise levels.

Using the Rainbow platform we designed an agent-based, self-organizing system for the real-time construction of noise maps and identification of the sources of noise.

Noise sensors are spread into the environment, linked to the computational nodes, and suitably wrapped inside the VOs. Each agent is directly associated with a VO representing a noise sensor. During the deployment phase, each agent is supplied by the knowledge of its neighbours (i.e. agent associated with a spatially near sensor).

We use a simple self-organizing algorithm, proposed by [16], to let sensor network to self-organize itself in a region partitioning based on similar sensing patterns (*noise levels*). Regions can grow or shrink according to the dynamic variation of noise levels. Organization in regions occurs by creating an overlay network made by agents connected by virtual weighted links. Agents belonging to the same region will have strong links, while agents belonging to different regions will have weak (or null) links.

In the following the details of the algorithm. Let s_i and s_j be two neighbour sensor agents. Let $n(s_i)$ and $n(s_j)$ the values of noise sensed by s_i and s_j, respectively. Let us assume that a distance function D can be defined for couples of v values. Region formation is then based on iteratively computing the value of a logical link $l(s_i, s_j)$ for each and every agent of the system as in following update_link procedure:

```
Update_link:
$if(D(n(s_i), n(s_j)) < T\{$
    $l(s_i, s_j) = min(l(s_i, s_j) + \Delta, 1)$
$\}else\{$
    $l(s_i, s_j) = max(l(s_i, s_j) - \Delta, 0)$
$\}$
```

Where: T is a threshold that determines whether the measured values are close enough for $l(s_i, s_j)$ to be re-enforced or, otherwise, weakened; and Δ is a value affecting the reactivity of the algorithm in updating link. Based on the above algorithm, it is rather clear that if $D(n(s_i), n(s_j))$ is lower than threshold T, $l(s_i, s_j)$ will rapidly converge to 1. Otherwise it will move towards 0. Transitively, two nodes s_h and s_k are defined in the same region if and only if there is a chain of agents such that each pair of neighbours in the chain are in the same region. From the Rainbow perspective, region information is stored adding/removing new acquaintance relationships among agents.

In order to properly map the noise pollution, it is necessary that each and every agent within a region is locally provided with information related to the overall status of the region. To this end, it is possible to integrate forms of diffusive gossip-based aggregation [6] within the described general scheme. The algorithm requires that the agents periodically exchange information with their neighbors about some local value, locally aggregate the value according to some aggregation function (e.g., maximum, minimum, average, etc.), and further exchange in the subsequent step the aggregated value.

4 Conclusions

In this paper we introduced Rainbow, an architecture that permits an easy development of large-scale cyber-physical applications. The novelty of Rainbow is that it relies on the adoption of a distributed multi-agent layer on top of the physical part that is, in turn, wrapped in suitable Virtual Objects. Rainbow aims to hide heterogeneity, cope with complexity and real-time issues. In the future, new intelligent, adaptive and decentralized algorithms will be explored for developing large-scale cyber-physical applications using Rainbow, such as those related to smart cities, power grid, water networks and so on. Furthermore, a well-established interface for the cloud part of the architecture will be defined.

5 Acknowledgments

This work has been partially supported by RES-NOVAE - "Buildings, roads, networks, new virtuous targets for the Environment and Energy" project, funded by the Italian Government (PON 04a2_E)

References

1. Abdelzaher T., Towards an architecture for distributed cyber-physical systems, Proceedings of NSF Workshop on Cyber-Physical Systems, Austin, TX, 2006.
2. Bicocchi N., Mamei M., Zambonelli F., Self-organizing virtual macro sensors, ACM Transactions on Autonomous and Adaptive Systems (TAAS), Volume 7 Issue 1, April 2012.
3. Bonabeau E., Dorigo M., Theraulaz G., Swarm Intelligence: From Natural to Artificial Systems, New York, NY: Oxford University Press, Santa Fe Institute Studies in the Sciences of Complexity, Paper: ISBN 0-19-513159-2,1999.
4. European Directive. The Environmental Noise Directive (2002/49/EG). Official Journal of the European Communities, 2002.
5. Fortino G., Guerrieri A., Lacopo M., Lucia M., Russo W.: An Agent-based Middleware for Cooperating Smart Objects, in Highlights on Practical Applications of Agents and Multi-Agent Systems, Communications in Comp. and Inform. Science (CCIS), Vol. 365, pp. 387-398, Springer, 2013.
6. Jelasity M., Montresor A., Babaoglu O., Gossip-based aggregation in large dynamic networks, ACM Transactions on Computer Systems 23, 3, 219 - 252, 2005.
7. Kennedy J., Eberhart R.C., Swarm Intelligence, Morgan Kaufmann publishers, 2001.
8. Koubaa A., Andersson B., A vision of cyber-physical internet, in proc. Of the Workshop of Real-Time Networks (RTN 2009), Satellite Workshop to (ECRTS 2009).
9. Lee A., Cyber Physical Systems: Design Challenges, Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, IEEE Computer Society Washington, DC, USA, 2008.
10. Leito P., Towards Self-organized Service-Oriented Multi-agent Systems, in Studies in Computational Intelligence Volume 472 2013, Springer.
11. Lin J., Sedigh S., Miller A., Modeling Cyber-Physical Systems with Semantic Agents, in Computer Software and Application Conference Workshops (COMPSACW), IEEE 2010.
12. Sanislav T., Miclea L., Cyber-physical systems - Concept, Challenges and Research Areas, in Control Engineering and Applied Informatics, Vol.14, No.2, pp. 28-33, 2012.
13. Schweizer I., Brtl R. , Schulz A., Probst F., and Mhlhuser M., NoiseMap - Realtime participatory noise maps, in ACM SenSys 2011 Second International Workshop on Sensing Applications on Mobile Phones (Eds.), 2011
14. Shi J., Wan J., Yun H., Suo H., A Survey of Cyber-Physical Systems. In proc. Of the Int. Conf. On Wireless Communications and signal Processing, Nanjing, China, November 9-11, 2011.
15. RaspBerry online, http://www.raspberrypi.org/.
16. Bicocchi N., Mamei M., Zambonelli F., Self-Organizing Virtual Macro Sensors, ACM Transactions on Autonomous and Adaptive Systems, Vol. 7, No. 1, 2012.