Journal of Web Engineering, Vol. 1, No. 2 (2003) 128–146
© Rinton Press

AN ENHANCED SERVICE ORIENTED ARCHITECTURE FOR DEVELOPING WEB-BASED APPLICATIONS

DOMENICO COTRONEO, CRISTIANO di FLORA[a] and STEFANO RUSSO
Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli "Federico II"
Via Claudio 21, 80125, Naples, Italy
(cotroneo,diflora,sterusso)@unina.it

Received October 25, 2002
Revised March 21, 2003

Web services architectures have recently emerged as a standard, service oriented approach for developing Internet-scale distributed systems. Such architectures are characterized by discovery and delivery infrastructures, since service provisioning follows the publish-find-bind paradigm. Recently, a variety of service oriented architectures have been proposed, where service discovery infrastructure does not take into account non-functional requirements associated to a service. Furthermore, service repositories are merely conceived as service descriptors containers, without any relationship with the actual availability of services. This paper proposes an enhanced service oriented architecture, called PRINCEPS, particularly suited for developing web-based applications. PRINCEPS resorts to a novel service discovery protocol which assembles services at run-time according to both functional and non-functional client requirements. The protocol is XML-based and it exploits a lease mechanism to maintain service repositories consistent with actual running service instances. PRINCEPS is endued with a service delivery infrastructure, which is based on the HTTP protocol, and allows extended client-server models to be implemented. PRINCEPS is interoperable with web-service technologies standardized by the world wide web consortium. A complete example, which shows the advantages of PRINCEPS architecture, is also provided.

Keywords: Service Oriented Architectures, Web Services, Service Discovery, Service Delivery, Jini
Communicated by: D Schwabe & J Whitehead

1 Introduction

Recent advances in network technology and computing systems are leading to a new model for distributed system, which stems from the integration of Commercial Off The Shelf (COTS), legacy, and ad-hoc components. Such a model provides the user with services that are accessible via a web-based infrastructure regardless of client physical location and/or devices. Heterogeneity and diversity are thus the trickiest issues to be addressed when developing modern distributed systems. Although middleware technologies simplify heterogeneity problems, they do not completely solve them. As a matter of fact, abstraction layers introduced by mid-

[a]Contact author's contact info: Dipartimento di Informatica e Sistemistica, Via Claudio 21 – 80125 Napoli, Italy; tel: +39-0817683869; fax: +39-0817683816.

dleware to address integration issues may also cause interoperability problems when dealing with different middleware platforms [1]. This is especially true when considering Internet-scale software systems. Heterogeneity and diversity of such systems are nowadays addressed by adopting a component based approach, which focuses on assembling systems from distributed services. More specifically, new standards, supported by many commercial products such as IBM's WebSphere and Sun One Application Server, are emerging, namely Microsoft .NET platform [2] and Java 2 Enterprise Edition [3]. These platforms are quite suitable for small-scale enterprise applications, for they are based on the concept of creating services through configuration rather than through programming. Component-based approaches allow to obtain lower maintenance costs, since they focus on separation between development and deployment issues. However, the main weakness lies in application configurability, which has become nearly as complicated as programming: suffice it to think of the deployment descriptor design of Enterprise Java Beans (EJB) [3]. In this scenario, the concept of service is becoming crucial, since a distributed system consists of a collection of interacting services, each providing access to a well-defined set of functionalities. The resulting system is defined as *service oriented architecture*(SOA), and its evolution is guided by the addition of new services. As defined in [4] "*a service is implemented as course-grained, discoverable software entity that exists as single instance and interacts with applications and other services*". SOAs federate such services into a single, distributed system capable of spontaneously configuring itself upon service connections and disconnections. SOAs can be implemented according to the following paradigms: *i*) service implemented on a single machine, *ii*) service distributed on a local area network, and *iii*) service more widely distributed across several company networks.

A particularly interesting case is when services use the Internet as communication infrastructure, i.e., web-services architecture [5]. SOAs is not a new notion. The first definition was given in 2000 [6], and it became important because of emerging web service technology. Service oriented applications are developed as independent sets of interacting services offering well-defined interfaces to their potential users [6]. A SOA supports applications to browse and discover collections of services, select those of interest, and assemble them to create the desired functionality, i.e., the one which satisfies client requirements. In such architectures the service discovery infrastructure does not take into account non-functional requirements associated to a service; furthermore, service repositories are merely conceived as service descriptors containers, without any relationship with the actual availability of services. Hence, in order to develop Internet-scale applications, more flexible solutions are required.

This work proposes an enhanced service oriented architecture, called PRINCEPS (*Pluggable Reliable Infrastructure for Network Computing and Enhanced Properties of Services*), particularly suited for developing web-service applications. PRINCEPS resorts to a novel service discovery protocol (SDP) which assembles services at run-time according to both functional and non-functional client requirements. The protocol is XML-based and it exploits a lease mechanism to keep descriptors repositories consistent with actual running service instances. This mechanism allows each service that becomes available in the system to renew its lease periodically. Services that are not effectively available loose their leases (e.g., the lease expires). The implemented mechanism potentially increases service availability by registering a redundant set of server clones which have the same interface.

PRINCEPS provides also a service delivery infrastructure (SDI), which uses HTTP as transport protocol, and allows to implement extended client-server models. Indeed, clients' resource limitations may require certain operations to be normally performed on PRINCEPS servers. PRINCEPS supports different client-server computing paradigms, from thin client (i.e., all the logic resides on the server) to fat client (i.e., all the logic resides on the client).

The rest of the paper is organized as follows. Section 2 introduces technologies exploited for the implementation of PRINCEPS; it also discusses standards concerning service oriented and web-service architectures used throughout the paper. Section 3 details the complete architecture, highlighting the SDP and the SDI. Section 4 describes PRINCEPS as web-service infrastructure. In particular, it deals with the design and implementation of components, which make PRINCEPS interoperable with the standard universal description discovery and integration protocol (UDDI), and simple object access protocol (SOAP). Section 5 describes a complete example, using a service oriented multimedia application as a case study. Section 6 gives an overview of related research in the area, while Section 7 concludes the paper.

2 Background

2.1 Service oriented architectures and web services

Service oriented architectures are spreading out within the Internet as web service architectures (WSA). As stated in [5], both basic and extended WSAs are based on SOA. A SOA is characterized by several entities: *i*) **service**, i.e., the logical entity, defined by one or more published interfaces; *ii*) **service provider**, i.e, the entity that implements a service specification; *iii*) **service requestor**, i.e., the software entity that requests a service to a specific provider (it can be an end-user application or another service); *iv*) **service locator**, i.e., a service provider that acts as registry and allows the lookup of service provider interfaces and service locations; and *v*) **service broker**, i.e., a service provider that forwards service request to one or more additional service providers. Description of services, the context of their use, and the strong heterogeneity of the environment impose a series of constraints upon development of SOAs. These are briefly summarized in the following [4]:

- **coarse-grained** - operations on services are frequently implemented to encompass more functionalities and operate on large data sets;
- **interface-based design** - implementation and interface are completely separated: multiple services can implement a common interface and a service can implement multiple interfaces;
- **discoverable** - services may be discovered at run time, by providing a unique service identifier, or by providing service characteristics;
- **single instance** - each service is executed as a single instance;
- **loosely coupled** - services connect to other entities, i.e., clients and services, using standard and decoupled message-based methods, such as XML document exchanges.

It is worth noting that these aspects differentiate a service oriented application from an application developed by using component-based middleware platforms such as J2EE or .NET.

A WSA is a particular class of SOA where services are uniquely identified by a uniform resource identifier (URI), whose interfaces and bindings may be defined, described, and discovered by XML artifacts.

The W3C and UDDI community have drafted vendor neutral open standards of the core web service protocols, such as simple object access protocol (SOAP) [7], web service description language (WSDL) [8], and universal description discovery and integration (UDDI) [9]. Nevertheless, as stated by the world wide web consortium (W3C) [5], "*the term Web Service does not presuppose the use of SOAP as a packaging format or a processing model. Nor does it presuppose the use of WSDL as service description language*". Some novel discovery protocols are emerging as alternatives to UDDI, such as the file-based web services inspection language (WSIL) [10]. WSIL represents a promising step towards the extensibility of existing mechanisms for managing web services descriptions. As stated in [10], WSIL documents allow their consumers to select and retrieve services from the available descriptions and to access only those they are able to understand. The main difference between WSDL and WSIL is that WSDL is a language for description format and does not specify a retrieval mechanism, whereas WSIL does. WSIL may be considered as an extension of the standard UDDI and WSDL technologies, since it provides bindings for WSDL descriptions and for their retrieval from UDDI registries.

2.2 Jini and Javaspaces

This sub-section briefly describes Jini technology [11, 12], highlighting the features exploited to build the PRINCEPS prototype.

Jini technology consists of an infrastructure and a programming model which address the fundamental issue of how clients connect to each other to form an impromptu community. Jini provides a service discovery layer, which allows services to be discovered at run-time. Such a layer is also able to group services in federations.

Jini services are characterized by a set of attributes, describing interface, properties and additional information about a specific implementation.

The core of a Jini system is the *lookup service* (LS), which allows *i*) clients to find and use services, and *ii*) servers to join the Jini system. Three specific protocols define interactions with the LS, which are described in the sequel.

The Jini SDP is used both by clients and by servers to locate available LSs. This protocol is composed of: *i*) a *multicast discovery protocol*, used to discover one or more LSs on a local area network (LAN); *ii*) a *unicast discovery protocol*, used to establish communication with a specific LS over a wide-area network (WAN); and *iii*) an *announcement protocol*, used by a LS to announce its presence to other LSs.

The *join protocol* enables services to be added to a Jini federation: any server willing to join the system provides the LS with the *service object* — which is the client-side component of the provided service — along with service attributes. The result of a successful registration is a *service registration* object. The *join protocol* forces service providers to lease service registrations and periodically renew the interest in joining the federation; if the lease expires, the LS removes the service.

The *lookup protocol* is used for locating services in one or more federations. It defines interactions between client and LS. Clients locate services according to their attributes and

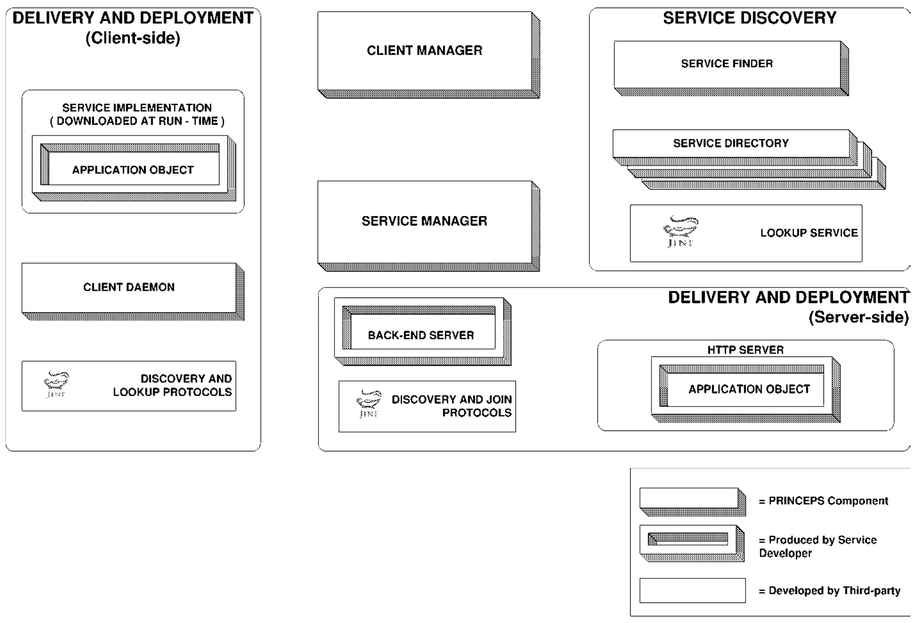
DELIVERY AND DEPLOYMENT (Client-side)
SERVICE IMPLEMENTATION (DOWNLOADED AT RUN - TIME)
APPLICATION OBJECT
CLIENT DAEMON
DISCOVERY AND LOOKUP PROTOCOLS
CLIENT MANAGER
SERVICE MANAGER
SERVICE DISCOVERY
SERVICE FINDER
SERVICE DIRECTORY
LOOKUP SERVICE
DELIVERY AND DEPLOYMENT (Server-side)
BACK-END SERVER
DISCOVERY AND JOIN PROTOCOLS
HTTP SERVER
APPLICATION OBJECT
= PRINCEPS Component
= Produced by Service Developer
= Developed by Third-party

Fig. 1. Overall architecture of the PRINCEPS infrastructure

interfaces, which are encapsulated into a single *service template.* Once a service has been located, the *service object* is dynamically deployed on client devices. Afterwards, clients interact with the *service object* by means of its interface, thus hiding service implementation details. The interested reader may refer to [12] for further details on Jini technology.

JavaSpaces technology provides a mechanism for distributed objects persistence and exchange, which is based on the tuple space model [13]. JavaSpaces are designed to deal with the implementation of distributed algorithms using the Java programming language; this functionality is provided through the abstraction of a shared container of distributed Java objects; JavaSpaces programming model consists of a few basic operations for writing, reading and extracting objects from the space. The adopted implementation of the JavaSpaces service is based on the Jini technology and it is shipped with the standard Jini distribution kit.

3 PRINCEPS architecture

3.1 Overall architecture

PRINCEPS is a SOA particularly suited for implementing web-service applications. A preliminary work on such an architecture was presented in [14], where focus was mainly on implementation of extended client-server models.

Figure 1 depicts the conceptual model of the PRINCEPS architecture along with the adopted technologies. The overall architecture is composed of the SDP and the SDI subsystems, which are partitioned between the client-side and the server-side. The *client manager* and the *service manager* components act as entry points for service requestors and service providers. Services and clients are represented by XML descriptors which are used by the

SDP for adapting services to client characteristics.

The client-side delivery infrastructure consists of the following components:

- the *application object* (AO) represents the application on the client side, for it contains all the business logic which has not been deployed on the server-side; it is downloaded at run-time;
- the *client daemon (CD)* is in charge of downloading and executing the AO.

On the server-side, the delivery and deployment infrastructure is mainly constituted by the *back-end server* (BES) and by one or more HTTP Server. The former is in charge of *i)* implementing server-side application logic and *ii)* registering the service with the discovery infrastructure. The latter represents the AO repository and enables clients to download AOs.

The SDP sub-system consists of the following components:

- the *service finder (SF)* is responsible for *i)* choosing services which best fit client characteristics, and *ii)* assembling services at run-time in order to provide more complex ones;
- the *service directory (SD)* manages the service descriptors repository, and updates service list and service descriptors upon dynamic connections / disconnections of services, and upon dynamic changes of service attributes.

As already mentioned, clients and service providers interact with the PRINCEPS architecture by means of the following components:

- *client manager (CM)* provides client registration and service discovery functionalities;
- *service manager (SM)* manages service registrations; service providers can register their services interacting with such a component.

3.2 Service discovery

Several issues are addressed by the PRINCEPS SDP sub-system on both the client and the server side. As far as the client is concerned, the CM provides service requestors with the abstraction of a *dynamic* and *adaptive* service list. Such a list is *dynamic* in that it is automatically updated upon service connections / disconnections, and it is *adaptive* since it is tailored to client characteristics. In order to adapt services to client requirements, service discovery is performed by analyzing both client and service descriptors. The client descriptor is an XML file containing client characteristics, in terms of computing power, network connection, and memory capacity. The service descriptor contains service functional characteristics, such as the kind of service (e.g., video on demand, e-commerce), the service interface, and the service requirements, in terms of needed bandwidth, connection reliability (this information is useful for wireless connections), and needed computing power. A list of quality attributes (i.e., reliability, availability, and security) is associated with each PRINCEPS service. Hence, the same service (i.e., the same service instance) may be associated with more than one implementation, each one satisfying several non-functional requirements. It is worth noting that a service may also represent the aggregation of more than one service. In particular, a PRINCEPS

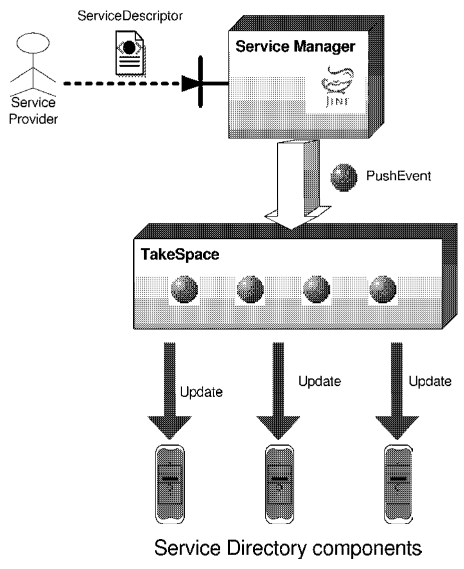

Fig. 2. Update of service descriptors

service may be *i*) a new service that has been developed from scratch, or *ii*) an existing service which has been integrated into the PRINCEPS infrastructure, or *iii*) a composed service which stems from the aggregation of functionalities provided by other services.

As pertains to the server, PRINCEPS adopts a lease-based discovery model in order to provide a flexible mechanism for managing service registrations. Each service provider has to renew its lease periodically. During the lease period, a provider can also update service descriptors by providing new implementations. In order to keep service directories consistent with actual running service instances, an event-driven mechanism exploiting JavaSpaces technology [13] has been implemented. Each provider registering a new service obtains a registration lease. During the renewal phase, the service provider may perform one of the following operations: *i*) a simple renewal, i.e., the service has not been changed and it is still available, or *ii*) creation of a new service descriptor, which is delivered to the SM component (i.e., the service has been changed). Subsequently, the SM component produces an event, which contains the new service descriptor and is persistently collected by the TakeSpace component; this is a sort of distributed shared memory, implemented by means of JavaSpaces technology, as illustrated in Figure 2. The TakeSpace delivers events to all registered SD components, i.e., such components act as event consumers for the TakeSpace.

If the provider does not renew its lease, the SM component delivers a *remove* event to TakeSpace. Subsequently, this forwards the event to SDs which erase the descriptor from their local repositories, thus marking the service as not available. The proposed mechanism enables providers to improve service availability. Indeed, a problem may arise if a service crashes and a client tries to use it before the expiration of the lease, since the service is not available. In this case, the PRINCEPS infrastructure will detect such a problem only upon the expiration of the registration lease. To deal with this problem, PRINCEPS allows service

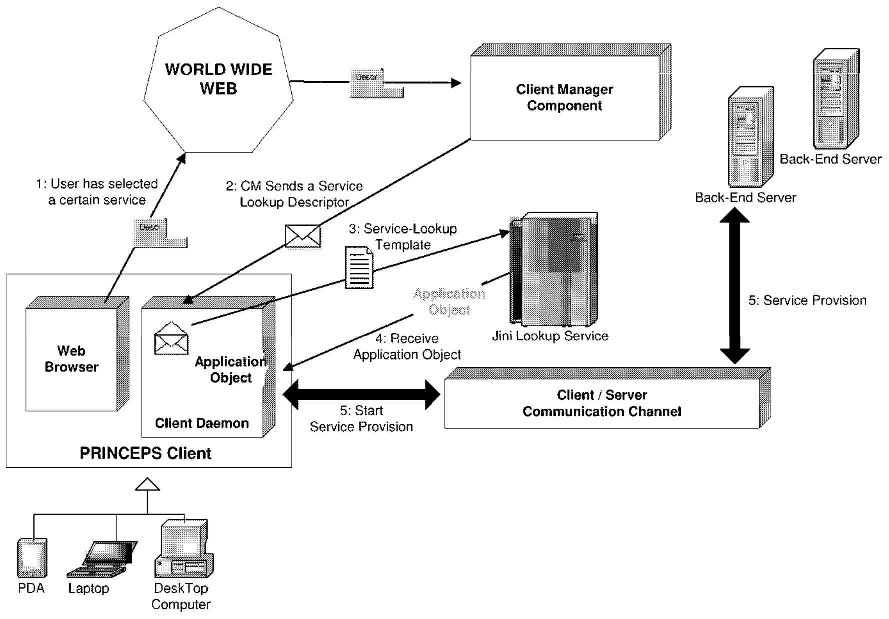

Fig. 3. Application object retrieval

availability to be enhanced by registering a redundant set of server clones which have the same interface and the same lookup attributes. Hence, all the server clones share the same AO, which can be retrieved by using the same service descriptor. As a matter of fact, when the CD receives a service descriptor object and looks up the service on a LS, it retrieves a redundant set of AOs. If the selected AO fails, an exception is thrown and the daemon tries to use a new AO.

3.3 Service delivery

The AO has been implemented as Jini service object. As depicted in Figure 3, when the user chooses a specific service, the *CM* component sends a *Jini service descriptor* (JSD) object to the CD; such an object is composed of a *lookup locator* (LL) and a *service template* (ST) object. The former is used to locate active Jini LSs, whereas the latter contains attributes that will be used to download the AO through Jini's *lookup protocol.* It is worth mentioning that the combination of a web-based selection mechanism with a Jini-based AO provision strategy allows the infrastructure to deal with nomadic users, giving them a means for browsing and using services regardless of service and client location. The abstractions of the AO and BES enhance the capability of the service-delivery infrastructure for a variety of reasons.

First, these abstractions allow the infrastructure to provide services by adopting the client-server paradigm that suites client characteristics. As a matter of fact, service developers may design different AOs depending on target devices, and enabling the infrastructure to adapt services to client characteristics and service requirements. Indeed, if a palmtop / personal digital assistant (PDA) computer is used, the AO should be light, while all the service logic should be executed on the server-side (thin client model). On the contrary, if the client device is a powerful laptop, the AO should be heavy, and the server logic could be divided between the client-side and the server-side (fat client model). For the sake of simplicity, let us consider an *mp3* stream player. The decoding process of an *mp3* media file is heavy compared to computing capabilities of some portable devices (i.e., some PDAs or mobile phones). Therefore, for such devices the AO should consist of a simple media player without decoding component; the decoding algorithm should be implemented by a streaming gateway, in charge of converting the required mp3-stream into a simpler (in terms of decoding costs) format. However, a lot of portable devices, such as laptop computers and some PDAs, can straightforwardly decode *mp3* files. In this case, the AO can be "heavy" and the decoding algorithm can be encapsulated into it.

Second, the SDI provides services with an architectural support to satisfy different non-functional requirements. Indeed, the same service can be delivered with different quality attributes. In order to improve the level of quality attributes, ad-hoc mechanisms should be implemented by AO and BES. For instance, the *mp3* streaming player may be delivered in two different fashions, namely an unreliable streaming service and a reliable one. Reliability is enhanced by implementing a replication technique. Basically, BESs is replicated according to a passive replication scheme: when the primary server fails, the AO switches its own data source from the original server to the replica, thus restoring service provisioning.

3.4 Implementation issues

This section provides some technicalities about PRINCEPS implementation which could be useful for readers who are going to address similar issues. In particular, two major issues are discussed here: *i*) interfacing clients with the infrastructure; and *ii*) deploying the AO on the client-side.

Interfacing clients with the PRINCEPS infrastructure

In order to perform service discovery and selection, a Jini-aware servlet (JAS) has been developed. SF and CM have been implemented as group of cooperating JASs. Upon connecting to the infrastructure, the user is given information about available services via a web-based interface. As the client connects to the infrastructure, the SF servlet retrieves its descriptor and generates a service list. Such a list is generated by collecting services that are compatible with client characteristics. When the user chooses a specific service, the name of the selected service along with optional arguments (e.g., the song name for an audio streaming service) are posted to the CM servlet. Such a servlet provides a connection to the CD, in order to send a JSD object. It is worth mentioning that CM must know the locations of all the actually available LSs in order to create a JSD object.

Figure 4 shows the unified modeling language (UML) class diagram of a generic JAS. This servlet instantiates an empty vector of Jini *service registrar* objects, i.e., service objects associated to LSs. During the initialization of the servlet, a new *lookup finder* (LF) object is created and it is provided with the *lookup vector* (LV) variable. LF and JAServlet share the LV: the former is in charge of updating it; the latter is responsible for its instantiation. The LF keeps track of all available lookup services, thus allowing the servlet to have a consistent view of the underlying service discovery infrastructure. This is accomplished by periodically

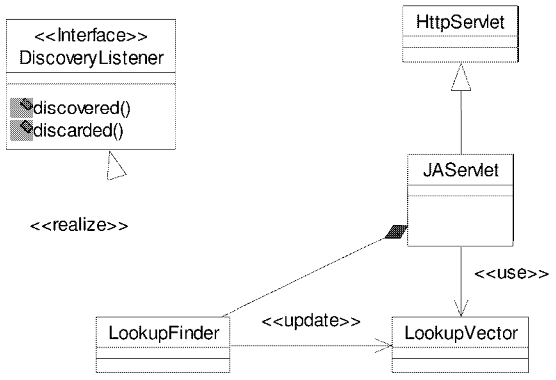

Fig. 4. UML class diagram of the JAServlet

updating the LV.

Four are the operations performed by the CM to create a JSD object. These are described in the following:

1. service parameters and remote user IP address retrieval. This is done by exploiting the functionalities of the *HttpServletRequest*;
2. *service template* creation . This operation is performed by using the posted data;
3. *lookup locator* retrieval. This is returned by the LV object;
4. instantiation and delivery of the JSD object. This accomplished by serializing this object and sending it on a TCP socket.

Retrieval and initialization of the application object

PRINCEPS SDI is based on an applet-like model for the deployment and execution of AOs. CD is a multi-thread Java application, which waits for a JSD to arrive on a TCP socket: upon receiving such an object, the CD is able to connect to a Jini LS, and to download the AO, i.e., it provides a LS with the received *service template.* All the AOs implement a common interface, namely the *GenericService* interface, which contains the default methods that will be invoked by the CD. The *startService()* method is in charge of providing service initialization features (as the init() method of a Java applet does), including user interfaces set-up and binding to remote and local resources. The *stopService()* method is in charge of providing dispose and clean-up operations. CD and the *GenericService* interface must be installed on the client-side; any other specific service interface or package can be included in the AO and downloaded at run-time.

4 PRINCEPS as enhanced web service infrastructure

Web services allow clients to invoke services using HTTP and XML-based wire protocols. From this standpoint, PRINCEPS is a web service architecture. Indeed, like web services,

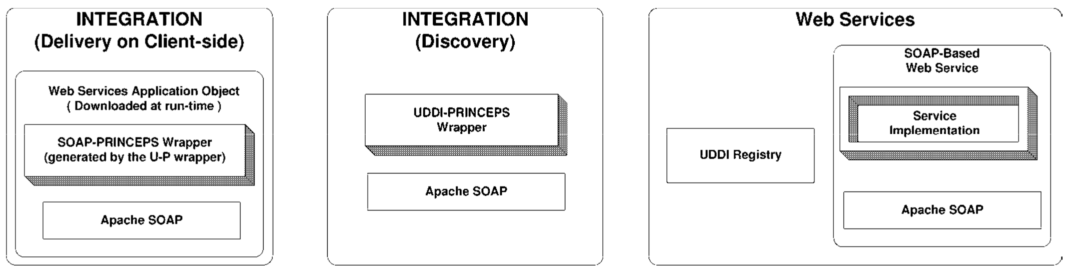
INTEGRATION (Delivery on Client-side)
Web Services Application Object (Downloaded at run-time)
SOAP-PRINCEPS Wrapper (generated by the U-P wrapper)
Apache SOAP
INTEGRATION (Discovery)
UDDI-PRINCEPS Wrapper
Apache SOAP
Web Services
SOAP-Based Web Service
Service Implementation
UDDI Registry
Apache SOAP

Fig. 5. Integration of web services into PRINCEPS

PRINCEPS envisions a "publish-find-bind" scenario. In such a scenario, service provisioning is composed of the following phases: *i*) services are published on a shared registry; *ii*) clients discover services through the registry; *iii*) client binds to the service and effectively uses it. To make PRINCEPS workable in practice as WSA, it is necessary to allow several applications, participating in a service, to correctly communicate on heterogeneous platforms. As mentioned in Section 1, in standard WSAs service repositories are merely conceived as service descriptors containers, without any relationship with the actual availability of services. Indeed, HTTP, SOAP, or one of the transport protocols does not address this important issue. However, one could reuse the HTTP caching semantics which allow browsers and firewalls to cache pages. Unfortunately, this strategy is not under the provider's control; moreover, the requestor may not be using HTTP. Alternatively, a lease mechanism can be integrated in the document exchange mechanism. For instance, encoding of messages between requestor and provider could include the leasing information for the client. However, this leads to a more complex solution. As already stated, PRINCEPS introduces a lease-based strategy for addressing this issue. In order to integrate a web service into PRINCEPS, WSAs interoperability issues have been addressed too. As far as interoperability with UDDI is concerned, the adopted scheme is not a novel one, for it was already presented in [15], which demonstrated that Jini and UDDI are complementary solutions. Our strategy is based on wrapping UDDI registries as PRINCEPS services and has been realized by means of the UDDI-PRINCEPS and SOAP-PRINCEPS wrapping components, depicted in Figure 5. The *UDDI-PRINCEPS wrapper* (UPW) allows UDDI repositories to be registered with the PRINCEPS system. Upon registering UDDI repositories, the UPW extracts all web-services descriptors contained in the UDDI-registries, and creates a *SOAP-PRINCEPS wrapper* (SPW) AO for every service. This mechanism has been implemented by using Apache SOAP [16] Java APIs. This AO allows the user to invoke SOAP methods by means of a graphical user interface (GUI). It is worth noting that the implemented integration mechanism allows to : *i*) discover UDDI-Registries more dynamically than traditional WSAs; *ii*) associate different GUIs to existing web-services; and *iii*) build more complex functionalities by assembling existing web-services, e.g., SOAP-based services, with a set of heterogeneous services, e.g., CORBA-based and even legacy systems.

UPW and SPW enable the integration of web-services into PRINCEPS. A general solution for the reverse integration process, i.e., translating PRINCEPS service into web-services, is not feasible in that ad-hoc solutions are thus needed. Indeed, PRINCEPS provides a flexible support to advanced client-server models, whereas the WSA does not: PRINCEPS code mobility allows the service model to vary from thin to fat client, and viceversa. Therefore, PRINCEPS clients are independent and conceptually unaware of the adopted model; on the contrary, web-services clients are strongly dependent. Therefore, even if BESs may expose their functionality through a SOAP-based interface, such a functionality may vary according to the adopted model: hence, a general solution for exposing PRINCEPS services on a web-services system is not conceivable.

5 A case study application

This section describes a complete example of a real-world application, consisting of a multimedia streaming service. In particular, it shows how SDP and SDI assemble and deliver services according to the capabilities of client devices and to the desired non-functional requirements.

The case-study is a service for the delivery of multimedia contents (audio and video) on demand. From a functional view-point, such a service is defined by the traditional media-management interface (play, stop, pause, fast forward, rewind).

The rest of this section focuses on the provision of such a service with particular quality attributes (i.e., non-functional requirements). For the sake of simplicity, two kinds of client are considered, namely a powerful laptop computer (in the following *heavy client*) and a resource-limited palmtop computer (*light client*).

5.1 Building an error-sensitive multimedia streaming service

Mobile users are often provided with different kinds of network connection, such as GPRS and Wireless LAN. Availability and performance of these network infrastructures may vary according to user location, client device and network congestion. In this context, monitoring the quality of a multimedia streaming and measuring it from the view-point of effective end-users is a crucial issue. In fact, a mobile device may exploit monitoring functions in order to choose the connection which presents the highest quality of service (QoS) "at the moment". Moreover, in a real scenario, users pay for a certain QoS, and service accounting is performed according to the desired QoS. Therefore, an error-sensitive streaming has to be used to monitor QoS as perceived by end-users and to control that no contract-violations occur. The implemented multimedia service is thus composed by the following services:

- multimedia streaming (MSTR): a multimedia streaming service, based on the real-time transport protocol (RTP) [18];
- monitoring service (MMON): an RTP monitoring service which extracts performance data concerning actual RTP sessions and eventually detects performance errors during an active session. This service uses the real-time control protocol (RTCP) [18].

It is worth noting that these services may exist independently from each other.

In order to realize the MSTR and MMON as PRINCEPS services, it is necessary to split the MSTR and MMON services into the AO and the BES. Such components have been here implemented exploiting the Java media framework (JMF) libraries [17]. As far as the MSTR is concerned, the AO is based on a media-player for presenting the requested media to the end-user, whereas the BES is in charge of streaming the requested media data to the end-user by using RTP. As far as the MMON is concerned, the AO evaluates performance of a certain

Table 1. Different versions of the media streaming service

Client	Non-functional requirement	Delivered service
Light Client	None	Simple media player
Light Client	Dependability	ESS with monitoring gateway
Heavy Client	None	Advanced media player
Heavy Client	Dependability	ESS with local monitor

streaming session by: *i*) communicating with the BES in order to evaluate some network-performance parameters (i.e., the network delay), and *ii*) exploiting JMF RTCP APIs to retrieve RTCP reports in order to get RTP-dependent information (i.e., packet loss and jitter) at run-time.

We have built an error-sensitive streaming service (ESS) by assembling the *MSTR* and *MMON* through the PRINCEPS infrastructure. Such a service is composed of:

1. AOs and BESs of service components (i.e. MSTR and MMON);
2. an ESS AO which contains the "composition logic" of the assembled service;
3. an ESS BES which is in charge of registering the service on PRINCEPS and renewing registration leases;

Table 1 depicts the different versions of the media streaming service which we have deployed on PRINCEPS.

5.2 Service discovery issues

Problem

The description and discovery of service implementations depicted in Table 1 require to:

- identify client capabilities (*light client* or *heavy client*) and represent different implementations of the ESS service;
- allow clients to dynamically discover all different versions of the ESS service that match their characteristics;
- present the (assembled) ESS service as single service.

Solution

PRINCEPS allows to satisfy the outlined requirements. Each client descriptor contains all the information that is necessary for distinguishing light clients from the heavy ones. Moreover, as far as services are concerned, it is possible to represent different implementations of the same service through PRINCEPS service descriptors. Furthermore, PRINCEPS SDP enables the automatic discovery of available implementations. Service discovery can be tailored to some meaningful attributes that are used on the client-side to retrieve the AO from the Jini LSs, as shown in Section 3. It is worth noting that service discovery is tailored to client device characteristics at run-time, and eventually to the desired non-functional requirements, since these parameters are passed from the CM component to the CD. Moreover, the service assembly has been masked behind a certain interface, providing the client with the abstraction of a much more complex service.

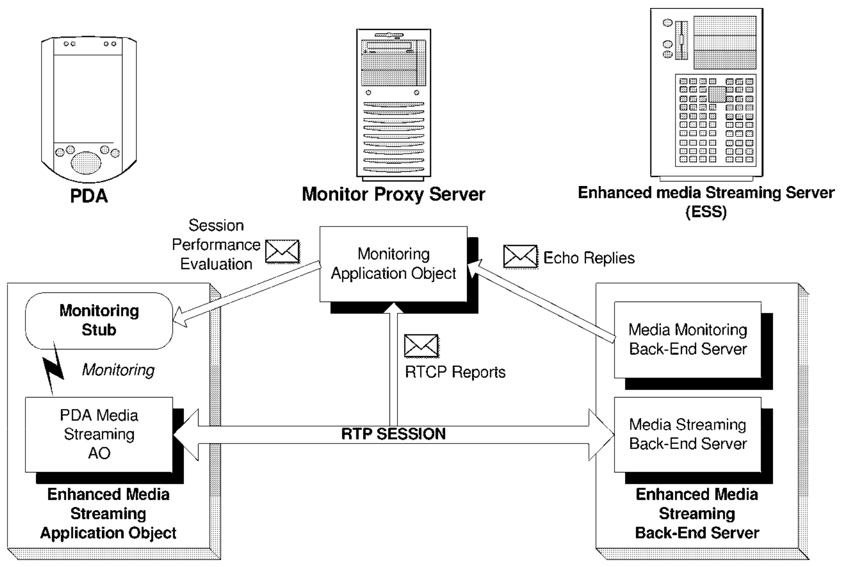
PDA
Monitor Proxy Server
Enhanced media Streaming Server (ESS)
Session Performance Evaluation
Monitoring Application Object
Echo Replies
Monitoring Stub
Monitoring
PDA Media Streaming AO
RTCP Reports
RTP SESSION
Media Monitoring Back-End Server
Media Streaming Back-End Server
Enhanced Media Streaming Application Object
Enhanced Media Streaming Back-End Server

Fig. 6. Delivery of the ESS service to a resource-constrained client

5.3 *Service delivery issues*

Problem

Several client-server models may be adopted for the resulting delivered application, as shown in Table 1. In particular, if the client device is resource-limited, a thin-client approach is preferable; on the contrary, the media streaming service should be provided adopting a fat-client model in order to exploit capabilities of *heavy clients.* Moreover, the deployment of service components at run-time does not depend on implementation details, such as wire protocols and streaming protocol.

Solution

In order to serve *light clients*, PRINCEPS provides the following implementations:

simple media player : a media streaming AO with a light-weight (PDA-suited) GUI is sent to the client; no monitoring functions are performed;

ESS with gateway : the media streaming AO is sent to the *light client*, while the monitoring AO is sent to a monitoring proxy server; by this way monitoring is performed by a third-party entity, namely the monitoring gateway, allowing the ESS service to be delivered even to resource-constrained devices, as described in Figure 6.

Heavy clients may be provided with the following implementations:

Advanced media player : a media streaming AO with a complex and heavy-weight GUI is sent to the client; no monitoring functions are performed;

ESS with local monitoring : both the MSTR and MMON AOs are sent to the *heavy client*, without recurring to a monitoring proxy server.

By describing the above implementations and registering them in the PRINCEPS service directory, we have been able to present the mostsuited implementations to the right clients, and to assemble and deliver their AOs at run-time. In fact, PRINCEPS has enabled to automatically adapt the "media streaming" service to the particular client device and to the desired service attributes, leveraging simple services, such as the MSTR and MMON services, to a much more complex enhanced media streaming service.

It is worth noting that the service is delivered in a plug-and-play fashion without any human interaction. Traditionally, users have to: *i*) look for the needed libraries, i.e., JMF, *ii*) install them, and *iii*) finally configure the framework before starting and initializing a player. PRINCEPS provides mechanisms to automate these processes, and it allows all the needed libraries to be packaged into the application object, thus allowing to free device's memory upon service disconnection.

6 Related work

This section reviews some of the major trends in the area of SOAs and discusses how they relate to our work and in which ways our approach is different.

As stated in [4], discoverability of services is a crucial issue in SOAs. While the standard UDDI technology provides a client-initiated yellow-pages mechanism for looking up services, there are a lot of emerging service discovery infrastructures and architectures, such as Jini, Universal Plug and Play (UPnP), and Bluetooth that might leverage dynamism of WSAs through their support for spontaneous discovery and adaptation of services. The interested reader may refer to [19] for a comprehensive survey of modern service discovery technologies and protocols.

Many current service discovery technologies do not explicitly take into account service delivery issues and non-functional requirements. Moreover, most of them do not provide code-mobility and dynamic composition of services, which are two fundamental means of addressing the outlined issues. Jini is the most important technology which provides Java-based code mobility features. Moreover, language independence, characterizing all the above technologies except from Jini, results in architectures presenting a less dynamic and configurable delivery infrastructure. On the contrary, Jini is based on the Java programming language and constitutes a first step for leveraging Java code-mobility to service pluggability [b]. In this context, the Openwings framework presents some similarities to PRINCEPS. Openwings [20] is a framework for ad-hoc integration of distributed components. It is targeted to the development of distributed systems for mission critical applications. Openwings uses Jini as service discovery infrastructure: like PRINCEPS, it is possible to make Openwings interoperable with different SDPs; so far Openwings has been based on Jini, even though plug-ins for UDDI and Bluetooth technologies are under development. Adaptation to different client devices is not explicitly addressed by Openwings. Therefore, adaptation is more spontaneous in PRINCEPS rather than in Openwings. Openwings is focused on availability, security and interoperability of the delivered services. As far as availability is concerned, this work differs from Openwings in the management of service failures, in that PRINCEPS provides *i*) an automatic lease-based strategy for maintaining consistency of the service directory upon

[b]Pluggability is the possibility of plugging a software component onto a service oriented system and having it automatically provide its services

service failure (as shown in Section 3.2), and *ii*) a BES replication strategy to increase service availability. Openwings, instead, delegates these responsibilities to service developers. The interested reader is referred to [21] for a comprehensive analysis of the behavior of service discovery architectures in presence of communication failures.

Cooltown [22] is a recent SOA for nomadic computing. This work recognizes limitations of web technologies, exacerbating their need to be integrated with sensing and service discovery technologies in order to build nomadic computing systems. More than providing an architecture for delivering services with a service-oriented approach, Cooltown represents an effort to make the web more suitable for nomadic computing. In particular, it presents an approach for context-awareness in web-based environments, providing a means of announcing web-entities whose nature may vary from enterprise application servers to home appliances and embedded systems. In this context the web is conceived as "the most suitable middleware" for connecting heterogeneous distributed components to mobile users. Interactions between users and Cooltown services are exclusively based on web pages: even though this assumption is reasonable when dealing with nomadic computing for embedded systems, it might be too restrictive in the scenario depicted in Section 2, in which different (and much more complex) kinds of AOs are needed in order to effectively exploit client device capability and deliver services with enhanced non-functional requirements.

Agent technologies are an alternative approach for delivering web services. A variety of agent-based solutions has been developed in order to address service provision and adaptation issues [23, 24, 25]. Software agents [26], both intelligent and mobile, seem to be quite suitable for developing service oriented architectures [27]. In particular, the mobile agent paradigm bears a resemblance to our approach [28, 29]. FIPA [30] and KQML [31] specifications constitute the main efforts for standardization of mobile agent systems (MASs). These specifications, promoting agent interoperability, do not explicitly consider adaptivity issues. RAJA (Resource Adaptive Java Agent) [32] is an agent-based infrastructure for mobile resource-adaptive applications which extends a FIPA-compliant MAS providing it with resource management services. RAJA (as well as PRINCEPS) is based on separation of non-functional requirements from application functionality. PRINCEPS AOs differ from mobile agents in that they are downloaded owing to a user action, whereas MAs autonomously migrate from host to host. Moreover, AOs are always executed as single instances; on the contrary, a single instance of a mobile agent may be executed on multiple hosts.

There are projects, presented in [33, 34, 35], which do not explicitly refer to SOA but propose solutions quite similar to those adopted in SOAs. In particular, in [33] a framework is proposed, which allows each service to have multiple implementations that can coexist at the same time. Services are designed to be composable; moreover, applications do not depend on a particular service implementation. The work in [34] proposes a strategy for adapting services to client device characteristics. Such a strategy is location-aware, and can be used of composing services at run-time. Challenges in broadening the Internet to a real ubiquitous computing environment are discussed in [35]. This work aims to provide an infrastructure to make services more independent of different terminal devices, access/core network technologies, and service providers. A local area SOA, namely Vinci, designed for rapid development and management of robust web applications, is presented in [36]. Vinci is based on XML document exchange. Most of the works we have studied mainly focus on the service discovery mechanisms (which are not dynamic, i.e., they statically bind to one running service), and on the concept of service substitutability. Services no longer available are not detected. Moreover, these works consider service discovery and service delivery as two independent processes, related one to each other exclusively by means of sequential timing constraints. Our solution, instead, is based on interaction between service discovery and delivery: in particular, these two activities have been composed to build services "on the fly".

7 Conclusions

This work presented an enhanced SOA, called PRINCEPS, particularly suited for implementing web-service applications. The main contributions of this paper are the SDP and the SDI sub-systems. PRINCEPS implements a novel SDP which is able to assemble services according to functional and non functional client requirements. The foregoing protocol is XML-based, implemented using Jini middleware, and is able to discover and create services at run-time. The implemented lease-based mechanism resulted in a flexible discovery mechanism. An event-driven model was implemented to maintain service directories consistent with actual running service instances. The implemented SDP subsystem allowed to improve service availability by registering a redundant set of server clones which have the same interface and the same lookup attributes. PRINCEPS provides an SDI, which is based on HTTP as transport protocol, and allows extended client-server models to be adopted. This approach can be extremely effective when users own different kinds of devices (having different computing capabilities). While developing the proposed architecture, we always strive to keep compatibility with the web-services standards. To this aim, we carefully designed and implemented two interoperability modules to integrate web-services, defined in terms of SOAP and UDDI, as PRINCEPS services. On the contrary, being PRINCEPS a SOA which is more complex than WSA, a general solution for translating PRINCEPS services into web-services is not feasible in that ad-hoc solutions are thus needed. Finally, the provided example demonstrated that: *i*) a service can be created on the fly by assembling existing services; *ii*) the SDP is able to discover a service depending on client requirements (functional and non-functional), client and service characteristics; and *iii*) service discovery components allow clients to dynamically download the particular application object that satisfies all the above requirements. We believe our experience can be used as guideline for software developers to solve similar problems.

Acknowledgements

This work has been partially supported by the University of Naples Federico II, the Consorzio Interuniversitario Nazionale per l'Informatica (CINI) and by the Italian Ministry for Education, University and Research (MIUR) in the framework of the FIRB project "WEB-MINDS: Wide-scalE, Broadband, MIddleware for Network Distributed Services".

References

1. S. Vinoski (2002), *Where is middleware*, IEEE Internet Comput., Vol.6(2), pp. 83–85
2. Microsoft Corporation (2002), *.NET Framework Reference*, http://msdn.microsoft.com/netframework/techinfo/documentation/default.asp

3. B. Shannon (2002), *Java 2 Platform Enterprise Edition Specification, v1.4*, http://java.sun.com/j2ee
4. A.W. Brown, S. Johnston and K. Kelly (2002), *Large-Scale, Using Service-Oriented Architecture and Component-Based Development to build Web Service Applications*, Rational Software White Paper, TP032
5. M. Champion, C. Ferris, E. Newcomwe and D. Orchard (2002), *Web Service Architecture, working draft 14* http://www.w3.org/TR/2002/WD-ws-arch-20021114/
6. A.W. Brown (2000), *Large-Scale, Component-Based Development*, Prentice Hall
7. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelson, H.F. Neilsen, S. Thatte and D. Winer (2000), *Simple Object Access Protocol (SOAP) Ver. 1.1*, World Wide Web Consortium (W3C), http://www.w3.org/TR/SOAP.html
8. E. Christensen, F. Curbera, G. Meredith and S. Weerawarana (2001), *Web Service Description Language (WSDL) Ver. 1.1*, World Wide Web Consortium (W3C), http://www.w3.org/TR/wsdl.html
9. D. Ehnebuske, B. McKee and D. Rogers (2002), *Universal Description Discovery and Integration (UDDI) 2.04*, http://www.uddi.org/specification.html
10. K. Ballinger, P. Brittenham, A. Malhotra, W.A. Nagy and S. Pharies (2001), *Web Services Inspection Language (WSIL) 1.0*, International Business Machines Corporation (IBM) http://www.ibm.com/developerworks/library/ws-wsilspec.html
11. K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler and J. Waldo (1999), *The Jini Specification*, Addison-Wesley, Reading
12. W.K. Edwards and B. Joy (2001), *Core Jini 2ed*, Prentice Hall PTR, http://www.kedwards.com/jini/
13. Sun Microsystems Inc. (2002), *JavaSpaces Service Specification, Version 1.2.1* http://wwws.sun.com/software/jini/specs/jini1.2html/js-title.html
14. D. Cotroneo, C. di Flora and S. Russo (2002), *A JINI framework for distributed service flexibility*, in proc. of 10th Euromicro Workshop on Distributed and Parallel Network-based Processing, pp. 109 -116
15. S. Ghandeharizadeh, F. Sommers, K. Joisher and E. Alwagait (2002), *A Document as a Web Service: Two Complementary Frameworks*, in proc. of the Second International Workshop on Multimedia Data Document Engineering (MDDE 02)
16. Apache Software Foundation (2002), *Apache SOAP Ver. 2.3.1* http://ws.apache.org/soap/
17. Sun Microsystems Inc. (2002), *Java Media Framework API Ver.2.1.1b*, http://java.sun.com/products/java-media/jmf/
18. H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson (1996), *RTP: A Transport Protocol for Real-Time Applications*, IETF Request for Comment (RFC1889), http://www.ietf.org/rfc/rfc1889.txt
19. A. Rakotonirainy and G. Groves (2002), *Resource discovery for pervasive environments*, in CoopIS / DOA / ODBASE 2002: Lecture Notes in Computer Science, Springer Verlag: LNCS 2519, pp. 866–883
20. G. Bieber and J. Carpenter (2002), *Openwings: A Service Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems, Rev. 2.0*, http://www.openwings.org
21. C. Dabrowski and K. Mills (2002), *Understanding self-healing in service-discovery systems*, in proc. of the first workshop on self-healing systems, ACM
22. T. Kindberg and J. Barton (2002), *People, Places, Things: Web presence for the Real World*, ACM Mobile Networks and Applications, Vol.7(5), pp. 365–376
23. J. Kiniry and D. Zimmerman (1997), *A hands-on look at Java mobile agents*, IEEE Internet Comput., Vol.1(4), pp. 21–30
24. K. Sycara, M. Paolucci, M. van Velsen and J. Giampapa (2003), *The RETSINA MAS Infrastructure*, To appear in the special joint issue of Autonomous Agents and MAS, Vol.7, Nos. 1 and 2, July, 2003

25. P. Bellavista, A. Corradi and C. Stefanelli (2001), *Mobile agent middleware for mobile computing*, IEEE Computer , Vol.34(3), pp. 73–81
26. M. Wooldrige (1999), *Intelligent agents*, Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence, MIT Press, pp. 21–35
27. B. Schulze and E.R.M. Madeira (1997), *Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture*, Mobile Agents, Lecture Notes in Computer Science - LNCS 1219, pp. 74–85
28. Z. Wang and J. Seitz (2002), *Mobile Agents for Discovering and Accessing Services in Nomadic Environments*, in proc. of fourth international workshop on mobile agents for telecommunication applications, Barcelona, Spain, Springer-Verlag LNCS 2510, October 2002
29. J. Jing, A.S. Helal and A. Elmagarmid (1999), *Client-server computing in mobile environments*, ACM Computing Surveys, Vol.31(2), pp. 117–157
30. FIPA: The Foundation for Intelligent Physical Agents, http://www.fipa.org
31. T. Finin, Y. Labrou, and J. Mayfield (1995), *KQML as an agent communication language*, in Software Agents, MIT Press
32. Y. Ding, R. Malaka, C. Kray and M. Schillo (2001), *RAJA: a resource-adaptive Java agent infrastructure*, in proc. of the fifth international conference on autonomous agents, ACM Press, pp. 332–339
33. R. Feiertag, T. Redmond and S. Rho (2000), *A framework for building composable replaceable security services*, in proc. of DARPA Information Survivability Conference and Exposition (DISCEX 00), IEEE Computer Society, Vol.2, pp. 391–402
34. T. Hodes and R. Katz (1999), *Composable Ad hoc Location-based Services for Heterogeneous Mobile Clients*, ACM Wireless Networks, Vol.5(5), pp. 411–427.
35. D. Mandato, E. Kovacs, F. Hohl and H. Amir-Alikhani (2002), *CAMP: a context-aware mobile portal*, IEEE Commun. Mag., Vol.40(1), pp. 90–97.
36. R. Agrawal, R.J.Jr. Bayardo, D. Gruhl and S. Papadimitriou (2002), *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*, Elsevier Computer Networks, Vol.39 http://www.almaden.ibm.com/cs/people/bayardo/vinci/vinci.html