Ubiquitous Capability Access for Continuous Service Execution in Mobile Environments

Ramon Alcarria[1], Unai Aguilera[2], Tomas Robles[1], Diego López-de-Ipiña[2], Augusto Morales[1]

Dept. of Telematics Engineering[1]
Technical Univ. of Madrid
Madrid, Spain
ralcarria,trobles,amorales@dit.upm.es

Deusto Institute of Technology – DeustoTech[2]
University of Deusto
Bilbao, Spain
unai.aguilera,dipina@deusto.es

***Abstract*— Dynamic mobile environments are characterized by integrating a set of devices (sensors, actuators, intelligent terminals) whose availability continuously changes. In addition, these devices are heterogeneous in their technology, access protocols and in the format of the exchanged data. This paper proposes an architecture that allows continuous and ubiquitous access to capabilities, i.e. the functionality provided by these devices, to solve some of the problems associated to dynamic mobile environments. Focusing on the issue of continuous capability invocation, this work uses a set of software engineering patterns for defining the communication architecture, which support the most common types of resource access. Finally, a contribution to resource access management is described, through the implementation of two algorithms and their evaluation through two different use cases.**

Keywords-resource management; prosumer; software engineering; ubicuitous computing; communication paradigms

I. INTRODUCTION

Uniform access to resources and devices is one of the most discussed topics in ubiquitous computing related work. This is demonstrated by the large amount of work on communication middleware available today [7] [8]. These middleware solutions often face the problem of device interoperability, due to the wide heterogeneous nature of existing devices.

This work covers the design and validation of a solution for continuous capability access whilst service execution is taking place in ubiquitous environments. In this paper, we propose a resource access middleware that integrates a set of access technologies and communication paradigms, which are commonly used and requested by available capabilities.

The presented middleware contributions are part of the mIO! project, which aims at the provision and consumption of prosumer services in a mobile environment. For further information about the prosumer concept and the mIO! architecture the reader is referred to our previous work [1].

A. Mobile prosumer environment

In our view, the user is placed in the centre of the device environment. Using their smartphone, users can design their services by selecting an appropriate set of components from a catalogue and, with the help of a creation tool, connect and configure them. The generated service is able to use the available functionalities which are offered by surrounding entities. Users, while moving, will interact with close elements and the middleware will try to guarantee that the service execution is maintained even though the used elements may change or disappear.

Mobile prosumer environments establish some requirements that determine the design of the proposed architecture. Focusing on non-expert users, a high level of abstraction is required in order to enable users to create their own services in an easy way. Besides, the architecture needs to adapt to changes in the availability of resources and services, as well as to provide a communication infrastructure for uniform resource access.

The provision of a higher level of abstraction for prosumer users leads to the following concepts: *Service*, as a unit supplied and consumed by the prosumer, *Component*, which represents a basic and functional unit used by a service, and *Capability*, which is the implementation of the functionality defined by a component and provided by some element (hardware or software). It is also necessary to introduce the concept of *Orchestration*, which manages the interaction between the different components of a service, *Resolution*, which manages the association of capabilities to components and, finally, *Invocation*, which provides a uniform access to the infrastructure capabilities.

Based on the requirements of the mobile prosumer environment, the service must present a logical structure defined by different layers, described in the next Section. Section III analyses the communication paradigms currently used in communication middleware. Section IV describes the overall architecture, which has been designed using various design patterns in order to meet the requirements imposed by the ubiquitous environment and the studied communication paradigms. Section V and VI describe the integration between the resolution and capability invocation processes while Section VII makes a contribution to the problem of resource and connection management. Finally, the paper concludes with a validation of the designed system, related work and some conclusions of the proposed solution.

II. SERVICE LOGICAL MODEL

A service can be defined in many ways, depending on the state of the life cycle in which the service is. We define the logical structure of a service by different levels: the **service level**, the **component level** and the **capability level**. To facilitate the understanding of these concepts we present the example of a simple prosumer service, called *Sport Tracker*, which aims to access the location information of a user and to represent it on a map along with information about his

heartbeat. This service consists of three components: a *Map* provider, a *Location* provider and a *Pulse* provider. These components provide an interface that must be dynamically bound so services can execute them. For example, the *Location* component needs to be resolved into a GPS device or a GPS capability (e.g. offered by a mobile phone) in order to obtain the information about the user's location.

Fig. 1 shows the proposed service logical model, adapted to this simple example service. The three-level model is explained below:

First level is the **service level,** where services are seen as abstract resources with the capability of performing tasks where the different internally used components remain hidden. Service behaviour and orchestration logic are described by a Service Description Language (SDL) document.

Continuing with the **component level**, the service is split into different logical units called components, which provide a higher level abstraction to make the creation process easier, and are implemented by an available capability depending on the service execution conditions. In order to make a step towards the new mobile prosumer environment, developers must implement and publish a big number of different components, which cover all the creation possibilities that a user could wish.

The **orchestration process** manages the interaction between components, i.e., how the components are interconnected to compose services, how are the components managed and the data interchanged inside the architecture and how the components obtain the appropriate capability to implement their functionality. This process takes place during creation time and is performed by the creation subsystem.

Finally, in the **capability level** a service is seen as a set of capabilities which offer the functionalities that are demanded by the service. The capabilities generally access to local (in-device), nearby or remote resources and are designed to achieve the components' objectives. The division between the component and the capability models is made for two reasons. First, orchestration logic is decoupled from implementation. This way, a component can be resolved into different capabilities depending on the service execution conditions and the preferences given by users in the creation process. Second, components are defined as functionalities that are easy to understand for non-expert users in order to help them to create simple services.

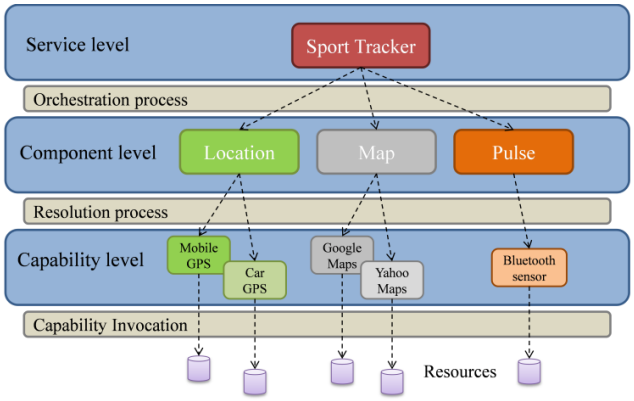

Figure 1. Service Logical Parts.

The **resolution process** assigns, during execution time, each component to the best available capability that can implement it. This process takes place in the *harmonization subsystem*. Finding the optimal capability depends on multiple factors, for example, the configuration options established by the user during creation time or component requirements.

In the *Sport Tracker's* example scenario, the *Pulse* component always calls the Bluetooth sensor capability whereas the *Map* component can choose a map provider based on user preferences or service restrictions.

The **capability invocation process**, which is the main topic of this work, is the one responsible for requesting and obtaining all the required resources. An effective coordination between the resolution and the invocation processes will enable mobile continuous service execution in dynamic environments. In these environments, the capabilities can appear and disappear at any time and the wide variety of sensors, actuators and other devices makes necessary to design different mechanisms for the access and invocation of heterogeneous capabilities. This process takes place in the Capability Middleware and is explained in section IV, along with the Creation, Execution and Harmonization subsystems.

III. Communication Paradigms in Capability Access

Most mobile middleware solutions for resource access include only one communication paradigm, ignoring the fact that the scenarios in such environments are extremely varied. Therefore, we designed a capability access middleware that includes several communication paradigms, classified under three criteria:

- *Coordination Mechanism*: Differentiates between synchronous or asynchronous communication models.
- *Notification Model*: determines if consumers explicitly retrieve new messages or are notified when new messages are produced (synchronous or asynchronous notification).
- *Connection Orientation*: many middleware platforms employ the notion of message as a fundamental building block (Message-oriented Middleware). Other middlewares use the concept of session to communicate with resources [9], providing channel and transaction management [6]. A connection oriented middleware uses sessions instead of single message interchange as the most natural method of communication.

The studied paradigms are described below. Table 1 shows the features of these paradigms according to defined criteria and the requirements that they impose on the design of system's architecture, presented in the next section.

Request-Reply model: a synchronous model is adopted in situations that require the communicating entities to be connected simultaneously. The sending entity delegates the control to the receiving entity, which performs some processing and responds, allowing the first to continue its execution.

TABLE I. COMMUNICATION PARADIGMS AND FEATURES

Paradigm	Coordination mechanism	Notification model	Connection Orientation	Connection initiated by	Design requisites
Request /Reply	Synchronous	Synchronous	✓	Consumer (Middleware)	Client-server model
QP2P	Asynchronous	Synchronous	×	Producer/ Consumer	Message are retrieved in a predefined order
Tuple Spaces	Asynchronous	Synchronous	×	×	Intermediated by a tuple space service
Publish-Subscribe	Asynchronous	Asynchronous	✓	Event Channel Service	Event channel service must be external

QP2P (Queue-based Point-to-point Paradigm): distributed queues are used for sending and receiving messages. Using this model, messages are obtained in a predefined order based on queue type (FIFO, LIFO, etc). Producers and consumers are fully decoupled.

Tuple Spaces: this paradigm provides a distributed shared memory for the exchange of tuples between various entities. Like QP2P, Tuple Spaces uses an indirect model, mediated by a *Tuple Space Service*, but in this case the consumer gets messages (tuples) by requesting them directly to this service.

Publish-Subscribe: communicating entities exchange messages by publishing events and subscribing to them. An intermediate service called Event Channel [13] registers the subscriptions and forwards the events published. In pub-sub systems, message delivery depends fully on the actions of the receivers, which frequently are unknown to the senders.

IV. OVERALL ARCHITECTURE FOR SERVICE ORCHESTRATION, RESOLUTION AND INVOCATION

The service provision and consumption platform described in this paper consists of a set of subsystems (Fig. 2), which perform the functions of orchestration, resolution and capability invocation. The design of these subsystems is affected by the communication paradigms that address the capability access, which relate to the need for external services to manage deployed tuples and publication/subscription records (4a) and the environmental requirements for mobile prosumer users, stated in the introduction section.

Design patterns are used to address the requirements of the prosumer environment in an elegant and effective way [10]. The application of these patterns can impact the ability of systems to achieve their quality attribute goals, and, therefore, they affect the system architecture and help to address key issues that are resolved in the following sections. The proposed subsystems are:

Creation Environment (1): provides mechanisms for service creation and composition by non-expert users through component interconnection and customization. Therefore, this environment performs the service orchestration process, resulting in the generation of the SDL document (1a), which describes the set of components required for the service to run properly, in addition to a number of restrictions that will be used by the harmonizer for optimal component resolution into capabilities.

Execution Environment (2): is responsible for processing the SDL document and generating the graphical visualization of the service. This environment starts the process of component resolution, which is carried out by the harmonizer subsystem.

Harmonizer (3): its main function is to make the matchmaking between the component to execute and a compatible capability (3a) from those available at the capability repository (3b). The aim of the Harmonizer is to select the best capability for each component grounded on different sources of information (user profile, customization options in components, context information, capability definition and so on).

Capability Access Middleware (4): performs capability discovery and invocation tasks and manages the events received, providing a uniform interface to the Harmonizer for data access (4b). To deal with asynchronous communication we have chosen to use the Proactor pattern [17], that uses the inversion control mechanism (in callback methods, 3c) to decouple application-independent asynchrony mechanisms from application-specific functionality. Callback methods are invoked when an event appears, such as a message arrival to the Access Middleware through a connection to a capability and perform application-specific processing. The component resolution process as well as the synchronous and asynchronous invocation management is further explained in Section IV.

An important requirement to be considered in

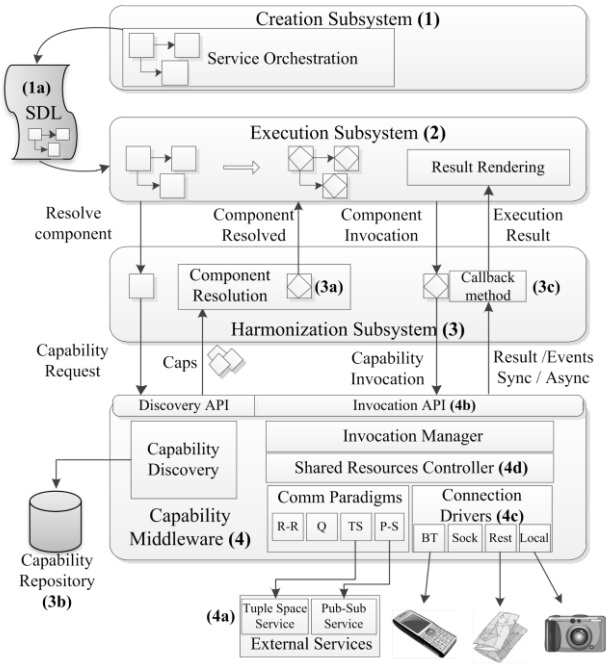

Figure 2. Overall Architecture.

```
Object invokeSync (int capabilityId, String invocationArgs[]);
void invokeAsync (int capabilityId,  String invocationArgs[],
CapabilityHandler cHandler);
void cancelAsync (int capabilityId);
```

Figure 3. Capability invocation API

environments with a large amount of heterogeneous capabilities is how to provide mechanisms for effective reuse of communication technologies during resource accessing. The Capability Access Middleware represents the fundamental level of reusability and follows the Acceptor/Connector pattern [17], which decouples the connection among tasks from the processing performed once the connection was carried out. This is achieved using various connection drivers (4c). In Section V we develop the key aspects of event and connection management.

An access middleware for mobile environments is characterized, from the ubiquitous computing point of view, by the large number of connections and disconnections that occur in a continuously changing environment and the appearance and disappearance of new capabilities. Proper management of resources is needed for efficient capability access. This management is facilitated by the use of the Monitor Object pattern [17] (4d) which synchronizes method execution to ensure that only one method runs within an object at a time. It also allows an object's methods to cooperatively schedule their execution sequences.

Section VII describes resource management in detail and the optimization algorithms we have defined for the Middleware.

V. Continuous Component Resolution in Mobile Environments

The Harmonizer subsystem provides a continuous component resolution process. The resolution is carried out using capabilities that are available in the user's current context. These can be capabilities which are accessible in the own mobile device (e.g. GPS device) or by proximity (e.g. printers, screens, etc.) or capabilities which are globally accessible using telecommunication networks (e.g. 3G, GSM, etc). Since these capabilities may disappear, the resolution process not only takes place at the beginning of each capability usage but also occurs when the Harmonizer determines that a change of capability is appropriate or necessary (e.g. a user with a GPS device enters an indoor environment). This subsystem also incorporates other advanced features, such as the suspension of running services and the detection of those new available capabilities that impeded a service execution.

The selection of the optimal capability for each component is done taking into account the user's preferences (e.g. higher priority for cheapest or closest devices) and component and capability descriptions, expressed using a XML language (see [1] for more details). A set of conditions can be defined to act as restrictions over a property, using comparison operators (i.e. ==, >=, <=, =, !=). These conditions are converted to other query languages like SPARQL or SQL to perform the matching process.

Once the resolution process has finished, the Harmonizer is responsible for transmitting any component invocation

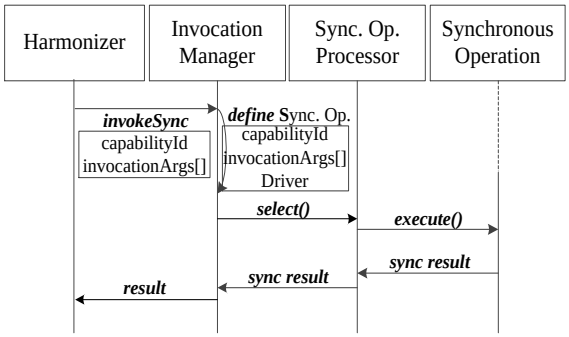

Figure 4. Interaction diagram of synchronous invocation.

performed by the execution subsystem to the Capability Middleware and returning the execution results to the rendering module (Fig. 2). The messages exchanged among the Execution, Harmonization and Capability Middleware subsystems are Java Objects, which encapsulate the transmitted parameters. The Invocation API (Fig. 3, 4b) contains generic methods for capability invocation, distinguishing between synchronous and asynchronous invocations.

The *invokeSync* method returns the result of the synchronous invocation through the Invocation Manager (Fig. 4). The *capabilityId* parameter indicates the selected capability and the *invocationArgs[]* parameter contains the name of the method to be invoked and the parameters required for the method to run properly. This method returns a Java Object as a result, which is transmitted to the execution environment. After the Invocation Manager receives the synchronous access request, it creates a Synchronous Operation, designed by following the Command design pattern [17], which encapsulates all the necessary information to process the request (capability identification, *capabilityId*; arguments needed to perform the invocation, *invocationArgs[]*; driver identification, *driverId*) and defines an *execute()* method. This method performs the invocation request through a Driver, which controls the access technology, using *capabilityId* and *invocationArgs[]*. After that, the Invocation Manager selects a Synchronous Operation Processor to perform the Synchronous Operation in a new thread. There exist a limited number of Operation Processors, according to the number of Communication Paradigms that this middleware supports.

In the case of the asynchronous call (*invokeAsync* method), the common solution is to use a multi-thread technique to perform operations in parallel (synchronous multi-threading). Every requested operation is executed in a thread that is scheduled by a manager. It is easy to write code for one thread, but the synchronization among many threads is a challenging task [11]. Nevertheless, in our work, the inversion mechanism provided by the Proactor pattern is used. The Proactor architecture pattern demultiplexes and dispatches completion events that are triggered by the completion of asynchronous operations. These completion events are dispatched to concrete service handlers that process them. Fig. 5 shows the developed implementation of the Proactor pattern for asynchronous communication between the Harmonizer and the Capability Middleware subsystems.

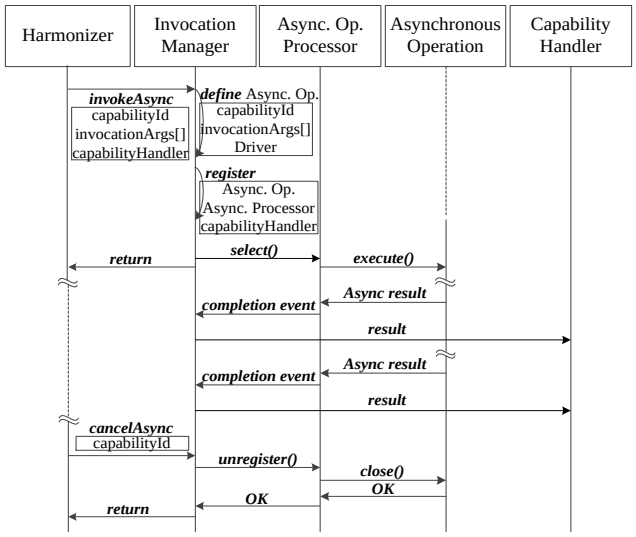

Figure 5. Interaction diagram of asynchronous invocation.

The Harmonizer consumes the API provided by the Middleware and invokes the *invokeAsync (capabilityId, invocationArgs[], capabilityHandler)* method where *capablityHandler* is the reference to an object that can process the asynchronous result once it is received. The *invokeAsync* method is implemented by the Invocation Manager, which has two different roles: on one hand it defines the Asynchronous Operation as in the synchronous case and on the other hand registers the Asynchronous Operation with the *capabilityHandler* provided by the Harmonizer. Thus, once the asynchronous invocation is completed and the result is returned, the Invocation Manager can retrieve the Capability Handler from the registry and send it the result.

Once the registration is performed, the Invocation Manager selects an Asynchronous Operation Processor to perform the Asynchronous Operation in a new thread. When the operation finishes executing, a completion event is generated by the Asynchronous Operation Processor, which notifies the Invocation Manager. Then, the Invocation Manager dispatches to the associated capability handler, which processes the results of the asynchronous operation.

When the execution environment does not wish to receive asynchronous invocation results from capabilities, either because the service is over or the service execution logic no longer requires asynchronous access to data, the Harmonizer uses the *cancelAsync* API method, to cancel the event subscription. This invocation requests the Asynchronous Processor that is processing the Asynchronous Operation to terminate the connection to the capability. Once the connection has been completed, the Invocation Manager unregisters the Asynchronous Operation and returns a message indicating whether everything went well or not.

VI. COMMUNICATION ARCHITECTURE

A middleware that provides a single communication paradigm could not cope with the variety of sensors, actuators, controllers and other devices that act as capabilities in our environment, making their use very limited. This middleware solution offers a set of communication paradigms ranging from the traditional synchronous model to different variations of the asynchronous model.

We define a *Synchronous/Asynchronous Operation Processor* as an entity chosen by the Invocation Manager to perform a sync/async operation. This operation may return an immediate result, as in the case of synchronous invocation or it may generate a series of events routed toward a Capability Handler, which is responsible for their processing. The way events containing invocation results are handled depends on the type of communication paradigm applied; therefore, there must be as many Operation Processors as Communication Paradigms are supported by the Middleware.

In the Capability Access Middleware we have implemented support for Request-Reply, QP2P and Publish-Subscribe communication models. The implications on the proposed architecture are described below:

- *Request-Reply*: the Operation Processor defined for this synchronous model runs directly the invocation operation, blocking the execution and awaiting the outcome, which is returned as a synchronous result. In order to avoid blocking problems on long-lasting requests, the Harmonizer controls the invocation requests using threads.
- *QP2P*: the Operation Processor, through a queue used for sending messages, has the possibility to handle asynchronous capability invocation in an independent way. In addition it can also wait to receive some execution orders in order to make complex capability invocations. By having a queue for the receiving messages, the Operation Processor can return Completion Events composed of several responses. This is useful to send several responses received from the capability in a single message to the upper layers.
- *Publish/Subscribe*: this paradigm provides Subscribers with the ability to express their interest in a topic or set of topics in order to be notified subsequently of any incoming events generated by a Publisher, which match the registered interest. This middleware integrates a topic-based publish-subscribe mechanism with the addition of an external service called Event Channel, which provides storage and management for subscriptions and efficient delivery of events.

Because of the need to establish and maintain connections that use scarce resources in the mobile terminal (Bluetooth stack and ports), a Resource Controller Module has been incorporated to the proposed middleware for connection management, which optimizes connection duration and reduces data access delay. In the previous section, we described the usefulness of the connection drivers to decouple the invocation processing from the technology used for capability access. In order to implement this decoupling we have used the Acceptor/Connector pattern, which defines two entities called Acceptor and Connector. The Acceptor is responsible for creating an endpoint that passively listens to connection requests in a

particular address. The Connector connects to a remote Acceptor. In this pattern there is an element called *ServiceHandler*, which provides a hook method that is called by an Acceptor or Connector to activate the application service when the connection is established. Once a Service Handler is completely initialized by an Acceptor or Connector factory it typically does not interact with these components any further.

Invocation drivers contain an Acceptor and Connector entities. The first one listens to capability connection requests while the latter (that is the one used most often) makes requests over external capabilities. *ServiceHandlers* adapt and uniform the invocation result and deliver it to the Sync/Async Operation Processor for further processing.

Fig. 7 shows an implementation of the communication between the Access Driver, the Operation Processor that executes it and the Resource Controller. In each driver there is a pool of ServiceHandlers, managing information from different types of capabilities. This design seeks to standardize data from heterogeneous devices so that can be recognized by Middleware's upper layers (Harmonizer and Execution subsystem). Between drivers and the Resource Controller two interfaces are defined, Resource API and Connection API, which exchange messages for controlling resources, an issue that we describe in the next section.

VII. Resource Management for Capability Discovery and Invocation

The environment described in our work defines capability access as a fundamental mechanism for prosumer service and enables to obtain the needed functionality at execution time. We consider that these services request access to capabilities for repetitive invocations with a constant frequency. These invocations concurrently use resources of the mobile terminal that can be considered as limited (communication ports and Bluetooth stack). Therefore, we have defined mechanisms for resource management by using the Monitor Object concurrency pattern. This pattern synchronizes method execution to make sure that only one method is executed at a time. Thus, different drivers can concurrently attempt to access a common resource, but an internal mechanism will synchronize access to it, allowing access to one driver at a time. In Java, *synchronized* methods are used for this task. Fig. 6 shows how a driver obtains a resource and uses it: first the driver should contact the Resource API for a Resource Object, then it establishes the priority to acquire the resource and, after using the resource, the driver releases it.

The Decissor module, in the Resource Controller, selects which invocation acquires the resource based on profiles. If the selected profile (by user preferences or depending on the capability type) seeks to reduce energy consumption in the Access Middleware, it will minimize the parameter $\overline{U_{C_x}}$. This parameter defines the average utilization rate of a resource for connections with an X capability, given that maintaining an open connection without being used increases battery consumption (from 6.6 mW in stand-by state to 69 mW in connected state for Bluetooth in the work of Cano et al. [14]). But if the objective is to minimize the invocation delay the middleware adopts a profile that attempts to increase $\overline{U_{C_x}}$, so that connections are always active (see delay analysis in the validation section). We have implemented these two profiles with two algorithms called ESA (Energy Saving Algorithm) and SOA (Session Optimization Algorithm).

```
Resource res = getResource (resourceId);
res.setPriority(Priority.HIGH);
doSomething(resource);
res.setPriority(Priority.LOW);
releaseResource(resourceId);
```

Figure 6. Resource request in drivers.

The ESA algorithm is simple: When the driver requests a resource to perform a capability connection, the Resource Controller blocks the request until the resource has become free. When the driver stops using the resource, it can invoke *setPriority(Priority.LOW)* or *releaseResource()* to indicate that it does not need this resource in a while.

The SOA algorithm is defined below: Be R_x a resource X, Q_{Hx} and Q_{Lx} priority and non-priority invocation queues that use R_x, and I_{C_y} an invocation to Y capability, the Decissor applies the algorithm of Fig. 8 when it receives a resource request.

In order to release and assign resources, the Decissor interacts with the Connection API for communicating to drivers. Finally, there is also a thread that assigns R_x to the first element of Q_{Hx} and, if Q_{Hx} is empty, to the first element of Q_{Lx}.

VIII. Validation

This work has been validated as part of a prototype implementation that consists of a Creation Environment, Execution Environment, Harmonizer Subsystem [1] and Access Middleware, according to the mIO! project's architecture devised for mobile service provision.

In this work we focus in the Ubiquitous Capability Access for Continuous Services Execution in Mobile Environments, in particular, in the Capability Access Middleware and its integration with the Harmonizer.

The developed middleware follows the architecture described in Fig. 2, integrating the Request/Reply, QP2P and Publish/Subscribe communication paradigms and the explained design patterns. The discovery module and some drivers that control capability access have also been

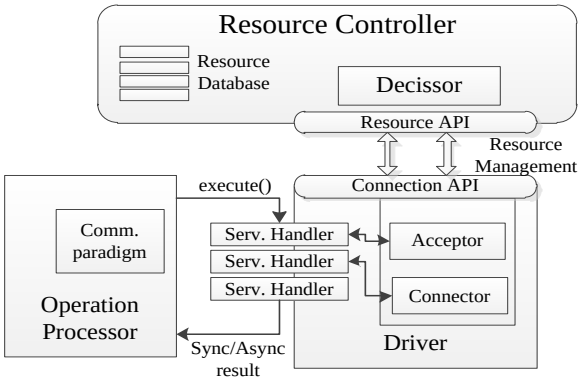

Figure 7. Internal Driver Communication.

(I_{C_y},R_x) setPriority (Priority.HIGH)
1: add to Q_{Hx}
2: if R_x is used by I_{C_z} with Priority.LOW then
3: release(R_x) from I_{C_z}
4: assign(R_x) to first element of Q_{Hx}
5: wait() until R_x is assigned to I_{C_y}
6: return
(I_{C_y},R_x) setPriority (Priority.LOW)
1: add to Q_{Lx}
2: if R_x is not used then
3: assign(R_x) to first element of Q_{Lx}
4: wait() until R_x is assigned to I_{C_y}
5: return
(I_{C_y},R_x) releaseResource ()
1: release(R_x) from I_{C_y}
2: assign(R_x) to first element of Q_{Hx}

Figure 8. Pseudocode of Session Optimization Algorithm.

developed, using REST, Bluetooth (RFCOMM and OBEX), SOAP and Java local access. For this proof of concept we have tested access to Google Maps and Yahoo Maps using REST, control of an UPnP / DLNA network hard drive (Model Media Iomega Home Network Drive) through SOAP and connection to a B600 FRWD heart rate monitor and a BT microX Medical RGB [12] pulse oximeter.

The harmonizer and the middleware have been implemented in Java ME with LWUIT interface and have been tested in a Nokia N97 and Nokia 5800 XpressMusic (O.S. Symbian S60 5º Ed) devices.

As a proof of concept for resource management we present a performance evaluation of capability access using the internal Bluetooth capability (through JSR 82) of the mobile terminal that is executing the Capability Access Middleware. The aim of this study is to compare the behaviour of the Resource Controller for each of the defined algorithms (ESA and SOA) in these two use cases:

Case # 1: The system runs a service that accesses the Bluetooth resource every 6 seconds.

Case # 2: The system runs two services that access via Bluetooth to different capabilities periodically, with a frequency of 5 and 12 seconds.

In these cases we are not taking into account the discovery time and we assume that the Bluetooth service accessed is known. If Bluetooth capabilities were unknown, the Discovery module (which also uses the Bluetooth resource) would be needed. Thus, discovery is modelled as another capability that accesses resources for the Resource Controller's point of view.

We found that the average delay for data access using the studied Bluetooth capability (BT microX Medical RGB pulse oximeter) corresponds to 3953 ms (0.2 standard deviation) and 1988 ms (0.3 standard deviation) if the connection was already established before. Fig. 9 analyzes the value of $\overline{U_C}$ (Average resource usage rate for connections with studied capabilities) for both use cases and ESA and SOA algorithms, knowing that a low value of $\overline{U_C}$ optimizes power consumption, while a value of $\overline{U_C}$ close to 100% determine a lower access delay.

These figures show that the difference between the two algorithms in terms of channel usage for connections is clear. In Case #1 with ESA algorithm the average resource usage for connections tends to 100% as time passes, due to the fact that the Middleware creates a single connection, which holds every invocation (20 invocations in 1 connection for 120 seconds). In the case of SOA, the resource is used just to receive the capability data, which corresponds to about 50% utilization. In Case #2 (which is a more realistic behaviour for a multi-execution environment), the difference in values, although significant, is not as extreme; since in both cases it is necessary to make disconnections (22 versus 15) to release the resource in order to be used by other capabilities.

IX. Related Work

This section concentrates on reviewing previous work on resource access middleware, since the related work in the area of service provision in mobile prosumer environments was previously studied in [1].

There are many types of communication middlewares, Message Oriented (MoM), Remote Procedure Call (RPC), Object Request Broker (ORB) and even Service-oriented Architecture Middlewares [15]. While traditional middleware platforms typically employ synchronous, RPC-style client/server interactions, MoMs provide asynchronous, peer-to-peer style interactions, leading to a more loosely coupled architecture which is more adequate for mobile computing [4].

In ubiquitous computing environments, devices might not be connected at all times. Several proposals take this into account and support that devices enter and leave networks on an ad hoc basis. This behaviour can be modelled by using P2P networks [3], in which devices are peers and communicate via ad hoc protocols. To locate these devices, some content-based techniques are used, such as Distributed Hash Tables (DHT).

Integrating communication paradigms in Access Middleware has been tackled in [4], proposing an architecture which supports the traditional synchronous model and different variations of the so-called asynchronous models. Other works, as GREEN [5], focus in the concept of reconfiguration in continuous execution environments, and provide a reconfigurable middleware (according to application requirements and context information) that supports publish-subscribe interaction types (topic-based, content-based and location-based) but only for one communication paradigm.

Related works in resource management are divided between those that provide mechanisms for overload prevention, that is, provide message prioritization and load balancing [16], and those which rely on adaptation mechanisms that change the access protocol or session QoS parameters. Regarding the latter, MUM [2] proposes a dynamic and flexible middleware to support continuous services to mobile users by migrating the session state in response to user movements during service provisioning. It also integrates some sync/async client/server paradigms but it focuses in session management and preservation rather than device access or architectural issues. In [6] a Session Initiation Protocol middleware is provided for session

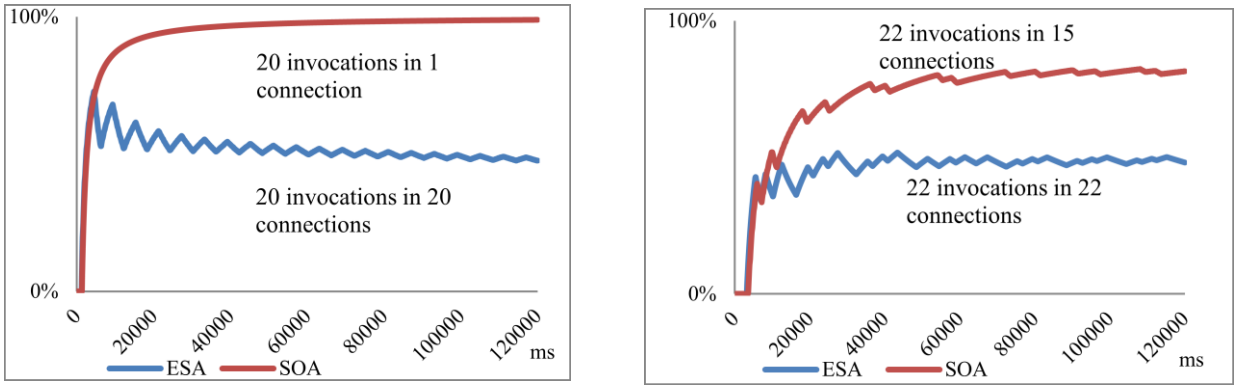

Figure 9. Average Resource Utilization for Bluetooth in Case #1 (left) and Case #2 (right).

management, which also provides resource reservation and QoS management for user services. In our work, session management is carried out by the Harmonizer subsystem [1], while physical resource reservation is performed in the access middleware.

X. CONCLUSIONS

This paper proposes a solution for homogeneous resource access with continuous service execution in ubiquitous environments. This solution is based on the decoupling between the data access and their processing and interpretation. The defined architecture is focused on the prosumer mobile environment in which the user is the centre of the environment and his mobile phone is the gateway for interacting with the surrounding capabilities. The requirements imposed by this environment determine the existence of three processes: Orchestration, Resolution and Capability Invocation. Focusing on the latter, our main contribution in this work is specified by the definition and implementation of an architecture for a communication middleware as part of an overall architecture for the mIO! project. This architecture has been developed following the design patterns Proactor, Acceptor/Connector, Monitor Object and Command [17], in order to meet the requirements of asynchronism, reusability and efficient resource management respectively.

The communication paradigms that are present in the Access Middleware (Request/Reply, QP2P and Publish/Subscribe) allow it to cope with the heterogeneity of sensors, actuators, controllers and other devices in the environment. The middleware implementation fulfils the task of decoupling capability access from the selection of the optimal capability and from the processing of the generated information.

Finally, we have made a contribution related to the management of limited resources in the mobile terminal that performs capability access by comparing the performance of two algorithms for Bluetooth access in terms of energy consumption and data access delay. This leads to the conclusion that the Access Middleware must be able to decide which algorithm to use depending on the parameter to optimize (delay or consumption), which will be given by user preferences or provided by contextual information.

REFERENCES

[1] U. Aguilera, A. Almeida, P. Orduña, D. López-de-Ipiña, and R. de las Heras, “Continuous service execution in mobile prosumer environments,” UCAmI’10, pp. 229-238, Sept. 2010.

[2] P. Bellavista, A. Corradi, and L. Foschini, "MUM: a middleware for the provisioning of continuous services to mobile users," ISCC’04, vol.1, pp. 498- 505, July 2004.

[3] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha, “Service composition for mobile environment,” Mobile Networks and Applications, 4, 10, Aug. 2005, pp. 435-451.

[4] Y. Morais and G. Elias, “Integrating Communication Paradigms in a Mobile Middleware Product Line ,” ICN’10, pp. 255-261, April 2010.

[5] T. Sivaharan, G. Blair, and G. Coulson, “GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing,” LNCS 3760, pp. 732-749, Springer 2005.

[6] T. Guenkova-Luy, H. Schmidt, A. Schorr, F.J. Hauck, and A. Kassler, "A Session-Initiation-Protocol-Based Middleware for Multi-Application Management," ICC’07, pp. 1582-1587, June 2007.

[7] S. Hadim and N. Mohamed, "Middleware for Wireless Sensor Networks: A Survey," Comsware’06, pp. 1-7, Jan. 2006.

[8] M.M. Molla and S.I. Ahamed, "A survey of middleware for sensor network and challenges," ICPP’06 , pp.-228, Aug. 2006.

[9] M. Fähndrich et al., “Language support for fast and reliable message-based communication in singularity OS,” EuroSys’06, pp. 177-190, April 2006.

[10] N.B. Harrison and P. Avgeriou, "Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation," WICSA’08, pp. 147-156, Feb. 2008.

[11] L. Cheng, Z. Wang, and X. Huang, "A stream-based communication framework for network control system," BMEI’10, vol.7, pp. 2828-2833, Oct. 2010.

[12] RGB Medical Devices. http://www.rgb-medical.com/.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.M. Kermarrec, “The many faces of publish/subscribe,” ACM Comput. Surv. 35, 2, June 2003, pp. 114-131.

[14] J.C. Cano, J.M. Cano, E. González, C. Calafate, and P. Manzoni, “Evaluation of the energetic impact of Bluetooth low-power modes for ubiquitous computing applications,” PE-WASUN’06, pp. 1-8, Oct. 2006.

[15] N. Ibrahim, "Orthogonal Classification of Middleware Technologies," UBICOMM’09, pp. 46-51, Oct. 2009.

[16] A. Erradi and P. Maheshwari, "wsBus: QoS-aware middleware for reliable Web services interactions," EEE’05, pp. 634-639, April 2005.

[17] F. Buschmann, K. Henney, D. D. Schmidt, Pattern-oriented software architecture: On patterns and pattern languages. John Wiley & Sons, 2007.