Seamless Nomadic System-Aware Servants

Walter Cazzola Dario Maggiorini
DICo - Department of Informatics and Communication
Università degli Studi di Milano
{cazzola,dario}@dico.unimi.it

Abstract

The growing diffusion of wireless technologies is leading to deployment of small-scale and location dependent information services (LDISs). Those new services call for provisioning schemes that are able to operate in a distributed environment and do not require network infrastructure. This paper describes an approach to a service-oriented middleware which enables a mobile device to be aware of the surrounding environment and to transparently exploit every LDIS discovered in the coverage area of the hosting wireless network. the paper introduces seamless nomadic system-aware (SNA) servant. SNA servants run on mobile devices, discover LDISs and are not associated with any specific service. The paper also describes the key features for the SNA servants implementation and for rendering them interoperable and cross-platform on, at least, .NET and JVM frameworks.

Keywords: Reflection, Adaptive Middleware, Service Provisioning, Service-Oriented Middleware.

1 Introduction

In the last few years there has been a considerable penetration of wireless communication technology in everyday life. This penetration has also increased the availability of *location-dependent information service*s (LDIS) [19], such as local information access (e.g. traffic reports, and news), nearest-neighbor queries (such as finding the nearest restaurant, gas station, medical facility, or ATM) and others.

New wireless environments and paradigms are continuously evolving and novel LDISs are continuously being deployed. City and tourist electronic guidebooks [7, 1] are recent examples of deployed LDISs. Such a growth means to deal with:

- services without standard interfaces - same or similar LDISs being offered by different vendors through different APIs but with the same standard functional interfaces;
- dynamically deployed services - LDIS made available on a need basis or when the scenario dynamically changes, this also calls for a dynamic roaming between services as well as for dynamic service interchangeability; and
- non-classified services (i.e., novel services).

Wireless networks are more and more often composed of stand-alone wireless nodes connected without any support infrastructure. Such networks are not administered by a central entity and client applications cannot assume any knowledge about which services are provided, where they are located and how long they will be available.

Therefore, it is necessary an infrastructure/middleware that enables a mobile device: (*i*) to automatically discover every LDIS server, (*ii*) to be aware of and to exploit the services available in its range, and (*iii*) to transparently change the LDIS server when it moves out from the current coverage area without compromising service fruition.

These issues have been faced by proposing a novel reflective and nomadic middleware especially designed for supporting dynamic adaptation of services in according with the LDISs in its coverage area. The main components of such a middleware are the *seamless nomadic system-aware (**SNA**) servants*. Their aim consists of discovering LDISs, adapting to the detected services and providing a transparent interface between the LDIS and the user.

The rest of the paper is structured as follows: section 2 presents the necessary background whereas sections 3, 4, and 5 are devoted to giving an overview of the SNA servants, of their implementation and of the benefits and drawbacks they offer. Finally, the proposed middleware has been compared with some related works in section 6, future works have been presented in section 7 and conclusions have been drawn in section 8.

2 Background

2.1 Computational Reflection.

Computational reflection (or *reflection* for short) is defined as the activity performed by an agent when doing computations about itself [20]. This activity involves two aspects: *introspection* and *intercession*. Bobrow et al. [4] define these two terms as follows:

> *Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state, or alter its own interpretation or meaning.*

Reflection applies quite naturally to the object-oriented paradigm [11]. Just as objects in the conventional object-oriented paradigm are representations of *real world* entities, objects can themselves be represented by other objects, usually referred to as *meta-objects*. Computation done by meta-objects (*meta-computation*) is for the purpose of observing and modifying the objects they represent, called *referents*.

2.2 Distributed Middleware

CORBA. The *Common Object Request Broker Architecture* (CORBA) [22] is an open distributed object computing middleware standardized by the Object Management Group (OMG). CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and unmarshaling; and operation dispatching.

CORBA provides mechanisms, such as the *dynamic invocation interface (DII)* and *stub invocation*, for requesting services to a server without knowing the server identity and location and the service interface. Basically, in the CORBA architecture there is an *implementation repository* which gathers both the interfaces of all registered servers and their services and an *object request broker* (ORB) that looks at the implementation repository for a server offering a service compliant to the client current request.

Microsoft .NET [13] (.NET for short) is a language-neutral environment for executing programs that can easily and securely interoperate. Rather than targeting a particular HW/OS combination, programs will instead target ".NET", and will run wherever .NET run-time is deployed. The .NET framework has two main parts:

- the *common language run-time* (CLR).
- a hierarchical set of class libraries

The CLR is described as the *execution engine* of .NET. It provides the environment within which programs run. The most important features are: conversion from *intermediate language* (IL) to native code (just-in-time compiling), memory management, loading and executing programs.

Java remote method invocation (RMI) [23] framework is a lightweight Java ORB-like architecture, written to provide easy access to objects existing on remote virtual machines. Once a reference to a remote object has been obtained, it can be treated as a local object. RMI performs the marshalling, transportation and garbage collection of remote objects transparent to the programmer, making it very easy to write distributed Java programs.

RMI is not a general purpose ORB with global applicability. It is a Java-only technology and is as such not a direct competitor to CORBA or .NET which are not limited to one language. Rather than having a special purpose interface definition language such as IDL, interfaces to remote objects are defined using ordinary Java interfaces, a benefit of RMI being Java-only.

3 Seamless Nomadic System-Aware Servants

Seamless nomadic system-aware (SNA) servants are *active object*s [8] running on mobile devices (e.g., PDAs, smartphones and laptops). They are mainly concerned in providing the user[1] with the services discovered in the coverage area.

SNA servants are *seamless*, *nomadic* and *system-aware* active objects. They are developed for mobile environments, this fact renders them *nomadic*. SNA servants look for the available services (any kind of service) during their wandering, when a service is discovered it is provided to the user. The servant can offer as many services as have been detected. Therefore, SNA servants provide services without being associated to any specific LDIS, that is, they are *seamless* servants.

From an architectural point of view a SNA servant is a sort of daemon running on the mobile device. During the device wandering, the SNA servant:

❶ monitors its coverage area looking for LDIS servers until a LDIS server is detected; then

❷ inspects the detected LDIS server for retrieving its services;

❸ traces such services keeping a representative of them in a *service pool*; finally

[1] Please note, that in this context with the term *user* both human and artificial actors (other SNA servants or client processes) are intended.

❹ the servant comes back to monitor its coverage area whereas the gathered services can be exploited by the user through their representative in the service pool.

Services gathered during the servant wandering are available on its behalf until they are reachable. A representative of the service[2] is managed by the servant. A representative in the service pool is updated when another service with the same description but closer or more accessible by the mobile device is found. The representative is removed from the service pool when the corresponding service becomes unreachable from the current position of the mobile device. Therefore the service pool always refers to the available best choice for each service.

Service introspection (point ❷ above) and service intercession (the second part of point ❹) well characterize the system-awareness of the SNA servants. SNA servants do not know the network infrastructure (i.e., how many servers are in their coverage area and where they are), and do not know the service nature a priori. As explained in section 5.3, they probe both the network and service structure by exploiting introspection and intercession. Introspection and intercession are provided by the underlying infrastructure and intrinsic in the servant nature.

Services in the pool are classified by using ontologies, our approach is similar to the *service classifier agent* (SCA) approach [29]. Service introspection is used to retrieve the data necessary for service classification then subsumption is used to automatically place services in taxonomies permitting a dynamic classification. This taxonomy helps in recognizing the services and in choosing the better and available service on client request.

Note that, at the moment, the SNA servants consider as a service only methods that are in the server's public interface and that can be remotely invoked, but this choice should not be considered restrictive because most of the platforms (e.g., Java, .NET and CORBA) that provide services are object-based and it is quite easy to write a method that wraps a generic, i.e., not method-based, service.

4 SNA Servant Benefits & Drawbacks

Adoption of SNA servants will lead to many benefits for service providers and implementors.

Clients' architecture and logic will be independent of type and implementation of the needed service. Requests will be dynamically adapted to the required interface and transparently forwarded to the right and more unloaded server. Therefore, the SNA servants also provide service roaming/routing and load balancing.

Moreover, services brokered by a SNA servant can dynamically change; the code needed to accommodate the change will be transparently downloaded at run-time from a trusted source with no express acknowledgement by the client (see section 5.3).

During operations, the connection status is stored in the SNA servant, thus, it migrates together with the client from a network to another and it is kept consistent with different LDISs providing the same service.

Obviously, together with these benefits we also get some drawbacks.

Many efforts need to be devoted to provide legacy to existing services, like web servers or DBMS engines, which are often needed but rarely offer an object-oriented reflection-capable interface. The lack of an object-oriented interface make legacy services difficult to be detected; port scanning on each in-range host could be needed, making the system too invasive and effecting network performances. Supporting these services will require the servant to know how to handle all possible services offered on well-known ports; thus the code could become too large for a mobile device like a PDA and the dynamic adaptation mentioned before will not be exploitable (e.g., a SNA servant trying to send mail is implicitly assuming to probe only on port 25 and to know the sendmail syntax). Furthermore, legacy services will be able to support client-initiated connections only, limiting even more SNA servants' functionalities.

Due to the nature of wireless networks, where bandwidth is a critical resource, as already mentioned, servers discovery needs to be as less invasive as possible. Application-level discovery is easy to implement and portable, but might imply unacceptable bandwidth usage as well as long timeouts to detect a server to be off-range. If the SNA servant gets integrated with the network interface driver less traffic will be generated since discovery will be possible using passive observation of MAC messages, but, on the other hand implementation will become dependent of the kernel structure, hardware platform and transmission protocol.

Having connection status on the client will expose the system to security issues regarding identification and non-repudiation for both SNA servant and LDIS. Security issues for ad-hoc networks have already been studied in literature (see [30, 14] and [28]), but application of existing security models to a SNA servants scenario is still ongoing work. Even solving authentication problems could not prove enough to secure the framework; security issues arise about sensible data located on the mobile device. The framework will be required to support applications and to avoid malicious code or the user itself to tamper with restricted information.

5 SNA Servants Technical Issues

This section goes deeper describing some technical issues that are arising during the prototype implementation of

[2] That is, a sort of stub that is used to invoke the corresponding service.

the SNA servants regarding their integration with two of the most popular object-based frameworks, that are .NET [13] and JVM [2], and with the LDIS servers running on them.

5.1 Cross-Platform Interoperability

Designing the SNA servants infrastructure, we have taken in consideration the heterogeneity of the existing LDIS servers. Many services for mobile devices are already offered, but often they are not compatible with any kind of clients or compliant with a common standard and, above all, they are developed by exploiting several different technologies, e.g., CORBA [22], Java [2], web [5] and COM/.NET [13]-based technologies.

A priori, this heterogeneity of technologies and standards hinders the nomadic aspect of the SNA servants. They wander on the net and must be able to interact with any kind of technology they encounter, both for discovering the available LDIS servers (see section 5.2) and for getting the services they offer (see section 5.3). Therefore, the SNA servants should adopt a technology that grants them the possibility of interacting with most of these technologies and executing on any system, i.e., that allow the SNA servants to achieve *cross-platform interoperability*.

Fortunately, most of the technologies adopted by the servers already take in consideration *interoperability* (Java and CORBA components interact through the Java RMI over internet inter-orb protocol RMI-IIOP [24] and the `org.omg.*` packages), or are included in novel and more flexible standards (the COM features are included in the .NET framework). Moreover Java and .NET frameworks have a very similar architecture based on just-in-time compilation and interpreted bytecodes that render them easily integrable [17].

After these considerations, rendering the SNA servants able to interact with Java- and .NET-based servers means to enable them to interact with most of the existing servers. Moreover, the Java and the .NET framework are simply integrable as proved by JNBridgePRO [17] and an hybrid Java/.NET architecture permits the SNA servants to run on any operating system (i.e., Windows, Linux and MacOS).

The interoperability among .NET and Java and thence with most of the LDIS server technologies is achieved by implementing the SNA servants with an hybrid architecture, similar to the JNBridgePro [17] architecture.

JNBridgePro is a Java-.NET interoperability tool that enables Java code to be called from .NET code, and .NET code to also be called implicitly from Java code. A system using JNBridgePro consists of components on both the Java side and the .NET side. .NET classes run on a CLR, whereas Java classes run on a JVM, and JNBridgePro transparently manages the communications between them. To expose classes from one platform to classes on the other, proxy classes are automatically created that offer access to the underlying real class.

SNA servants do not need a fully bidirectional Java-.NET interoperability as JNBridgePro offers. They simply must be implemented by using Java or .NET technology and be able to invoke remote services implemented by using the other technology. Therefore, .NET method invocations has been wrapped in Java methods which forward (by using the SOAP [6] technology) such invocations to a .NET proxy (which is a sub-component of the SNA servant architecture) that really invokes them. A similar, but specular, approach has been adopted for .NET-based SNA servants. In this way, .NET components always interact with remote .NET components and the same is valid also for Java whereas interoperability is carried out inside the SNA architecture and transparently to servers and users.

5.2 Localizing LDIS Servers

SNA servants should be independent, as most as it is possible, of the architecture and the technology adopted by the LDIS servers. Therefore, the SNA servants should be able to interact with and have a fully distributed architecture without making supposition about the service advertisement mechanism adopted by the LDIS servers. Above all, SNA servants should avoid to impose a support infrastructure to the servers that want to cooperate with them. Surely an architecture based on a brokering sub-system would have simplified the SNA servants implementation, but would also have compromised their potentiality by limiting the interaction exclusively to servers compliant with such an infrastructure.

The problem of localizing a LDIS server can be divided into two sub-problems: (*i*) localizing a host in the coverage area of the SNA servant, and (*ii*) detecting which servers are running on the localized hosts.

Localizing a host. Both structured (i.e., wireless network with a support infrastructure) and ad hoc networks (i.e., without an infrastructure) provide mobile devices with a mechanism for detecting their neighbors.

In structured wireless networks the neighborhood can be explored through the access point. Each access point internally keeps a list of hosts[3] in its zone of coverage. The involved access points update their lists when a palmtop moves from a zone of coverage to another. Therefore, the SNA servants can poll the access point looking for the IP address of the other computers in the same area.

In ad hoc networks, the neighborhood exploration takes place exploiting a MAC-level protocol. The system inter-

[3]Unfortunately, how this information is stored depends on the access point vendor, but we can suppose that there will be a standardization in the future.

cepts the signals from the handshake protocol at MAC level and monitors the protocol activities. In the Bluetooth system [3], for example, it monitors the inquiring and paging activities.

The SNA servants will adopt one of the above strategies according to the underlying network technology.

Localizing a server. The SNA servants must provide the user with services independently of the technology adopted by the LDIS server. Therefore, the SNA servants must be able to exploit the technology available on the hosts. Java, CORBA and .NET technologies provide a public repository, respectively the `rmiregistry`, the ORB and the UDDI registry [26], which supply the LDIS servers available on a certain host along with their public services and their signatures. The SNA servants will ask the repositories[4] for the registered servers when it discovers a new host in its coverage area.

In Java, the SNA servant gets the list of the servers running on a certain host by querying the `rmiregistry` of the host as follows:

```
String[] servers = Naming.list("//"+host);
Remote[] serverStubs;
for(int i=0;i<servers.length;i++)
   serverStubs[i] = Naming.lookup(servers[i]);
```

Similar investigation can be performed using .NET on a UDDI registry with the following code:

```
UddiConnection myConn = new UddiConnection(host);
FindService fs = new FindService("*");
ServiceList servList = fs.Send(myConn);
```

Whereas CORBA-based LDIS servers can be localized through the CORBA's *implementation repositories*. The implementation repositories are daemon processes that have a *server table* keeping track of the running servers. This table contains all information necessary for localizing a server (i.e., name, host and port) and the necessary stuff (e.g., a POA reference) for binding to the LDIS servers.

Section 5.1 is devoted to face the problems related to the coexistence of the previously mentioned technologies in a single entity with an hybrid architecture: the SNA servant.

Note that some nonstandard packages supporting the UDDI registry from Java are already available (e.g., the IBM UDDI4J [15]). These approaches are currently under investigation as an aid to limit the need of exploiting Java-.NET interoperability for detecting services.

[4]Please note, that the plural is used because more than a technology can be available on a certain host and the SNA servant must query each possible repository.

5.3 Service Awareness

Service awareness is the key feature of the behavior of the SNA servants. To provide the client with services, the SNA servants must be aware of the services offered by a LDIS server, that is, by using an object-oriented parlance, they must access to the *public interface* of the LDIS object. Moreover, they must be able to render available such services, that is, by exploiting the public interface, they must be able to invoke the methods provided by the server.

Reflection [20] helps in carrying out these actions. *Introspection* allows the investigation of the server looking for the service structure (that is, the signature of the corresponding method), whereas *intercession* allows requesting and obtaining the service (that is, to invoke the corresponding method and to get the result). Both .NET and Java are intrinsically reflective in nature therefore they provide the programmer with the basic mechanism for LDIS server introspection and intercession.

Basically when a SNA servant meets a Java server, the servant gets the list of the services (i.e., methods) provided by the server (i.e., an object) by inspecting its class:

```
Class serverClass = serverStub[i].getClass();
Method[] serviceList = serverClass.getMethods();
```

Whereas, the SNA servant provides the service by forwarding the request to the service representative obtained during server inspection:

```
serviceList[j].invoke(serverStub[i], arguments);
```

where `serverStub[i]` is the representative of a LDIS server (see section 5.2).

Similar steps can be carried out in the .NET framework by exploiting the classes in the `System.Reflection` namespace. The *common language run-time* (CLR) allows the SNA servant of investigating the *metadata* of the server *package*.

```
string serviceURL =
   GetServiceURL[5](servList[i].ServiceInfos);
Type serverClass = GetTypeFromWSDL[5](serviceURL);
object server =
   Activator.GetObject(serverClass, serviceURL);
MethodInfo[] ServiceList =
   serverClass.GetMethods();
ServiceList[j].Invoke(server, arguments);
```

[5]Note that `GetServiceURL` and `GetTypeFromWSDL` are dummy functions that are used for hiding some irrelevant details omitted for sake of clarity.

Thence, the SNA servant can invoke the service through its metadata.

CORBA does not provide real reflective facilities for inspecting LDISs exploiting its technology but service awareness can be achieved as well querying the implementation repository [12]. The implementation repository has been designed for providing a client with an implementation to the service they are requiring. It can also be used, by looking at the server table, for becoming aware of the services provided by a host.

5.4 Service Pool

As explained in section 5.3, the SNA servant provides the client with some services by exploiting intercession on representatives of such services. Therefore, each servant can provide as many services as many service representatives have been collected and are directly available to the servant.

The *service pool* is the data structure put in charge of storing service representatives. The heterogeneity of architectures, that can interact with the SNA servants, has raised two issues directly related to service pool content and the interoperability property:

(*i*) what is a representative and how to store/retrieve it;

(*ii*) how a representative running on the Java virtual machine (JVM) can coexist with one running on the .NET framework.

Stubs and *Proxies* are respectively representatives of remote servers in Java RMI [23] and .NET frameworks. Through stubs and proxies, it is possible to retrieve the list of the provided services and to build a representative of these services[6] that will be used by the SNA servant for invoking the service. Whereas, CORBA-based services do not require a representative of the services on the client-side. They simply require an object able to ask the server for the service by using the provided communication protocol, the *internet inter-orb protocol* (IIOP). Java RMI can communicate with CORBA-based servers by exploiting RMI-IIOP [24]. Hence, CORBA-based LDIS servers have been assimilated to Java objects and the service pool store a dummy stub able to connect to the CORBA-based service.

Of course, .NET and Java objects are not compatible and it is necessary a special infrastructure for overcoming this problem and transparently invoking a service independently of the technology adopted by the server. Therefore, the dual nature of the servant architecture has repercussions on the service pool.

[6] In Java and .NET, service representatives are respectively instances of `Method` and `MethodInfo` classes.

In the hybrid architecture of the SNA servant (see section 5.1) the responsibility for a task is given to the component that does the job best. The JVM is best at running Java bytecode, and the CLR is best at running .NET IL. Therefore, the invocation of .NET-based service is demanded to the SNA servant .NET proxy whereas the invocation of Java-based code is directly invoked by the JVM (cf. section 5.1).

The service pool is a data structure that masks this hybrid nature providing the servant with an uniform interface to the gathered services. Each service request directed to the pool is automatically forwarded to the component of the SNA servant able to invoke it.

Moreover, the service pool is used to handle service classification, service automatic roaming/routing and service reliability/availability. In the pool there will be a representative for any kind of service. Such representatives will refer to the closest or more reliable implementation of the corresponding services. The service pool is indexed on a service description similar to the WSDL [9]. Note that .NET services already have a WSDL description, whereas Java services are consistently described by the Sun WSDL extension. The SNA servants use this description both for filling up the service pool with services and for skimming the service pool off duplicated services, i.e., services with a similar description. Service pool indexing is based on services ontologies as described in [29], these ontologies also help in determining when two services are similar. The service pool is purged of services no longer active or replaced by a better service when the SNA servant coverage area changed. In this way, the growth of the service pool is limited.

Figure 1 shows some details of the inner and modular architecture of a SNA servant and its interaction with servers in the network. Each step — that is (1) server discovery, (2) service discovery, (3) service classification and finally (4) service brokering to the final user — is entrusted to modules. A modular architecture provides the necessary flexibility to extend the approach to technologies do not considered yet. Moreover, these modules, except for the last one whose execution is asynchronous and depends on the user requirements, form a sort of pipeline that allow the use of a multithreaded architecture for speeding up the SNA servant performances.

6 Related Works

6.1 Service Discovery

Many projects are focused on centralized approaches that assume the presence of a node keeping a service repository. LDIS servers register their services to such a repository and clients ask it for a service. Proposed architectures can be classified into three branches: (*i*) *centralized query*,

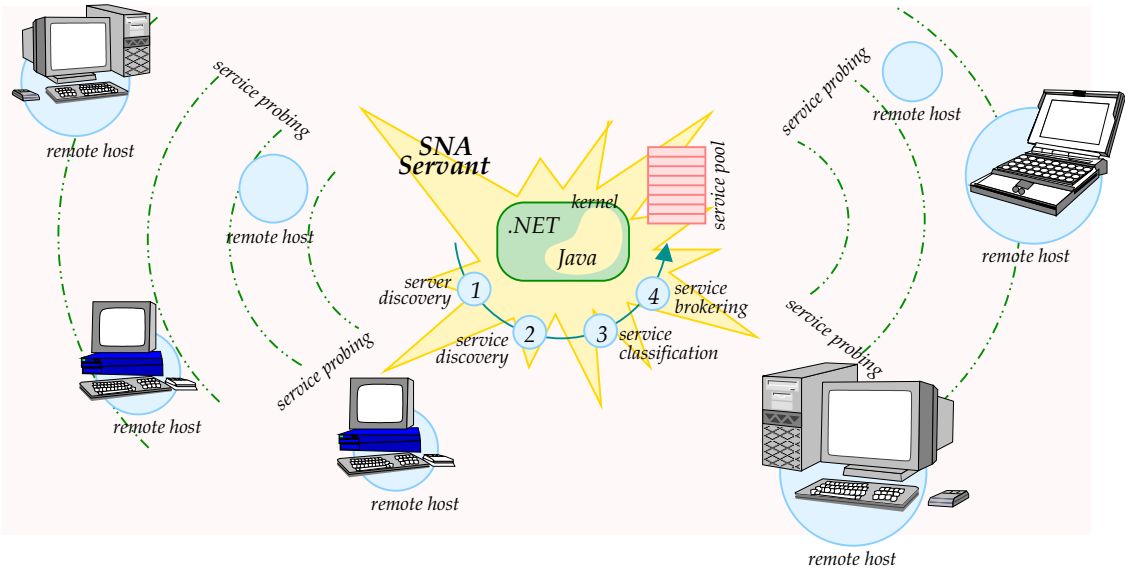

Figure 1. The architecture of the SNA servants and their interaction with the environment.

(*ii*) *local disconnected query* and (*iii*) *local interconnected query*.

The *web services* [5] architecture is an example of a centralized query architecture. It uses a UDDI server [26] to broker services. Each query for available services must be directed to the UDDI server that needs to be known to everyone and could become a bottleneck.

In the *local disconnected query* architecture, the entire network is divided in subnetworks and a *local coordinator* is responsible for coordinating the queries and service advertisement of its subnetwork. A mobile device must ask for the local coordinator when it passes from a sub-network to another. This kind of architecture is called "disconnected" because there is no information sharing among coordinators. The local disconnected query architecture is exploited in [27] and [16].

In the *local interconnected query* architecture, there still are local coordinators managing queries and service advertisement, but these coordinators also share information. This is the case of the *salutation architecture* [25] that employs brokers named *salutation managers*. Each query is directed to a local salutation manager that provides the answer or routes the request to another manager.

All the above mentioned architectures are poorly scalable because the repositories could become a bottleneck when the network grows larger. Moreover, each repository uses legacy protocols for communicating with servers and clients, therefore both clients and servers must be compliant with the infrastructure adopted by the repository in order to exploit its services. The *legacy* aspect of these architectures often hinders service fruition from foreign clients.

The SNA servants are independent of any specific architectures, cross-platform and able to interact with the most diffuse repositories. Thence, the mobile device can enter in a foreign network and SNA servants are able to exploit the services of that network without imposing an infrastructure layer and without giving up exploiting some services.

6.2 Distributed Middleware

Java RMI [23], CORBA [22] and .NET [13] can be considered distributed middleware, that is, a set of services that resides between the application and the operating system and aims to facilitate the development, deployment, and management of distributed applications [10]. CORBA and .NET are mainly service-oriented middleware whereas Java RMI is a server-oriented middleware where the distinction is related to their approach to service provisioning. Server-oriented middleware will provide service through a named server, that is, the client asks for a service to a specific remote server and the middleware will link the client to exactly that server, whereas service-oriented middleware links the client to the service without worrying about who is serving such a request.

The SNA servant infrastructure can also be considered a distributed middleware which provides mobile devices with a mechanism for retrieving services available in the surroundings. The approach is obviously service-oriented because the service pool hides to the user the nature, the location and the identity of the server providing the service. The user will simply retrieve a service whose description satisfy its request.

Both .NET and CORBA supply a mechanism (respec-

tively WSDL description and the interface-based approach) for describing the services, but similar services, i.e., services that differ in the description or in the applicability rules are treated as different services, therefore their approach is redundant and also serving requests in a balanced way is difficult. SNA servants use ontologies to classify the discovered services for similarity, similar services are treated as the same service.

Besides, .NET, CORBA and Java RMI foresee the use of their own infrastructure to be able to provide the servers and the clients with their services. Therefore, both servers and clients must be compliant to such infrastructure in order to exploit the middleware services, e.g., in CORBA client must direct their query to the ORB. The SNA servants, because of reflection, are independent of every underlying architecture, can exploit the infrastructure provided by the other middleware and transparently provide the user/client with the services without imposing the use of a middleware-dependent API.

The major difference between SNA servants approach and the other middleware-based approaches is represented by the fact that a SNA servant is a sort of architecture-independent broker that is able to adapt itself to the architecture of the host network and to directly query a server for the interface of its services and then to use such interfaces for getting the services. Therefore, the SNA servants neither need their own service repository nor use legacy protocols for achieving the services of the host network, but adapt themselves to the technology exploited by the LDIS servers.

7 Future Works

The SNA servants approach has several possibilities and many of them are still to be explored. In the future we are planning to enhance the interoperability of the approach and the mechanism for service classification in order to compose them.

7.1 Global Interoperability

At the moment, SNA servants do not take in consideration services provided by servers that are not CORBA, .NET or Java compliant such as TCP/IP based servers (e.g. a "classic" web server). Therefore, the SNA servants do not achieve a global interoperability with all kinds of servers yet. In the future, this limitation will be overcome providing such kind of servers with a wrapper, called *reflective proxy agent*, that enables the interaction of the wrapped server with our SNA servants.

In this way, the SNA servants should be able to discover and inspect these reflective proxy agents as well as the LDIS servers CORBA, .NET or Java compliant whereas the reflective proxy agents will provide a transparent reflection-based interfaces to standard services without modifying the structure of the corresponding servers.

To be compliant with the SNA servants infrastructure the reflective proxy agents should wrap the server autonomously, that is, they should be able to detect these servers (e.g., by sniffing the network traffic) and adapting their structure to the communication protocol of the detected server. For example, if the detected server is a web server the reflective proxy agent must translate each servant request to an HTTP request.

We strongly believe usage of reflection-proxies disseminated on the network can be a good compromise to grant interoperability with all kinds of servers without imposing a change to their structure in order to support the approach.

7.2 Service Composition

A completely new feature that should be interesting to investigate, and would be a natural extension of the SNA servant is the ability to create a new service as the composition of two or more services available in the coverage area of (and therefore already gathered by) the SNA servant. An example of service composition could be a navigation system service that may get composed by a `compass` and a `map` service discovered in the coverage area.

The advantages of service composition are easily understandable: a larger number of services will be available in a coverage area while a fewer number of LDISs are necessary to provide them in the same area; obtaining also a reduction of network congestion. Moreover, service composition offers a simpler way for service customization.

To compose services, as well as to compose object functionalities, is not a simple task [21] but we trust that by exploiting .NET generics [18] and by adding some constraints on service composability we should get a good compromise among composition intractability and powerfulness. The basic idea consists of categorizing the gathered services in generic classes and of planning the composition in advance on these generics. When a SNA servant discovers a service the corresponding generic class can be instantiated providing a base for the construction of a novel service by exploiting the planned service composition.

In the navigation system of the example above, the generic map service could be described by a generic class parameterized on (among other things) the type of the coordinate system adopted by the map. Therefore when a real map service is discovered this parameter is instantiated following the characterization (Cartesian or polar system) of the just discovered service. Similar steps are taken when a compass service is discovered. Then a pre-defined composition rule can be applied to get the expected navigation

service. Moreover, this approach allows the novel service to adopt the best implementation for the required component services choosing among the services that the SNA servant has available in the service pool.

8 Conclusions

This paper discussed about wireless service provisioning in ad-hoc networks. We presented a service-oriented middleware for location dependent service provisioning based on reflection. This middleware exploits SNA servants as local service providers on a mobile device to provide the user with surrounding LDISs as they are discovered. By means of SNA servants we grant architecture-independency and service awareness to the final client application. Many issues about SNA servants have been discussed and we argued about their adoption will lead to (*i*) cross-platform interoperability, (*ii*) easier service discovery and (*iii*) services composition. Even so, many issues still exists and are the subject of ongoing work. During the paper we addressed security, which should be studied at the infrastructure level, and legacy to existing TCP-based services like web servers and DBMS. We strongly believe that the adoption of the SNA servants architecture will lead to a better service exploitation and deployment on next generation wireless networks.

Acknowledgements

This work has been partially supported by Italian MIUR (FIRB "Web-Minds" project N. RBNE01WEJT_005).

References

[1] M. Ancona, W. Cazzola, and D. D'Agostino. Smart Data Caching in Archeological Wireless Applications: the PAST Solution. In A. Clematis, editor, *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euromicro PDP 2003)*, pages 532–536, Genova, Italy, on 5th-7nd of Feb. 2003. IEEE Computer Society Press. ISBN: 0-7695-1875-3.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, Dec. 1997.

[3] Bluetooth SIG. Specification of the Bluetooth System Version 1.1. *Core, Specification Volume 1*, Feb. 2001. Available at `http://www.bluetooth.org`.

[4] D. G. Bobrow, R. G. Gabriel, and J. L. White. CLOS in Context - The Shape of the Design Space. In A. Pæpcke, editor, *Object Oriented Programming: The CLOS Perspective*, pages 29–61. MIT Press, 1993.

[5] D. Booth, M. Champion, C. Ferris, F. McCabe, E. Newcomer, and D. Orchard. Web Services Architecture. Technical Report, May 2003. Available at `http://www.w3.org/TR/2003/WD-ws-arch-20030514/`.

[6] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Frystyk Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Recommendation available at `http://www.w3.org/TR/SOAP`, May 2000.

[7] K. Cheverst, N. Davies, K. Mitchell, and A. Friday. Experiences of Developing and Deploying a Context-Aware Tourist Guide: The Lancaster Guide Project. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (Mobicom 2000)*, pages 20–31, New York, USA, 2000. ACM Press.

[8] R. S. Chin and S. T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, Mar. 1991.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Recommendation available at `http://www.w3.org/TR/wsdl`, Mar. 2001.

[10] G. Coulson. What is Reflective Middleware? In IEEE Distributed Systems On-Line, 2000. `http://boole.computer.org/dsonline/middleware/RM.htm`.

[11] J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, Oct. 1989.

[12] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, Massachusetts, 1999.

[13] K. Hoffman, J. Gabriel, D. Gosnell, J. Hasan, C. Holm, E. Musters, J. Narkiewickz, J. Schenken, T. Thangarathinam, S. Wylie, and J. Ortiz. *Professional .NET Framework*. Wrox Press., 2001.

[14] J.-P. Hubaux, L. Buttyán, and S. Čapkun. The Quest for Security in Mobile Ad Hoc Networks. In *Proceeding of the 2nd ACM Symposium on Mobile Ad Hoc Networking and Computing*, Long Beach, CA, Oct. 2001.

[15] IBM and HP. UDDI4J Version 2.0, Jan. 2003. Available at `http://www-124.ibm.com/developerworks/oss/uddi4j/`.

[16] J. Jawanda. Mobile Service Discovery over Wireless Links. Internet Draft, Nov. 1998.

[17] JNBridge LLC. Connecting Java with the .NET Common Language Interface: The JNBridgePro Solution. Technical report, 2002.

[18] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI01)*, pages 1–12, Snowbird, Utah, USA, June 2001.

[19] D. L. Lee, W.-C. Lee, J. Xu, and B. Zheng. Data Management in Location-Dependent Information Services. *IEEE Pervasive Computing*, 1(3):65–72, 2002.

[20] P. Maes. Concepts and Experiments in Computational Reflection. In N. K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, Oct. 1987. ACM.

[21] P. Mulet, J. Malenfant, and P. Cointe. Towards a Methodology for Explicit Composition of MetaObjects. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notice*, pages 316–330, Austin, Texas, USA, Oct. 1995. ACM.

[22] Object Management Group. Common Object Request Broker Architecture: Core Specification. Technical Report 2002.12.06 Revision V.3.0.2, OMG, Dec. 2002.

[23] SUN Microsystems. Java™ Remote Method Invocation - Distributed Computing for Java. White paper, SUN Microsystems, 1998. Internet Publication - `http://www.sun.com`.

[24] SUN Microsystems. Java™ RMI over IIOP. White paper, SUN Microsystems, 1999. Internet Publication - `http://java.sun.com`.

[25] The Salutation Consortium. Salutation Architecture Specification Version 2.1, 1999. Available at `http://www.salutation.org`.

[26] UDDI.org. UDDI: Universal Description, Discovery and Integration. White paper, Sept. 2000. Available at `http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf`.

[27] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service Location Protocol. IETF RFC 2165, June 1997.

[28] A. Weimerskirch and G. Thonet. A Distributed Light-Weight Authentication Model for Ad-hoc Networks. In K. Kim, editor, *Proceedings of the 4th International Conference on Information Security and Cryptology (ICISC 2001)*, LNCS 2288, pages 341–354, Seoul, Korea, Dec. 2001. Springer.

[29] P. Weinstein and W. P. Birmingham. Runtime Classification of Agent Services. In *Proceedings of the AAAI-97 Spring Symposium on Ontological Engineering*, Stanford, Palo Alto, CA, USA, Mar. 1997.

[30] L. Zhou and Z. J. Haas. Securing Ad Hoc Networks. *IEEE Network*, 13(6):24–30, Nov. 1999.