# Civitas: The Smart City Middleware, from Sensors to Big Data

Félix J. Villanueva, Maria J. Santofimia, David Villa, Jesús Barba, Juan Carlos López
Department of Information Technologies and Systems
School of Computer Science
Ciudad Real, Spain
{felix.villanueva, mariajose.santofimia, david.villa, jesus.barba, juancarlos.lopez}@uclm.es

***Abstract*—Software development for smart cities bring into light new concerns, such as how to deal with scalability, heterogeneity (sensors, actuators, high performance computing devices, etc.), geolocation information or privacy issues, among some. Traditional approaches to distributed systems fail to address these challenges, because they were mainly devoted to enterprise IT environments. This paper proposes a middleware framework, called Civitas, specially devoted to support the task of service development for the Smart City paradigm. This paper also analyzes the main drawbacks of traditional approaches to middleware service development and how these are overcome in the middleware framework proposed here.**

## I. Introduction

The concept of Smart City has lately been adopted by the research community, companies and public gobernants to summarize a set of advanced services devoted to make current cities more citizien friendly, efficient and sustainable. Traffic management, public transport scheduling, waste management, pollution control, or preventing and avoiding security threats are, among some of the most relevant, examples of problems that the Smart City has to deal with [1].

The Smart City paradigm can therefore be seen as a distributed system in which different sources of information provide data to a set of applications that use these data to elaborate responses at strategic and tactical level. Distributed systems are normally supported in an abstraction layer platform, known as middleware. Ideally, middleware provides a set of tools and software libraries that make transparent to developers the heterogenety in operating systems, programming language, devices, etc., while abstracting them from the different network features (technology, topology, protocol stack, etc.).

The role therefore played by the middleware is essential for the success of the future smart city. Specially relevant is how the smart city service development could be benefit from the abstraction layer provided by a middleware. Productivity in terms of services developed and deployed could be considerably increased. On the contrary, the lack of this comprehensive approach for smart cities would result in the formation of "information island" phenomenon, making synergy of services impossible.

This paper presents the first distributed object-oriented middleware, called Civitas, specifically designed for smart cities. This middleware provides services that range from environmental sensor deployment to the necessary hardware for high performance algorithms devoted to extract information from raw data. Additionally, in order to cope with the *smart* facet of the smart city paradigm, Civitas has also be enhanced with reasoning capabilities. Leveraging reasoning capabilities enables the middleware to have few hard-coded features that are rather deduced from the available data. In this sense, Civitas is able to adapt to the deployed city, without requiring important modifications or adaption works. The main intention of this work is to promote tightly integrated systems and managed smart cities to simplify the IT environment for service developers.

## II. State of Art

So far, most of the literature concerning Smart Cities, as to the solutions for the challenges that arise in this field, is mainly oriented to specific applications for certain city domains. Therefore, it is recurrent to see a *reinventing-the-wheel* approach rather than an holistic one. The later would need of a thorough analysis, isolating the common requirements that would lead to a shared technological platform. This work aims to fill this void detected in the revised related work.

Some of the works focus at the data-acquisition level, describing how the abstraction of the different sources and their integration into the system is performed. In this sense, it is worth mentioning the OpenIoT EU co-funded FP7 project [2]. The OpenIoT architecture enables the development of cloud-based applications using Internet of Things services. The bulk of the OpenIoT framework resides in the virtualization infrastructure used to integrate different sensor networks technologies into the cloud. OpenIoT defines a series of extensions to well known standards such as SSN-XG ontolgy for sensor network definition or RESTful services for sensor access.

The work of Fazio et al. [3] also makes it use of standard specifications and protocols, such as Sensor Web Enablement, to tackle the heterogeneity problems in smart sensor platforms. Three levels of incompatibility are identified in complex sensing infrastructures: device technology, communication and data levels. At the end, sensors are accessed via a REST API which relies on a Sensor Observation Service Agent that abstracts the observations and the sensor system.

However, the use of web (text) based protocols for sensor integration has two major drawbacks: (a) limits the kind and type of devices to use, since text based communication demands of a fair amount of CPU processing to parse the protocol messages; (b) battery life of the sensors shrinks considerably; and (3) is not very suitable for low bit rate communications (e.g. 802.15.4) even in compressed forms (RESTFul, CoAP, etc.).

Whereas some of the works make a strong point at the physical/sensing layer (putting aside how the data is gathered at this level), others tip-toe around this topic and directly start to define the architecture and mechanisms to store, access, process or represent such information. The cloud proposal made by Khan and Kiani [4] emphasizes the contextual-aware data processing, aimed to deliver the right information to applications, citizens or any other agent. First, data is classified into specific thematic categories, then a service composition layer builds the necessary paths between processing components to feed the application service layer which is in charge of representing such outcomes or perform further contextual processing for decision-making, for example. Nevertheless, the *intelligence* of the platform claimed by the authors is quite restricted and static since the raw data classification process seems to be drove by a predefined set of application domains and no automatic nor run-time service composition means are referenced, hampering its applicability to dynamic scenarios.

The SOFIA (Smart Objects for Intelligent Applications) project [5] proposes a service oriented architecture that process the information coming from the embedded world (as a stream of events). The architecture for smart city has two main building blocks: the KBPs (Knowledge-Based Processors) that process the events and SIBs (Semantic Information Brokers) that play the role of event channels. There exits a cooperation between KBs and SIBs and among SIBs in order to build the necessary semantic net.

As mentioned in the beginning of this section, it is necessary to move to specific application domains to find an exhaustive analysis of requisites on what a middleware for smart cities should look like. For example in [6], An agent-oriented middlewares for distributed smart cameras is presented. Two aspects are introduced in this work that were not considered above: dynamic loading/migration of processing tasks and collaborative work between nodes at the physical layer. In Civitas, the deployment and operational services [7] are key since they are tightly related with the dynamics of the system, its adaptability and maintainability in the the ever changing smart city environment. It is worth noticing that the Civitas approach allows spanning the kind and type of sensing devices that can implement these features due to its minimal footprint.

Very few approaches for Smart City solutions focus on the concerns of service developers. In our view, we share with [8] that this is one of the main challenges, and Civitas is working for. Service interoperability and deployment mechanisms, for example, should be open, standard and easy-going, facilitating the development of smart applications. If the master lines for smart city software are not devised taking into account these principles, we will incur in the same mistakes that, for example, are described by Lee et al. in [8] for the U-Cities initiative: *islands of services* that must be later integrated using a centralized approach.

## III. Smart city ecosystem

The ecosystem notion of a smart city is an abstraction that comprises the IT infrastructure deployed by governmental institutions all over the city, such as semaphores, traffic sensors, cameras, public wifi networks, etc. The Civitas platform counts on all these sources of information and actuation as its raw elements from which smart city operations can be articulated.

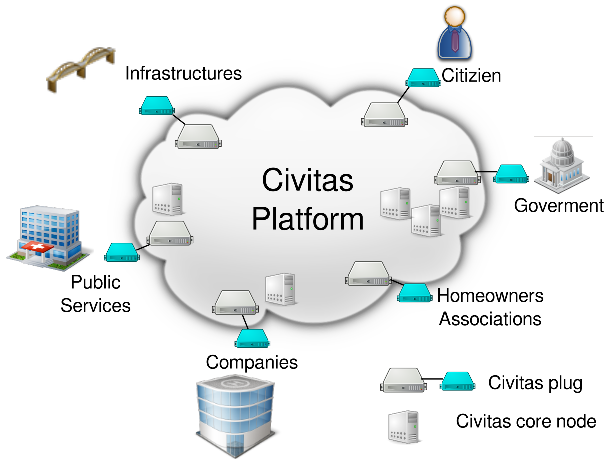

Fig. 1. Smart city ecosystem

In order to fulfill the vision of the future smart city, governments have to start considering investment in IT infrastructure as relevant as providing other public services such as street pavement, public lighting, supply networks, etc. In this sense, IT infrastructures should be scheduled, designed and deployed as a basic service.

The picture envisioned in this article sees the Civitas platform at the core of the IT infrastructure deployed in the smart city. In this sense, the Civitas platform orchestrates the different entities connected to the platform such as citizens, companies, homeowners associations, or even public institutions. The way how any of these entities can connect to the Civitas platform, and therefore be part of the overall platform, is through an element known as *Civitas plug*. A *Civitas plug* is a device, properly certified to provide and to consume information to/from the smart city. The *Civitas plug* can therefore be seen as a way of certifying that all consumers and producers of the smart city hold the appropriate credentials to do so. Examples of devices that can work as *Civitas plugs* include smart phones, residential gateways, company servers, etc. Depending on the type of device, its conversion into a *Civitas plug* can vary from simply installing an smart phone app to installing a Civitas software manager that enable additional services to be deployed in that device. In any case, it is important to highlight that whenever a device is connected to the Civitas platform, part of the device becomes part of the smart city platform, as depicted in Fig. 1. More specifically, from the software layer point of view all these devices have installed a small run-time middleware application that turns them into objects of a general object-oriented middleware.

Fig. 1 depicts a simplified version of the envisioned smart city ecosystem in which the Civitas platform is found at the core. The Civitas platform is not only fed with external entities but it also counts on its own entities, referred as *Civitas core nodes*. The *Civitas core nodes* work as servers,

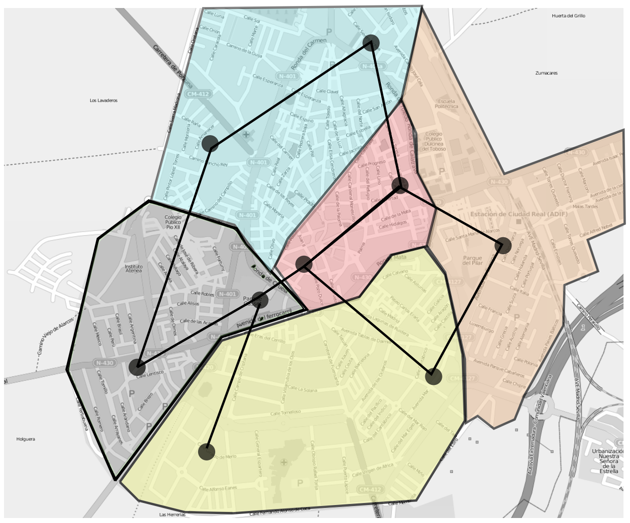

Fig. 2. Smart City district-based infrastructure

generally supported by governmental and public institutions, in which a wide range of services are deployed, such as security strategies, public data services, city layout modeling services, or the platform for deploying services running in these nodes. More specifically, these nodes provide useful services to any of the smart city entities. However, the offered services are not only limited to those provided by public institutions but, on the contrary, any private company can also contribute to this core infrastructure according to the service they provide/require to/from the city. Nonetheless, an arising issue turns up regarding how communication among heterogeneous devices and services is supported.

This issue is easily resolved thanks to the abstraction provided by the middleware layer. From the service developers point of view, any platform entity is seen as a distributed object of the object-oriented middleware. In this sense, from the physical layer point of view, how different devices communicate each other depends on the specific city communication network infrastructure. This aspect is however standardized and made transparent to final service developers, who delegate that responsibility to the middleware layer.

An additional arising issue is how to deploy the different *Civitas core nodes* all over the platform. In this sense, the core infrastructure should be deployed according to criteria such as city population distribution, business centers situation, critical infrastructure, etc. For example, a good criterion could be dividing the core infrastructure according to the administrative districts. Fig. 2 represents the layout of a small city, in which *Civitas core nodes* have been deployed according to administrative districts.

Therefore, the Civitas platform is used to communicate, deploy and manage services from government and companies. Always keeping in mind the object-oriented model, we will manage this platform as an IT cloud.

## IV. CIVITAS DESIGN PRINCIPLES

This section describes the main design principles and characteristics exhibited by Civitas, as response to the weaknesses identified in current middleware solutions for Smart Cities, pointed out in Section II: (1) unsuitability of generic middlewares for resource-constrained devices; (2) heavy communication protocols that, despite being compliant with standards, eats up an important part of the computing and battery resources in autonomous systems; (3) limited flexibility, and run-time management of the platform; (4) absence of intelligence; an undefined development model and (6) lack of integration of high performance computing devices (e.g. FPGAs)

These design principles and major characteristics are described underneath:

**Everything is a software object**, from a sensor attached to a 8-bit processor to a powerful algorithm implemented in a FPGA. All objects are accessed by an interface which is invoked throught an efficient binary protocol.

**Low footprint** of the middleware stack to make it feasible the *in-place* implementation of Civitas plugs in low-cost resource-limited devices such as sensors, actuators, WSN nodes, etc. The functionality of these tailored Civitas plugs do not differ from other devices or platforms, allowing: direct communication from/to the Civitas cores or other Civitas plugs; and *hot* deployment of service software. Details about this low-footprint integration can be found in our previous work [9].

**Ad-hoc hardware processing platforms** can be used in Civitas. Special Civitas plugs and cores have been developed in order to integrate reconfigurable hardware platforms such as FPGAs (Field Programmable Gate Arrays) in our smart city middleware. FPGAs adds high performance capabilities to the Civitas infrastructure. It is then viable *in-platform processing* of huge amount of data without a centralized infrastructure (i.e. data-centers), saving in investments and electricity bill costs due to the excellent operation per watt and dollar ratio. Moreover, reconfigurable hardware is flexible and the algorithms running in it can be replaced in the future, making it the most of the initial inversion. Hardware in-depth details about how to integrate a FPGA in a object-oriented middleware is shown in our previous work [10].

**Native support of audio and video** sensing platforms and applications. In this regard, Civitas adds stream-based communications and a component model to transmit and process audio/video dataflows.

**True intelligence, common sense reasoning**. Context-aware applications and services for smarter cities need a higher degree of intelligence than the one currently offered by rule-based systems. Moreover, the latter solutions lacks of the adaptability to unforeseen situations since it relies on a set of static, hard coded premises. In Civitas, common sense reasoning [11] is introduced in some of its services in order to provide the most appropriate response.

**Independence from the city layout**. The main goal of conventional middleware is usually to abstract communications, networks, operating systems, in brief, the platform details in order to offer a homogeneous view of the IT infrastructure. In

TABLE I. STANDARS AND GUIDELINES USED IN CIVITAS

| Format | Used for | Defined by |
|---|---|---|
| Slice language | Service interface definition | Ice middleware |
| Mobile Location Protocol | Position, areas, polygons, etc. | Open Mobile Alliance |
| ISO 8601 | Date and Time | ISO |
| AV-Streams | Audio & Video sources | Object Management Group |
| OSM format | City layout | Community |

Civitas, it is introduced a new level of *transparency* that we called *city layout transparency*, which has not been considered before. City services should not be exposed to the city layout in order to be deployed or operate in the right geographical context.

**Standarization**. Civitas considers the use of existent protocols and tools whenever its utilization does not break any of the previous design principles. For example, for all related with geographic data representation we are based on Mobile Location Protocol (MLP) from Open Mobile Alliance [12]. However, the aim of Civitas is to define a set of standarized interfaces and processes for Smart City application development. Table I shows some of the standars and guidelines used in Civitas.

The concept of object is at the foundations of the Civitas middleware and it has been used to model:

- **Functional interfaces**. Every entity in Civitas exposes its functionality as a set of methods regardless its nature. The access to sensors, actuators, cameras, persons or hardware components in a FPGA is performed through the invocation of such methods.The interfaces defined in Civitas are expressed in Slice language. Slice is an interface description language used in ZeroC ice [13], a generic object-oriented distributed middleware.
- **Events and channels**. Civitas also supports event-based communication. Event channels are objects that implements the publish- subscribe pattern. It is worth noticing that every event is tagged with geopositional and time information that is used by the brokerage system for synchronization and delivery optimization purposes.
- **Data-flow processing and composition**. In Civitas, a *component infrastructure* has been developed to ease the implementation of data-flow applications or services such as multimedia streamming or complex event processing. A component is a stateless objects with an standarized interface in a certain domain. Data delivery and synchronization between components is responsibility of Civitas, freeing the developer of such task. Therefore, it is easy to build complex and scalable solutions by means of composition.
- **Services**. This is the principal developer's view. A service is a bundle composed by either a Civitas component or object, Civitas development process is explained in detail in section V.

### A. Addressing objects and layout representation

From the software developers point of view, any object in Civitas is identified by an IPv6 address plus object ID. This

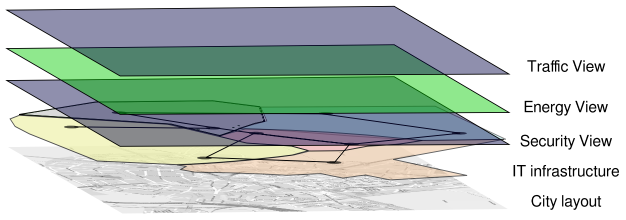

Fig. 3. Smart City layer vision

UID is used to locally instantiate a proxy object representing any distributed object.

In a Smart City, a service is almost always related to a location and the environment that surrounds it. The description of space and location is therefore a relevant feature of Civitas. In this sense, Civitas complies with the OSM format, already employed in numerous open projects like OpenStreetMap. Using the same standards as projects as OpenStreets enables Civitas to get richer information about locations. For example, in OpenStreetMap project the community agrees on certain key and value combinations for tags that are informal standards. This tags allows Civitas to get additional information about uses of the spaces in the city, type of buildings, limits, etc.

### B. Interfaces and service description

As previously stated, a set of simple interfaces are defined to play the same role that POSIX interfaces play in operating system market, that is, to homogenize basic service design, improve the portability and maintainability. We can see an example of this interface specification:

```
module Civitas{
  module Traffic{
    enum State {Red, Yellow, Green};
    interface Semaphore{
      void setState(State st);
    };
  ...
}
```

In this sense, any object devoted to control a traffic semaphore should implement the interface listed above. It is important to highlight that implementation and communication details are hidden behind the interface.

Similarly to the interfaces designed for the traffic semaphore control, Civitas considers additional profiles devoted to monitor and manage certain aspects, such as semaphores, barriers, and sensors for the traffic monitoring profile. Any manufacturer of semaphores (Civitas compliant) should implement, embedded in the semaphore, a software object according to semaphore interface defined in the traffic management profile. For legacy devices and/or protocols, a wrapper can be defined in order to integrate in a seamless way, any device already deployed. Currently, interfaces are being designed for the following profiles: smart grid, traffic management, water flow, environment control, structural monitoring and security. These profiles will represent different views of a city (fig 3).

Along with the functional interfaces defined in the profiles there are two additional common interfaces that have to be implemented by each relevant object in Civitas. First, a property manager interface for accessing the general properties of the service (e.g manufacturer, version, geographical position, covered area, etc.). The interface defines methods to access to the name of the property, its type, its value and security properties (who has access to consult and/or to modify the property). Each profile, along with the functional interfaces, defines a minimum set of characteristics to be defined.

The other interface to be implemented is a *Listener* one. Any object interested in a change of the status of a specific Civitas object which control a, for example, semaphore must implement its *Listener*. Additionally, the semaphore object should implement the SemaphoreAdmin interface to manage *listeners*. Both interfaces are defined in the appropriate profile.

Continuing with semaphore example, in the Traffic profile we can find[1]:

```
interface SemaphoreListener{
  void Status(State st);
};
interface SemaphoreAdmin {
  void addListener(SemaphoreListener* listener);
  void removeListener(SemaphoreListener* listener);
};
```

The idea behind this subscription mechanism is to provide developers with a generic mechanism to ”listen” what happens in each object of Civitas. For example, any change in a semaphore object should be communicated to all objects subscribed invoking the method Status with the new status. With this mechanism we also enable a powerful mechanism for information federation, filter, processing, composition, etc.

## V. Service development

This section guides the reader through a trip around the Civitas architecture, and more specifically through a homeland security related service development.

Figure 4 provides an overview of Civitas under a specific service development process. The proposed case study considers a scenario in which a service for license plate based vehicle tracking is required. First, a source of digital images is required in order to undertake the tracking operation. To this end, the traffic surveillance cameras are going to be used as video providers. Once the video images are available, a hardware version of the image analysis module is employed in order to determine whether the tracked license plate is present or no in the analyzed video sequence. Finally, those sequences in which the tracked license plate is recognized conforms the video stream sent back to a user device, in which the tracking is being followed in real time.

From the software developers point of view, the traffic surveillance cameras belong to the city an therefore any service intended to access them have to use a universal ID (UID), currently formed by IPv6 plus object ID. This UID is used to locally instantiate a the proxy object representing a distributed object, in this case a camera. In the instantiating process, a set of credentials have to be provided so that the middleware permits the object instantiating with the appropriate permissions

[1]For the shake of readability, exception definitions and error management code have been removed from the interface examples

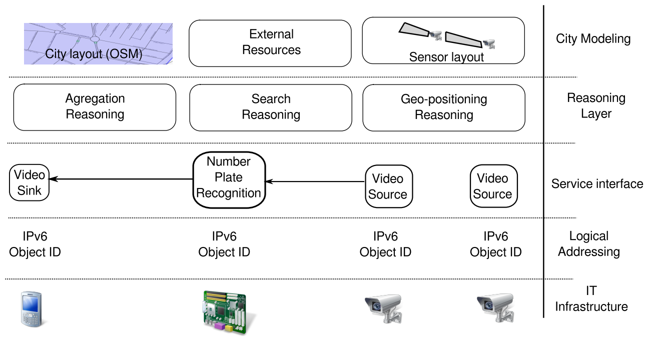

Fig. 4. General overview of Civitas

based on the security level of the instantiating request. For example, in this particular case, if the client accessing those cameras would be the police, their security level would allow them to change the configuration or properties of the camera. On the contrary, other clients such as a company providing traffic congestion free paths services, could not move and/or zoom the camera.

The interface for controlling the camera is predefined and compliant with the AV/Stream specification from OMG. The next issue to take care of is how to integrate the camera video stream with the algorithm that performs license plate recognition, for the sake of performance implemented in VHDL. The developer of the VHDL algorithm is responsible for providing an interface that abstract the implementation details from those using the algorithm. This interface is specified using Slice. This interface is provided to a compiler which automatically generates a VHDL wrapper for the algorithm. Additionally, the Slice definition can be used to obtain the proxy object[2] to the license plate recognition algorithm running in the FPGA. By means of the UID of the algorithm, this can be used as any other software object.

Finally, it is necessary to implement a visualization tool to show the tracking video stream in the mobile device used in this service. Once again, the Sink interface, defined in the streaming specification as guideline for my viewer, can be used.

## VI. Injecting common sense in the middleware

The proposed Civitas framework does not only cope with service developer issues but it also provides a set of high level services that can be combined among themselves resulting in new services with new functionality.

This section describes the process of combining middleware basic services into more elaborated ones. In this sense, the proposed approach overcomes the limited classic approaches for service composition such as web services, service input/output pattern matching, or rule-based systems. On the contrary, it advocates for a more flexible and intelligent strategy based on a common-sense knowledge strategy to drive the composition process. The work in [14] provides a thorough description of the employed solution. For the sake of

[2]In java, python, C++, etc.

simplicity, this section focuses in describing how a common-sense knowledge supports the service composition process. In order to do so, a case study is employed to drive the explanation.

The case study describes the same scenario presented in previous section, in which a license plate tracking service is required and no specific basic service exists to do so. However, the Smart City counts with enough resources to undertake such task, such as video cameras and a license plate recognizer system. The combination of both services can give rise to a new service to fulfill the license plate tracking service. It is tempting to think that this type of situation can be addressed by providing the middleware with a list of basic service that when combined provide new functionality. This solution would suffice to recognize license plates in video streams, but it will fail to track a specific subject. To do so, a more intelligent solution has to be worked out.

It has to be highlighted that the proposed solutions resorts to a reasoning engine capable of combining information about the city, such as sensor locations and capabilities or street maps, with high level information about how the work works, so called common sense. This combination of information enables the reasoning engine to devise a solution to track license plates based on the knowledge it holds about the video camera locations and how a car movement is ruled.

The common-sense law of inertia is an example of rule that describes how the world works. This rule states that objects tend to remain in the same state until affected by an external action. In this sense, if the car is moving in one direction it tends to continue moving in the same direction. This rule is combined with the rule that relates movement, space and time so that the location of a moving object can be calculated based on its speed, direction and its original location. These and similar rules are generally referred as common-sense knowledge. These rules enable the system to identify the specific video camera in which range the license plate is going to appear. If the car makes a turn, the system is still able of figuring out the new location. When the license plate is not recognized in the video stream in which it was expected, the system recalculates the possible locations given the last known position, the possible street directions, and the car speed. However, these operations are not hard-coded in the middleware. On the contrary, the common-sense reasoning engine counts on a description of how physical systems behaves and evolves. This information is what entitle the reasoning engine to track a moving object through the camera network or CCTV that populates the city.

## VII. Conclusion

In this paper we presented our ongoing work achieved to design a middleware for smart cities. In summary, Civitas:

- integrates devices ranging from small footprint devices to flexible hardware devices (e.j FPGA).
- develops a set of profiles to create common POSIX-like interfaces for developers
- proposes a set of standards and tools that can be the vertebral column of the future smart cities ecosystems.
- includes common sense reasoning services in the middleware simplifying developers design.

The current effort is focused on creates a sandbox to provide developers a safe environment for developing and testing services. Also the security model has to be designed since, privacy issues regarding citizens, has to be considered carefully.

## Acknowledgment

This research was supported by the Spanish Ministry of Science and Innovation and CDTI through projects DREAMS (TEC2011-28666-C04-03), ENERGOS (CEN- 20091048) and PROMETEO (CEN-20101010).

## References

[1] M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris, "Smarter cities and their innovation challenges," *Computer*, vol. 44, no. 6, pp. 32–39, June.

[2] A. L. Tu"n, H. Quoc, M. Serrano, M. Hauswirth, J. Soldatos, T. Papaioannou, and K. Aberer, "Global sensor modeling and constrained application methods enabling cloud-based open space smart services," in *Ubiquitous Intelligence Computing and 9th International Conference on Autonomic Trusted Computing (UIC/ATC), 2012 9th International Conference on*, Sept., pp. 196–203.

[3] M. Fazio, M. Paone, A. Puliafito, and M. Villari, "Heterogeneous sensors become homogeneous things in smart cities," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, July, pp. 775–780.

[4] Z. Khan and S. Kiani, "A cloud-based architecture for citizen services in smart cities," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, Nov., pp. 315–320.

[5] J. Wan, D. Li, C. Zou, and K. Zhou, "M2m communications for smart city: An event-based architecture," in *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*, Oct., pp. 895–900.

[6] M. Quaritsch, B. Rinner, and B. Strobl, "Improved agent-oriented middleware for distributed smart cameras," in *Distributed Smart Cameras, 2007. ICDSC '07. First ACM/IEEE International Conference on*, Sept., pp. 297–304.

[7] B. Rinner, M. Jovanovic, and M. Quaritsch, "Embedded middleware on distributed smart cameras," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 4, April, pp. IV–1381–IV–1384.

[8] C. Lee, S. Baik, and C. Lee, "Building an integrated service management platform for ubiquitous cities," *Computer*, vol. 44, no. 6, pp. 56–63, June.

[9] F. Moya, D. Villa, F. J. Villanueva, J. Barba, F. Rincn, and J. C. Lpez, "Embedding standard distributed object-oriented middlewares in wireless sensor networks," *Wireless Communications and Mobile Computing*, vol. 9, no. 3, pp. 335–345, 2009.

[10] J. D. Dondo, J. Barba, F. Rincn, F. Moya, and J. C. Lpez, "Dynamic objects: Supporting fast and easy run-time reconfiguration in fpgas," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 1 – 15, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762112000860

[11] S. Fahlman, "Marker-passing inference in the scone knowledge-base system," in *First International Conference on Knowledge Science, Engineering and Management (KSEM'06)*. Springer-Verlag (Lecture Notes in AI), 2006.

[12] O. M. Alliance, "Mobile location protocol," Open Mobile Alliance, Tech. Rep., 2011.

[13] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, 2004.

[14] M. J. Santofimia, S. E. Fahlman, X. del Toro, F. Moya, and J. C. López, "A semantic model for actions and events in ambient intelligence," *Eng. Appl. of AI*, vol. 24, no. 8, pp. 1432–1445, 2011.