An Enhanced Service Oriented architecture for building Web-based applications

D. Cotroneo, C. di Flora, S. Russo,
Università degli Studi di Napoli “Federico II”
Dipartimento di Informatica e Sistemistica
{cotroneo, diflora, sterusso}@unina.it

Abstract

Service Oriented Architectures have recently emerged as a promising approach to develop Web-Based service provision environments. These architectures are characterized by federating services into a single, dynamic distributed system which is capable of spontaneously discovering and configuring itself upon service connections and disconnections. This work proposes a new architectural model, namely PRINCEPS, of Service Oriented Architecture particularly suited for implementing Web-service applications. In particular, two are the main contributions: Service Discovery protocol and Service Delivery Infrastructure. PRINCEPS implements a novel Service Discovery protocol which is able to assemble service according client’s requirements in terms of desired quality attributes. Such a protocol is XML-based, implemented using Jini middleware, and it is able to discover services at run-time. PRINCEPS is also able to provide a Delivery infrastructure, which is based on the HTTP protocol, implementing extended client-server models. PRINCEPS can be used as a Web Service Architecture, being interoperable with the standard web service technologies carried out by the World Wide Web Consortium (W3C). A complete example, which demonstrates benefits in adopting PRINCEPS architecture, is also provided.

keywords: Service Oriented Architectures, Web Services, Service Discovery, Service Delivery, Jini

1 Introduction

In recent years, advances in network technology and computing systems are leading to a new model of distributed system. Such a model stems from integration of application and/or Commercial Off The Shelf (COTS) components,
and provides users with services that are accessible via a web-based infrastructure regardless of client’s physical location and/or devices. Heterogeneity and diversity are thus the most tricky issues to be addressed when developing such systems. We recognize that even though middleware technologies make easier the diversity and heterogeneity problem, they do not completely solve it. As described in [1], the abstraction layers introduced by a middleware to address integration issues may also cause interoperability problems between different middleware platforms. This is especially true when considering internet-scale software systems. Heterogeneity and diversity of enterprise- and/or internet-scale software systems are nowadays addressed adopting a component based approach, which focuses on assembling systems from distributed services. In this scenario, the concept of service is becoming crucial: a distributed system is composed of a collection of interacting services. Each service provides access to a well-defined set of functionalities. The resulting system is defined as Service Oriented Architecture and its evolution is guided by addition of new services. Several Service Oriented Architectures can be implemented depending on the service implementation paradigm: service implemented in a single machine, distributed on a local area network or distributed more widely across several companies’ networks. A particularly interesting case is when services use the Internet as communication infrastructure (Web-services architecture [9]). Service oriented architecture is not a new notion. The first definition was given in 2000 by a published book [2], and it is important at this time because of emerging Web service technology. As described in [2], service oriented applications have to be developed as independent sets of interacting services offering well-defined interfaces to their potential users. A Service Oriented Architecture has to support applications to browse and discover collections of services, select those of interest, and assemble them to create the desired functionality (i.e. that satisfies client’s requirements). Middleware technologies are changing to address such new challenges: there are emerging new technological approaches such as Microsoft .NET platform [10] and Java 2 Enterprise Edition [11] standards supported by many commercial products such as IBM’s WebSphere and Sun One Application Server. Such approaches, that adopt component-based design ad implementation methodology, are quite suitable for small-scale enterprise applications, because they are based on the concept of creating services through configuration rather than through programming. By separating development from deployment issues, lower maintenance costs can be obtained. However, the main weakness is that configurability of applications has become nearly as complicated as programming. Suffice to think at the deployment descriptor design of Enterprise Java Beans (EJB) [11].

In order to facilitate internet-scale applications, more flexible solutions are thus required. As defined in [4] a service is implemented as course-grained, discoverable software entity that exists as a single instance and interacts with applications and other services. Service Oriented Architectures should have to federate such services into a single, dynamic distributed system which is capable of spontaneously discovering and configuring itself upon service connections and disconnections. This work proposes a new architectural model, namely PRIN-
CEPS (Pluggable Reliable Infrastructure for Network Computing and Enhanced Properties of Services), of Service Oriented Architecture particularly suited for implementing Web-service applications. In particular, two are the main contributions: Service Discovery Protocol and Service Delivery Infrastructure. PRINCEPS implements a novel Service Discovery protocol which is able to assemble services according to client’s requirements in terms of desired quality attributes. Such a protocol is XML-based, implemented using Jini middleware, and it is able to discovery service at run-time. This is based on lease mechanism: each service that become available in the system must renew its lease periodically. Services that are not effectively available loose their leases (lease expired). The implemented mechanism has an interesting advantage: it is possible to increase service availability registering a redundant set of server clones having the same interface. PRINCEPS is also able to provide a Delivery infrastructure, which is based on HTTP as a transfer protocol, and it is able to implement extended client-server models. Indeed, resource limitations of clients may require certain operations to be normally performed on PRINCEPS servers. PRINCEPS supports different paradigms of client-server computing: from thin client architecture (i.e. all the logic is put on the server) to a full client architecture (i.e., all the logic is put on the client). A complete example, which demonstrates benefits in adopting PRINCEPS architecture is also provided. The rest of the paper is organized as follows. Section 2 introduces technologies we exploited for the implementation of PRINCEPS, and it gives some definitions and discusses standards about Service Oriented and Web-Service architectures we used throughout the paper. Section 3 details our complete architecture, highlighting the service discovery protocol and delivery infrastructure. Section 4 describes how PRINCEPS can be used as Web-service Infrastructure, detailing how the PRINCEPS discovery protocol can interoperate with standard one (UDDI), and how the delivery infrastructure can be deployed over the Internet. Section 5 describes a complete example of a service oriented multimedia application. 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 becoming crucial with the emerging of Web Service Architecture(WSA). In fact, as stated in [9], WSAs, both basic and extended architectures, are based on SOA. The relationships between SOA and WSA are provided later in this section. A SOA is characterized by several entities: i) Service, i.e. the logical entity, defined by one or more published interface; ii) Service Provider, i.e the entity that implements a service specification; iii) Service Requestor, i.e. the software entity that calls a service provider (it can be an end-user application or an other service); iv) Service Locator, i.e. a specific kind of service provider that acts as a registry and allows
for the lookup of service provider interfaces and service locations; v) **Service broker**, i.e. specific kind of service provider that can pass on 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 in developing SOAs. Some keys for effective use of services are given in [4] and are briefly summarized in the following:

- **Coarse-grained** - Operations on services are frequently implemented to encompass more functionality and (often) operate on larger data sets;

- **Interface-based design** - SOAs services are characterized by a complete separation between implementations and interfaces: multiple services can implement a common interface and a service can implement multiple interfaces;

- **Discoverable** - Services need to be found at run time, not only by unique identity but also by interface identity and by service kind;

- **Single Instance** - Each service is a single, always running instance that a number of clients communicate with;

- **Loosely coupled** - Services are connected to other services and clients using standards, decoupled message-based methods such as XML document exchanges.

It is worth noting that it is the sum total of these attributes that differentiate a service-based application from an application developed using the recent component architectures such as a J2EE or .NET. A Web Service Architecture is a particular class of service oriented architecture, where each service is identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet-based protocols.

As stated by W3C [9], “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 a service description language”. However the W3Consortium and UDDI community have drafted vendor neutral open standards of the core Web Service Protocols like Simple Object Access Protocol (SOAP)[5], Web Service Description Language (WSDL)[6], and Universal Description Discovery and Integration (UDDI)[7]. However, some novel discovery protocols are emerging as open standards for Web services other than UDDI such as the file based Web Services Inspection Language (WSIL) [8]. This represents a promising step towards extensibility of existing mechanisms for managing Web Services Descriptions. As stated in [8] WSIL documents allow their consumers to pick and choose 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 while WSIL does. WSIL may be considered as an extension of the standard technologies UDDI and WSDL because it provides bindings for WSDL descriptions and for their retrieval from UDDI registries.
2.2 Jini and Javaspaces

This sub-section aims to briefly describe Jini technology [15] pointing out the features that we exploited in order to build our PRINCEPS prototype. In fact, PRINCEPS’ service discovery protocol and delivery infrastructure are based on JINI as a dynamic infrastructure for federating components and provided services; thus, we believe that a preliminary exploration of Jini technology itself may help the reader in better understanding PRINCEPS architecture and its protocols.

Jini technology consists of an infrastructure and a programming model addressing the fundamental issue of how clients connect with each other to form an impromptu community. The aim of Jini is to federate services and their clients into a single, dynamic distributed system which is capable of automatically configuring itself upon service connections and disconnections. Jini provides a software layer for dynamic service discovery and join, enabling services to be added or removed from a federation at any time according to demand, need or changing requirements of the distributed applications.

A Jini service is defined by its functional programming API, based on the Java language. However, for each service a set of attributes may be specified. Attributes are a mean of describing service properties and additional information about a certain service implementation.

The heart of a Jini system is the Lookup Service (LS). LS allows clients to find and use services and allows servers to join the Jini system. Interactions with the Lookup Service follow three specific protocols, which are defined into Jini specifications and described in the following.

The Discovery Protocol is used by both clients and servers to locate available Lookup Services; this protocol is composed by three closely related discovery protocols:

1. the Multicast Discovery Protocol is used to discover one or more lookup services on a local area network (LAN);
2. the Unicast Discovery Protocol is used to establish communication with a specific lookup service over a wide-area network (WAN);
3. the Announcement Protocol is used by each lookup service to announce its presence (to the other LSs) on a local network.

The Join Protocol defines how services can become part of a Jini system: any server wishing to join the system provides the Lookup Service with a Service Object - which is the publically visible part of the provided service - and a set of description 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 system. If the period of the grant for join expires, the Lookup Service removes the service from the system.

The Lookup Protocol defines interactions between clients and lookup services. Clients use it to locate services upon their specifications, encapsulating
attributes and interface into a single Service Template. As a result of any successful lookup operation, the Service Object is dynamically deployed over client devices at run-time: once it has been retrieved, clients interact with it by means of its interface, being unaware of the implementation details. The interested reader may refer to [14] for further details on Jini Technology.

JavaSpaces technology provides a distributed persistence and object exchange mechanism for Java-based software [22] which is based on the tuple space model. JavaSpaces are designed to address issues related to the implementation of distributed algorithm using the Java programming language. This functionality is provided through the abstraction of a shared distributed container of Java objects; JavaSpaces programming model contains a few basic operations for writing, reading and extracting objects from the Space. The reference implementation of the JavaSpaces service is based on Jini technology and it is contained into standard Jini distribution kit.

3 PRINCEPS’ Architecture

3.1 Overall architecture

PRINCEPS (Pluggable Reliable Infrastructure for Network Computing and Enhanced Properties of Services) is a Service Oriented Architecture particularly suited for implementing Web-service applications. A preliminary work on such an architecture was presented in [3], where the focus was put mainly on supporting extended client-server models.

Figure 1 depicts the conceptual model of the PRINCEPS architecture, representing adopted technologies too. As described by this model, the overall architecture is constituted by service discovery and service delivery infrastructures. The latter may be partitioned between the client-side and the server-side. Service requestors and service providers may access these infrastructures only by means of the Client Manager and the Service Manager components respectively. Services and clients are represented by XML descriptors which are used by the service discovery protocol to adapt services to client characteristics.

Client-side delivery infrastructure is composed by the following components:

- **Application Object**: it represents the application on the client side: it contains all the business logic which has not been put on the server-side; it is dynamically downloaded at run-time once a certain service has been requested;

- **Client Daemon (CD)**: it is in charge of performing download and execution of the Application Object;

On the server-side, the delivery and deployment infrastructure is mainly constituted by the Back-End Server (BES) and by an HTTP Server representing the Application Objects repository. The BES is in charge of (i) implementing server-side application logic and (ii) keeping the server-side implementation of the service available. Since PRINCEPS code mobility is based on the HTTP
protocol, one or more HTTP servers should be available in order to enable
dynamic download of application objects.

The Service Discovery sub-system provides functionalities which are used by
other components of the infrastructure and are provided by the two following
components:

• **Service Finder (SF)**: it is responsible of (i) choosing services which best
  fit client characteristics and (ii) assembling services at run-time in order
  to provide more complex ones;

• **Service Directory (SD)**: it manages the repository of descriptors of all
  available services; it updates service list and service descriptors upon
dynamic connections and disconnections of services, and upon dynamic
changes to service attributes;

As already mentioned, external access to PRINCEPS architecture is man-
aged by the following components:

• **Client Manager (CM)**: it provides client registration and service discovery
  functionalities;

• **Service Manager (SM)**: manages service registrations and updates service
  list and service descriptors;
3.2 Service Discovery

Different kinds of issues are addressed by PRINCEPS Service Discovery protocol on the client and on the server side. As far as client-side is concerned, the discovery protocol provides service requestors with the abstraction of a dynamic and adaptive service list, which is made accessible to service requestors through Client Manager’s interface. The service list is dynamic because it is automatically updated upon service connections and disconnections, and it is adaptive because it is tailored according to client characteristics. In order to adapt services to client’s requirements, service discovery is performed by analyzing both client and service descriptors. The client descriptor is an XML file which contains client’s characteristics, in terms of computing power, network connection, main memory, and storage memory. 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. By this way, the same service (i.e., the same service instance) can coexist in the infrastructure with more than one implementation, each of one satisfying different non-functional requirements. It is worth noting that a service may be one real service or may represent aggregation of two or more real services. All service descriptors are held in Service Directory components. It is possible to represent services which are effectively composed by:

1. a new service that has been developed from scratch;

2. the integration of an existing service into the PRINCEPS infrastructure;

3. the assembly of functionalities provided by other services.

As far as the server-side is concerned, in order to provide a flexible mechanism for registering and managing service descriptors, service registrations are managed with a lease-based mechanism. PRINCEPS adopts a lease-based discovery model, i.e. each service provider has to renew its lease periodically. During this phase, it can also update service descriptors by providing new implementations. In order to maintain all Service Directory components in a consistent state, we implemented an event-driven mechanism exploiting JavaSpaces technology [22]. Each provider registering a new service obtains a lease. During the renew phase, the service provider may perform one of the following two operations: i) it performs a simple renewal, i.e. the service has not been changed, or ii) it provides to the Service Manager component a new service descriptor, i.e. the service has been changed. The Service Manager component has been configured to produce an event, containing the new service descriptor, which is persistently collected by the TakeSpace component. As figure 2 shows, such a component is a sort of distributed shared memory, implemented by means of JavaSpaces technology. It is able to delivery events to all registered
Service Discovery components. In other words, such components act as event consumers for the TakeSpace.

If the provider does not renew its lease anymore, the Service Manager component delivers a ‘remove event’ to TakeSpace. This forwards the event to proper Service Directories which erase the descriptor from their local repositories. This means that the service is not available anymore.

The proposed lease-based mechanism has also an important benefit: it improves service availability. Indeed, a problem may arise if a service crashes and a client tries to use a service in the period between its failure and the expiration of its own lease. In this case, the service is not effectively available but the PRINCEPS infrastructure will be aware of this only upon the expiration of the registration lease. PRINCEPS allows to solve this problem: the implemented mechanism has an interesting advantage: it is possible to increase service availability registering a redundant set of server clones having the same interface and the same lookup attributes; this way they have the same Application Object which can be retrieved using the same service descriptor. In fact, when the daemon receives a service descriptor object and performs a discovery operation, it retrieves a redundant set of Application Objects. If the selected Application Object fails, an exception is thrown and the daemon tries again with a new Application Object.

3.3 Service Delivery

Service delivery infrastructure has been realized by adopting an applet-like model for deployment and execution of Application Objects. Each PRINCEPS
service is defined by two components, namely the Application Object and the Back-End Server. As described in figure 3 the Application Object has been implemented as a Jini Service Object. Once the user has obtained the service list, it is able to select a particular service. Upon a service selection, the Client Manager component sends a Jini Service Descriptor (see figure 3) Object to the Client Daemon (exploiting Java serialization on TCP sockets). This object is used to encapsulate a Lookup Locator and a Service Template object. The first one is used to locate active Jini Lookup Services while the second one contains attributes that will be used to download the Application Object exploiting Jini lookup protocol.

Client Daemon uses the received object to retrieve the Application Object in order to begin using the service. It is worth noting that the combination of a web-based selection mechanism with a Jini-based Application Object provision strategy allows the infrastructure to deal with nomadic users, giving them a means of browsing and using services independently from both service location and client location. The abstractions of the Application Object and Back-End Server leverage the capability of the service-delivery infrastructure for two main reasons.

First, these abstractions let the infrastructure deliver services implemented by the client-server paradigm which fits client’s characteristics. In fact, service developers may design different Application Objects for different target devices.
and let the infrastructure adapt services according to client characteristics and service requirements. Indeed, if a palmtop/PDA computer is used, the Application Object should be lighter, while all the service logic should be put on the server-side (thin client model). In contrast, if the client device is a powerful laptop, the Application Object might be heavy, and the server logic may be divided between the client-side and the server-side. For the sake of simplicity, let the service be 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 these devices the Application Object should be constituted by a simple media player with no decoding component; the decoding algorithm should be implemented by a streaming gateway, which is in charge of converting the required mp3-stream to a simpler (in terms of decoding costs) format. On the other side, a lot of portable devices, such as laptop computers and some PDAs, may straightforwardly decode mp3 files. In this case, the Application Object can be “heavy” and the decoding algorithm may be encapsulated into it.

Second, the service delivery infrastructure allows providing a service with different non-functional requirements. Indeed, it is possible to offer a certain service with different quality attributes: different versions of the same service may be treated as like as the ones for different target devices. Non-functional requirements may be managed with two different strategies. The first one is provided by the Service Discovery protocol: as we showed in the previous sub-section, PRINCEPS service discovery protocol provides an implicit support to improve availability of services by means of redundancy. The second and more general one is based on implementation of ad-hoc mechanisms for improving some non-functional requirements into Application Objects and Back-End Servers. For example, the mp3 streaming player may be delivered in two different fashions, namely an unreliable streaming service and a reliable one. Reliability may be enhanced implementing a replication technique. Basically, Back-End servers may be replicated according to a passive replication scheme: when the primary server fails, the Application Object may switch its own data source from the original server to the replica, thus continuing to deliver the service to end-user.

3.4 Implementation issues
This section gives some technicalities about PRINCEPS implementation which could be useful for readers who are going to address similar issues. In particular, we have identified three major issues:

1. interfacing clients with the infrastructure;
2. processing service selection;
3. run-time deployment of the application object on the client-side;

In the remainder of this subsection we shall describe how we addressed each single issue.
Interfacing clients with the PRINCEPS infrastructure

Service Finder and Client Manager (CM) have been implemented as a group of cooperating Java Servlets. As mentioned, CM is responsible of subscriptions and service selections. In order to perform service discovery and selection, we designed and implemented a Jini-enabled servlet, namely the JAServlet (Jini-Aware Servlet). When the user selects a particular service from a web page, his browser is redirected to a JAServlet, namely Client Manager’s ServiceSelected servlet, which receives posted data containing the univocal name of the selected service and optional arguments (like the song name for an Audio Streaming service). Such a servlet is in charge of connecting to end-user’s Client Daemon and sending it a “black-box” Java Object (the Jini Service Descriptor object). Therefore, this servlet knows the locations of all the Lookup Services that are alive at a certain moment.

Figure 4 shows the Unified Modeling Language (UML) class diagram of the Jini-Aware servlet. This servlet instantiates an empty Vector of Jini Service Registrar objects (service objects for Jini Lookup Services). During the initialization of the servlet, a new Lookup Finder object is created and the Lookup Vector variable is passed as a parameter to Lookup Finder’s constructor. Basically, Lookup Finder and JAServlet share the Lookup Vector variable: the former is in charge of updating it, while the latter is responsible of its instantiation (figure 4) and effectively uses it. The Lookup Finder keeps track of all available discovery services and updates coherently servlet’s Registrar Vector, thus allowing Servlets to have a consistent view of the PRINCEPS service discovery infrastructure by means of the Lookup Vector variable.

Processing service selection

As already mentioned, upon a service selection, the Client Manager component sends a Jini Service Descriptor Object to the Client Daemon which uses it
to retrieve the Application Object. In order to enable this mechanism, the Jini-aware servlet provides Client Daemon with the location of available Jini Lookup Services and with a Service Template that must be used to effectively download the application object, as described in the following sub-section.

The mechanism for preparation of a JiniServiceDescriptor has been implemented as follows:

1. Retrieve posted data and user’s IP address: The JAServlet recovers service parameters and remote user IP address exploiting HttpServletRequest’s functionalities;

2. Build the lookup template: JAServlet builds a Service Template using posted data and retrieves a valid Lookup Locator from the Lookup Vector object.

3. send the prepared JSD to Client Daemon: The JAServlet instantiates a JSD object and sends it to the Client Daemon by serializing it on a TCP socket.

Retrieval and initialization of the Application Object

Upon a service selection, the client receives the Application Object and it is able to start using the service. PRINCEPS Client Daemon (CD) is a multi-thread Java application, which is always listening on a certain TCP port: when the user selects a certain service, the Client Manager sends a Jini Service Descriptor to the Client Daemon component exploiting Java serialization on TCP sockets. Upon the reception of such a descriptor, the CD is able to connect to a Jini Lookup Service and download the Application Object (i.e. performs a lookup operation using the received Service Template). This object implements a common interface, namely the GenericService interface, containing default methods that will be invoked by the client daemon. The startService() method provides service initialization features (like the init() method in a Java Applet does), including user interfaces set-up and binding to remote and local resources. The stopService() method provides dispose and clean-up operations. Client daemon and the GenericService interface are the only pieces of Java code that must be installed on the client-side; any other specific service interface or package is included in the Application Object and downloaded at run-time.

4 PRINCEPS as Enhanced Web Service infrastructure

As mentioned in section 2, web services should allow any client to invoke any service over HTTP and XML wire protocols. From this point of view, PRINCEPS is a Web service architecture. Indeed, like Web Services, PRINCEPS envisions a “publish-find-bind” scenario where services are published and clients discover and bind to the services they need on the fly. To make PRINCEPS workable
in practice as a Web service architecture, it is crucial to warrant that various applications on heterogeneous platforms participating in a service communicate correctly and effectively. As already mentioned, PRINCEPS adopts a lease-based discovery model. We claim that this is fundamental issue that one might expect that HTTP, SOAP, or one of the transport protocols would handle this for us. For instance, one could reuse the HTTP caching semantics that allow browsers and firewalls to cache pages, but this is not under the provider’s control, and the requestor may not be using HTTP as transport. Alternatively, the lease mechanism can be integrated in the document exchange mechanism, in that messages between requestor and provider encode the leasing information for the client. However this leads to a more complex solution. As already stated, PRINCEPS uses Jini which is able to provide requestor with information and its lease.

In order to integrate a Web service into PRINCEPS, we addressed UDDI and SOAP interoperability issues. As far as UDDI interoperability is concerned, the scheme we adopted is not a novel one, it has already been presented in [13], which has demonstrated that Jini and UDDI are complementary solutions. Our strategy is based on wrapping UDDI registries as PRINCEPS services. Integration of PRINCEPS with UDDI, SOAP and WSDL has been realized with two main components, namely the UDDI-PRINCEPS and SOAP-PRINCEPS wrappers. The UDDI-PRINCEPS wrapper (UPW) exposes Web Services (whose descriptors are contained in registered UDDI repositories) on the PRINCEPS system. In order to add a UDDI registry to PRINCEPS it is necessary to register it with the UPW. In our first prototype we have created a UDDI-PRINCEPS Wrapper which extracts all Web Services contained into registered UDDI-registries and creates, for each service, a SOAP-PRINCEPS Wrapper (SPW) application object. This mechanism has been implemented using Apache SOAP [24] Java APIs. This application object presents a GUI with a button for each SOAP method that is contained into a certain WSDL descriptor and allows the users to invoke its SOAP methods by clicking on the related buttons. It is worth noting that the implemented integration mechanism allows to:

1. discover UDDI-Registries more dynamically than traditional Web Service Architectures do;
2. associate different User Interfaces to existing Web Services;
3. build more complex functionalities by assembling existing Web Services (i.e. SOAP-based services) with a set of heterogeneous services (i.e. CORBA-based and even legacy systems).

UPW and SPW represent support for 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, while PRINCEPS supports advanced client-server models in a flexible way, the Web Services Architecture does not: PRINCEPS code mobility allows to change the service model form thin to full client (and viceversa); therefore, client application is independent (and conceptually unaware) of the adopted model. In contrast, Web Services’ clients are strongly dependent from the adopted model. Therefore, even if Back-End Servers may expose their functionalities through a SOAP-based interface, these functionalities may vary according to the adopted model: this implies that a general solution for exposing PRINCEPS services on a Web Services system is not conceivable.

5 A case study application

This section aims to illustrate how PRINCEPS can be used to discover and deliver a real application in a service oriented fashion. In particular, this section shows how service discovery protocol and service delivery infrastructure assemble and deliver services according to the capabilities of client devices and to the desired non-functional requirements.

The presented case-study application is a Multimedia Streaming service, namely 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). In this context, availability of different versions of the service, satisfying different non-functional requirements, is a desirable feature.

In the rest of this section we shall focus on the provision of such a service with particular quality attributes. For the sake of simplicity, we shall consider two kinds of possible clients, namely a powerful laptop computer (in the following Heavy Client) and a resource-limited palmtop computer (in the following Light Client).

5.1 Building an Error-Sensitive multimedia Streaming service

Monitoring the quality of a multimedia streaming (Quality of Service, QoS) and measuring it from the view-point of effective end-users is a crucial issue. Usually, different kinds of network connections - such as GPRS and Wireless LAN - are available to mobile users. Availability and performance of these network infrastructures may vary according to user location, client device and
network congestion. In this context, a mobile device may exploit monitoring functions in order to choose the connection which presents the highest quality of service “at the moment”. Moreover, in a real scenario, users pay for service provisioning with a certain QoS and service accounting is performed according to the desired QoS. Therefore, in this context an Error-Sensitive streaming may be used to monitor quality of service as perceived by end-users and control that no contract-violations occur.

Such a service is composed by the following two services:

• Multimedia Streaming (MSTR): a multimedia streaming service based on the Real-time Transport Protocol (RTP) [21];

• Monitoring service (MMON): a RTP monitoring service that extracts data on performance of a RTP session at run-time and eventually detects performance errors during an active session.

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

Being a PRINCEPS service, the MSTR service is composed by an application object and a back-end server: we implemented these components exploiting the Java Media Framework (JMF) [20] libraries. The application object is based on a media-player for presenting the requested media to the end-user while the back-end server is in charge of streaming the requested media data to the end-user by using the Real-time Transfer Protocol (RTP). The Monitoring Service is constituted by a back-end echo server and an active monitoring application object. The application object evaluates performance of a certain streaming session by (i) communicating with the echo server in order to evaluate some network-performance parameters (i.e. the network delay) and (ii) exploiting JMF RTCP APIs to retrieve RTCP reports [21] and get some RTP-dependent information (i.e. packet loss and jitter ) at run-time.

We have built an Error-Sensitive Streaming service (ESS) assembling the MSTR and MMON services by means of the PRINCEPS infrastructure. Therefore, such a service is composed by:

1. an application object which contains the “composition logic” of the assembled service;

2. a Back-End server whose duties include staying tuned in order to register service on PRINCEPS and keep registration leases alive.
3. application objects and back-end servers of service components (i.e. MSTR and MMON).

Table 1 depicts different versions of the Media Streaming service which we have been able to provide by means of our infrastructure.

5.2 Service Discovery issues

Problem
The description and discovery of the Media Streaming service according to the different profiles depicted in table 1 requires to:

- identify client capabilities (Light Client or Heavy Client) and representing different implementations of the ESS service;
- allow clients to dynamically discover all different versions of the ESS service that match their capabilities, namely the standard service and the reliable one;
- present the (assembled) ESS service as an atomic service;

Solution
PRINCEPS satisfies the outlined requirements. Each client descriptor contains all the information that is necessary in order to distinguish 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, automatic discovery of available implementations is enabled by PRINCEPS’ discovery infrastructure. Service Discovery in PRINCEPS may be tuned by specifying some meaningful attributes that are used on the client-side to retrieve the application object from the Jini-based service discovery and deployment infrastructure (as shown in section 3). Since these parameters are passed from the Client Manager component to the Client Daemon, this service discovery process may be tailored (at run-time) to client device’s characteristics and to the (eventually) desired non-functional requirements. Finally, since service interface and implementations are completely independent in PRINCEPS, it is possible to mask a service assembly behind a certain interface, providing the client with the abstraction of a much more complex service.

5.3 Service Delivery issues

Problem
As described in table 1, different implementations of the same service may induce different architectural models on the resulting delivered application. In particular, if the client is resource-limited, a thin-client approach is preferable; on the other side, for the Heavy Client, the Media Streaming service should be provided
adopting a full-client model. Moreover, deployment of service components at run-time should be as more independent as possible from implementation details such as wire protocols and streaming protocol.

**Solution**

In order to serve Light Clients, PRINCEPS allows to provide two different suitable implementations:

**Simple Media Player** : a Media Streaming application object with a lightweight (PDA-suited) GUI is sent to the client; no monitoring functions are performed;

**ESS with gateway** : the Media Streaming application object is sent to the light client, while the Monitoring application object is sent to a monitoring proxy server; by this way monitoring is performed by a third-party entity, namely the Monitoring Gateway, allowing even resource-constrained devices to receive an ESS service (as shown in figure 6).

In order to serve Heavy Clients, the Media Streaming service may be provided through the following implementations:

**Advanced Media Player** : a Media Streaming application object with a complex and heavy-weight GUI is sent to the client; no monitoring functions are performed;
ESS with local monitoring: both the Media Streaming application object and the monitoring one are sent to the heavy client, without the need of recurring to a monitoring proxy server;

By describing these four implementations and registering them on PRINCEPS’ Service Directory, we have been able to present the suited implementations to the right clients and to assemble and deliver their application objects at run-time. In fact, PRINCEPS is able 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 MON 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, the users would have to look for the needed libraries (i.e. JMF), then to install them and finally to configure the framework before starting and initializing a player. PRINCEPS provides mechanisms to automate these tiring processes. PRINCEPS allows to package all the needed libraries into the application object, thus allowing to free device’s memory upon service disconnection.

6 Related Work

Several research projects have been conducted on building SOAs. In the following, we briefly review some of the major trends in the area and discuss 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 to lookup services, there are a lot of emerging Service discovery infrastructures and architectures like Jini, Universal Plug and Play (UPnP) and Bluetooth that might leverage dynamism of Web Service Architectures through their support for spontaneous discovery and adaptation of services. The interested reader may refer to [12] for a comprehensive survey of modern service discovery technologies and protocols.

Service discovery technologies “as they are” present some deficiencies because they do not explicitly take into account service delivery issues and non-functional requirements. In particular, most of them do not provide code-mobility and dynamic composition of services, that are two fundamental means of addressing the outlined issues: Jini is the only technology which provides some basic Java-based code mobility features. It is worth noting that language independence, that characterizes all the above technologies except from Jini, implies that resulting architectures present a less dynamic and configurable delivery infrastructure. In contrast, Jini is based on the Java programming language and constitutes a first step for leveraging Java’s code-mobility to service pluggability. In this context, the Openwings framework presents some simi...

---

1Pluggability is the possibility of plugging a software component onto a service oriented system and having it automatically provide services
larities to PRINCEPS. Openwings [25] is a framework for ad-hoc integration of distributed components. It is targeted to development of distributed systems for mission critical applications. Openwings is based on Jini as service discovery infrastructure: as like as PRINCEPS, it is possible to make Openwings interoperable with different service discovery protocols: at the time of this writing, it is based on Jini but UDDI and Bluetooth discovery plug-ins are under development. Adaptation to different client devices is not explicitly addressed by Openwings. Therefore, such 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, our work differs from Openwings in management of service failures: we provide (1) an automatic lease-based strategy for maintaining consistency of the service directory upon service failure (as shown in section 3.2) and (2) a Back-End server replication strategy to increase service availability. In contrast, the work in [25] delegates these responsibilities to service developers. The interested reader may refer to [26] for a comprehensive analysis of the behavior of Service Discovery architectures in presence of communication failures.

Cooltown [27] is a novel Service Oriented Architecture for nomadic computing. This work recognizes limitations of web technologies pointing out that they 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 applications. In particular, it presents an approach for context-awareness in web-based environments, providing a means of announcing web-presence of entities whose nature may vary from enterprise application servers to home appliances and embedded devices: in this context the web is conceived as “the best middleware” for connecting heterogeneous distributed components to mobile users. Interactions between users and Cooltown Services are solely based on web pages: even if this assumption is reasonable in the context of nomadic computing for embedded devices, it might be too restrictive in the scenario we depicted in section 2 in which different (and much more complex) kinds of application objects are needed in order to effectively exploit client device’s capability and deliver services with enhanced non-functional requirements.

There are projects presented in [16, 17, 18] which do not explicitly refer to SOA but proposed solutions are quite similar to those adopted in SOAs. In particular in [16] is proposed a framework which allows each service to have multiple implementations that can coexist at the same time. Moreover, services are designed to be composable. Applications and services are decoupled so that an application does not depend on a particular service implementation. In [17] focus is put on adaptation of a client device’s functionality to new services it discovers as it moves into a new environment. Challenges in broaden the internet to a real ubiquitous computing environment are discussed in [18]. Motivation of the work is to provide an infrastructure to make services more independent of different terminal devices, access/core network technologies, and service providers. A local area service-oriented architecture, namely Vinci, de-
signed for rapid development and management of robust web applications is presented in [19]. Based on XML document exchange, Vinci is designed to complement and interoperate with wide area service-oriented architectures. All the previous works are focused mainly on the service discovery mechanisms (which are not dynamic, i.e. they statically bind to one service running) and on the concept of the service substitutability. Service which are no longer available are not detected by the proposed systems. Moreover, the analyzed 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 of service discovery and delivery: in particular, these two activities may be composed in order to build services “on the fly”, as previously described.

7 Conclusions

This work presented a new architectural model, namely PRINCEPS, of Service Oriented Architecture particularly suited for implementing Web-service applications. Two are the main contributions: Service Discovery Protocol and Service Delivery Infrastructure. PRINCEPS implements a novel Service Discovery protocol which is able to assemble services according to client’s requirements in terms of desired quality attributes. Such a protocol is XML-based, implemented using Jini middleware, and it is able to discovery and create service at runtime. The implemented lease-based mechanism resulted in a flexible discovery mechanism. Furthermore, an event-driven model was implemented to maintain consistency among Service Directories. This mechanism has also an important benefit: it improves service availability by registering a redundant set of server clones having the same interface and the same lookup attributes. PRINCEPS provides a Delivery infrastructure, which is based on HTTP as a transfer protocol, and it is able to implement extended client-server models. This approach can be extremely effective when client connects with different kind of devices, with 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. In contrast, being PRINCEPS a SOA which is more complex than Web Services Architecture, a general solution for translating PRINCEPS service 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 service discovery is able to discover a service depending on client requirements (functional and non-functional), client characteristics, and the kind of service, and iii) service discovery components allow clients to dynamically download the application object which satisfies all the above requirements. We believe our experience can be used as a guideline for software developers to solve similar problems.
ACKNOWLEDGMENTS

This work has been performed with the support of the Italian Ministry of Research and University (MURST) under grant Progetti Cluster "Cluster 16 Multimedialita': LABNET2 "and under a grant PRIN "MUSIQUE".

References


23