Identification, Retrieval and Management of Services in Ambient Networks

N. Kalaydjiev

Key Words: *Pervasive computing; context-aware networks; gridsystems; autonomiccommunication; context-oriented networks; programmable context networks.*

Abstract. *Connecting devices other than workstations to the Internet is becoming commonplace now. Examples are the equipment, traditionally connected to a computer like a printer, a scanner or a disk (file server). In addition, the number of mobile devices like PDA's and telephones increases rapidly. All these devices are capable of providing services to users. Ambient networks and environments pose unique challenges, different from the challenges in conventional and static networks. Devices that need to join and leave the network dynamically require architectures and protocols aiding at making ease of the interaction between network participants. Service consumers need flexible ways to find the exact service provider they need. Networks like these could have static structure and dynamic content, but also could be spontaneously created (i.e. ad-hoc networks) when devices appear close together. This paper is a survey of protocols and taxonomy of architecture capabilities of existing service discovery technologies.*

1. Introduction

This paper describes existing service discovery protocols that are in use today. It focuses on service registration, the possibilities to search for a particular service in a network and how it can be invoked and utilized. The presentation starts with protocols oriented for relatively static environments, like DNS-Service Discovery, Service Location Protocol, UDDI, and then makes a step towards more dynamic networks – architectures and protocols supporting dynamic network structures, like JINI, Universal Plug and Play, Bluetooth. Finally, a strategy called Salutation aiming at unifying different protocol approaches is presented at the end. A short example is provided with each protocol overview, demonstrating how discovering a service looks like using the particular technology. A comparison of examined protocols is provided as a last chapter.

2. Web Services as a Research Problem

There are many different technologies enabling Web services to exist. Each one offers different level of completeness and different degree of built-in support for applications (clients and service suppliers). In a network, where a lot of services and a lot of clients exist simultaneously, a mechanism for interaction between them is necessary. Also, they may need to interact without knowing anything about each other in advance. This could be achieved by standartisation of steps for registration, discovery and activation of possible services. How to find a concrete service instance among all service suppliers? How to sift out by properties and attributes the results we have found, if they are more than one? How to present this information to the human user, so he can decide by himself, and how later successfully activate the service and communicate with it? All these are questions that a service technology has to take into account. A complete service technology needs to offer a conception and a mechanism for all steps from discovery to service activation. Some of the major tasks and issues that it has to manage with are:

- Client and Service Supplier mobility.
- Service auto-configuration when plugged into the network.
- Formal service description of class, category, service attributes, group.
- Service Registration a service needs ways to publish itself so clients can discover it.
- Service Discovery – Lookup and Service Browsing.
- Filtering of possible services by particular criteria - class, attributes, category, and group.
- Adequate and unique addressing of the service in a network.
- Formal description of service activation – methods, parameters, types.
- Application frameworks – API, easy and transparent activation, proxies.
- Remote events – notifications for service registration/deregistration, status change, etc.
- Abstraction of communication mechanism from particular network and media.
- Undependability of particular programming language and hardware/software platform.

All service discovery protocols further on are examined from the perspective of ways they realize the points described above.

2. DNS-SD

DNS Service Discovery uses the standard existing infrastructure to allow simple service discovery. It runs over IP-based networks (small, enterprise, or even Internet) and does not need any new configuration of existing DNS services, servers, and packet formats to browse the network [1]. Locating a service means finding its address on the network, when its type and characteristics are already known. This resembles the way where a conventional Web-browser finds the address of a host in Internet, when it knows its name. A big advantage of DNS-SD is that it does not require any new infrastructure to support service discovery, meaning no new hardware or software addi-

tions. All it needs is a running DNS server, which is true for many present networks [1]. Normally all participants in the network are already configured to use this DNS server (with manual configuration or automatically via DHCP server). Furthermore, the DNS-SD service does not have to be provided by the same DNS server hardware that is currently providing an organization's conventional host name lookup service. Service discovery using DNS-SD can be managed by the existing DNS server in the network, or by a dedicated one, or both of them can work together.

Using Internet-protocol networks as granted, DNS-SD offers the ability to query for services of a certain type in a certain logical domain and receive in response a list of named instances (network browsing, or Service Instance Enumeration) [1]. That way for a single service type it is possible to obtain all service providers that are available to operate the service. Later, when a particular named instance is selected from this list, using DNS the client can efficiently resolve that instance name to the required information to use the service, i.e. IP address and port number. An important detail about DNS-SD is that service instance names should be relatively persistent. If a user selects their default printer from a list of available choices today, then tomorrow they should still be able to print on that printer with no additional configuration. In DNS-SD this is achieved with one-to-one mapping between service instance and service URL. Thus, clients can remember the service URL of the service they have already found, and do not have to search for it again.

To store information about services DNS-SD uses standard DNS records – SRV, PTR and TXT [1,2]. SRV records can describe a service available in a domain, specifying host's IP address and port number to use. The protocol is either TCP (when SRV record has the form _tcp.domain) or UDP (SRV form is _udp.domain). For example, SRV record with the name „_lpr._tcp.mydomain.com." would allow a client to discover and get a list of all printers running LPR printer protocol in mydomain.com. To allow naming of service instance, another DNS record can be used – PTR. It allows creation of references to SRV records, adding new attribute – the name of service provider. Then the name of the service gets the following form:

Service Instance Name = <Instance>.<Service>.<Domain>

Some examples could be: UsualPrinter._lpr._tcp.mydomain.com
AdvancedPrinter._lpr._tcp.mydomain.com
WordTranslator._soap._udp.mydomain.com

Sometimes IP address and port number are not enough as address information. For example, a printer can have a print queue where documents can be sent for printing, or a file server may have multiple volumes, each identified by its own name. Therefore additional information stays in the third DNS record – the TXT record. This information should be regarded the same way as the IP address and port number – it is one component of the addressing information required to identify a specific instance of a service being offered by some piece of hardware. The specific nature of that additional data, and how it is to be used, is service-dependent, but the overall syntax of the data in the TXT record is standardized.

A service is registered like normal DNS-entries maintenance [1,11]. This could be done manually, by some tool that checks network members, or dynamically by each service itself, using the Dynamic DNS Update. When no conventional DNS server is running in the network, an alternative approach could be use of Multicast DNS (mDNS) [3]. It provides the ability to do DNS-like operations on the local network in the absence of any conventional unicast DNS server using IP Multicast. Thus, the network requires a little or no administration or configuration, and it can work when no infrastructure is present, or during infrastructure failures. DNS is organized as a hierarchy of servers, in the familiar Internet domain names. This scheme has been shown to scale up to the entire Internet [11].

Nevertheless, DNS-SD is a static environment where participants are relatively persistent and do not join and leave network frequently. This service discovery protocol is suitable for persistent services in IP-based networks running TCP or UDP protocols, allowing no personal configuration for any client. Service clients are allowed to be as mobile as they need to, and they can safely cache service URLs of services they know of. Administration of registered services is awkward because DNS databases are maintained by privileged users [11]. The big advantage of DNS-SD is that it can use existing infrastructure, standards, protocols and tools, and requires no need of additional software or hardware components.

3. Service Location Protocol (SLP)

Service Location Protocol allows computers and other devices to find services in an IP-based local network. The protocol and the network need little or no static configuration to run SLP, as long as the network supports UDP and TCP protocols [4]. SLP can scale from small and unmanaged networks to large enterprise networks and allow network participants to find existence, location and attributes of services they need, as well as it allows services to be published on the network. In SLP each service is identified by its unique URL, used to locate the service [4,11,10]. This Service-URL specifies the name of the service, protocol used to access it, its location and optionally additional information about it, like some attributes. Moreover, each service can be attended to one or more groups, called scopes that logically divide the network [4]. When a device needs a service, it can always specify a group where to look for

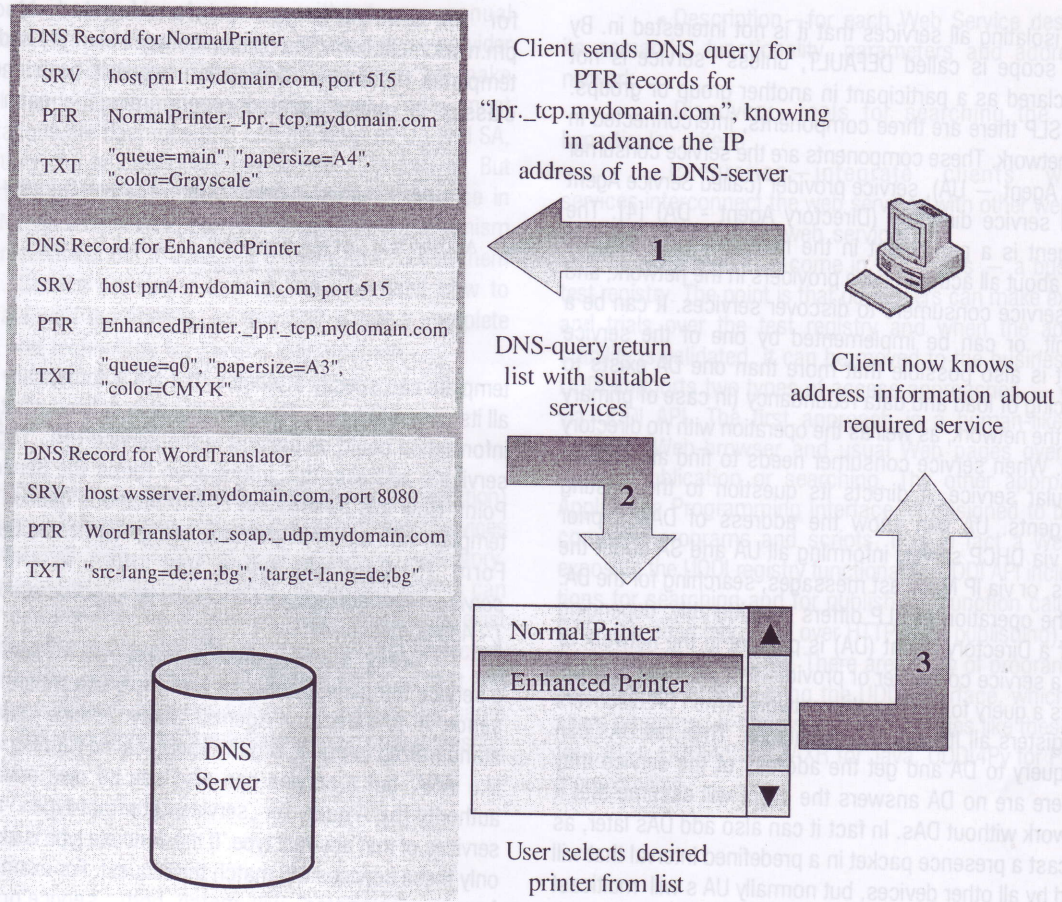


Figure 1. Operation of DNS-SD

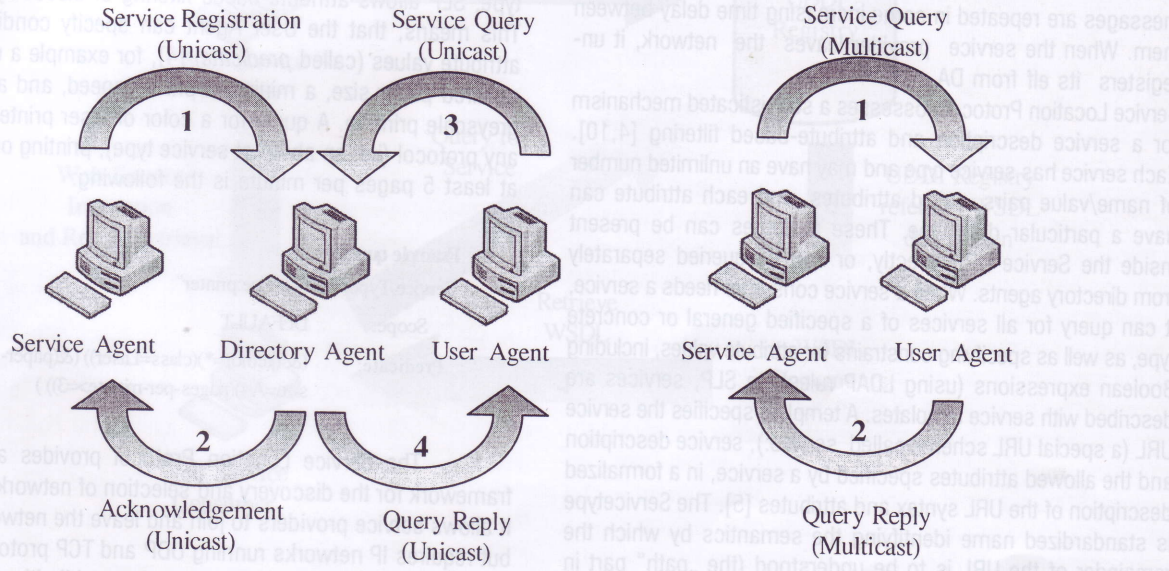


Figure 2. Service Discovery in SLP — left: when at least one Directory Agent is present; right: when no Directory Agent exists

it, and thus isolating all services that it is not interested in. By default, the scope is called DEFAULT, unless service is not explicitly declared as a participant in another group or groups.

In SLP there are three components, interconnected in a common network. These components are the service consumer (called User Agent – UA), service provider (called Service Agent – SA) and service directory (Directory Agent - DA) [4]. The directory agent is a participant in the network that maintains information about all active service providers in the network, and is used by service consumers to discover services. It can be a separate unit, or can be implemented by one of the service providers. It is also possible that more than one DA exists to allow balancing of load and data redundancy (in case of primary crashes) in the network, as well as the operation with no directory agent at all. When service consumer needs to find an address of a particular service, it directs its question to the existing directory agents. UA can know the address of DAs if prior configured via DHCP server, informing all UA and SA about the existing DAs, or via IP Multicast messages, searching for the DA.

The operation of SLP differs considerably, depending on whether a Directory Agent (DA) is present in the network or not. When a service consumer or provider first joins the network it multicasts a query for DAs on this network. When SA discovers a DA, it registers all its services at this DA. Then, all UAs can unicast a query to DA and get the address of the service they need. If there are no DA answers the client will assume that it is in a network without DAs. In fact it can also add DAs later, as they multicast a presence packet in a predefined interval that will be received by all other devices, but normally UA s will multicast a packet with the query. Then, all SAs that contain matches will send a unicast answer back to the UA. As a rule all queries and replies are sent in UDP packets, but when the answer is too big to fit into single UDP packet, it will be sent over a connected unicast TCP link. As UDP is an unreliable protocol, all lost messages are repeated in some increasing time delay between them. When the service provider leaves the network, it un-registers its elf from DA.

Service Location Protocol possesses a sophisticated mechanism for a service description and attribute-based filtering [4,10]. Each service has service type and may have an unlimited number of name/value pairs, called attributes, and each attribute can have a particular data type. These attributes can be present inside the Service URL directly, or can be queried separately from directory agents. When a service consumer needs a service, it can query for all services of a specified general or concrete type, as well as specifying constrains for attribute values, including Boolean expressions (using LDAP rules). In SLP, services are described with service templates. A template specifies the service URL (a special URL scheme called service:), service description and the allowed attributes specified by a service, in a formalized description of the URL syntax and attributes [5]. The Servicetype is standardized name identifying the semantics by which the remainder of the URL is to be understood (the „path" part in standard URL). It may denote either a particular network protocol, or an abstract service type. As an example a Service-URL along with the attribute names and data types for a shared printer could look like this:

This denotes a service URL for a queue called tray1 for the abstract type printer, located at address prn.mydomain.com. The relation between abstract and concrete templates resembles the relation between base and derived classes in object-oriented programming – each abstract

```
URL:         service:printer://prn.mydomain.com/tray1
             class=string
Attributes:  location=string
             color=keyword
```

template can specify attributes that are automatically derived to all its concrete types. The template also defines what additional information could be present in the Service-URL. Together with service's address, this information forms the Service Access Point. How this information is interpreted is defined inside the template with a formal grammar using Augmented Backus-Naur Form [5].Concrete types use URLs in the form of *service:<AbstractType>:<ConcreteType>://<Address> [/<AdditionalInformation>].*

When a service provider registers a service, it prepares a service template, combined with a set of attribute values and sends it to Directory Agents. Many Service templates are standardized in public organizations called Naming Authorities, like IANA, but a service can also use its own private naming authority [5]. A query for „service:<abstract-type>" matches all services of that abstract type. If the concrete type is also included, only these services will match the request. For example, a query for „service:printer" will match both „service:printer: lpr:// hostname" and „service:printer:http://hostname", whereas a query for „service:printer:http" would match only the second service.

In addition to searching by service abstract or concrete type, SLP allows attribute-based filtering of discovery results. This means, that the User Agent can specify conditions for attribute values (called *predicate*) [4], for example a minimum required paper size, a minimum printer speed, and a color or greyscale printing. A query for a color or laser printer running any protocol (i.e. an abstract service type), printing on A4 with at least 5 pages per minute is the following:

Example query:

```
Service Type:  "service:printer"
Scope:         DEFAULT
Predicate:     (& (|(color=*)(class=Laser)) (&(paper-
               size=A4)(pages-per-minute>=3)) )
```

The Service Location Protocol provides a scalable framework for the discovery and selection of network services. It allows service providers to join and leave the network easily, but requires IP networks running UDP and TCP protocols [11]. It offers rich tools for describing service and limiting discovery results according to the predefined circumstances [4,5]. Implementation of this relatively complex protocol could be an overhead for small ambient networks, although in some cases it is possible to eliminate TCP and implement only UDP protocol.

The protocol is capable of running without any manual configuration [11], and the presence only of one service provider and one service consumer is enough for them to make successful handshaking and service discovery. Today most implementations are daemons that can act both as UA and SA, and often they can be configured to become a DA as well. But it is important that this protocol is really used in practice in today's LANs. As a defect this protocol offers only a mechanism to discover services, but it does not explain how to use them afterwards. Service consumer has to know by itself how to communicate with the service, so this SLP is not a complete framework and technology for service management.

4. UDDI

UDDI (Universal Description, Discovery and Integration) is a technology, whose primary purpose is to publish Web Services into the World Wide Web. UDDI acts like a centralized repository, where business can publish its services, and where clients can look up for services they need. Several UDDI repositories exist currently (IBM, Microsoft, SAP) [6], and they are interconnected, i.e. the information is replicated on all of them and the client and business can use each of these servers to access the same information. The repository has the following main characteristics [6]:

- Description – for each Web Service description of its semantic, functionality, parameters and address in the network.
- Discovery – tools for searching the repository byparticular criteria.
- Integration – integrate clients with web services,interconnect the web services with other web services, and business with the web services.

UDDI registries come in two forms – a business and test registry. The point is that developers can make experiments and trials over the test registry and when the applicationis stability is validated it can be moved to the business registry. UDDI supports two types of access approaches – via Web or via UDDI API. The first approach is a human-like onewit a standard Web-browser and usual Web pages over HTTP for service publication or searching. The other approach, UDDI Application Programming Interface, is designed to be used by computer programs and scripts. It is in fact a Web Service, exposing the UDDI registry functionality. UDDI API includes functions for searching and for publishing. Function calls are over HTTP (for searching) and over HTTPS (for publishing), encapsulated in SOAP messages. There are plenty of program modules and libraries implementing the UDDI interface, which ease the client programs or scripts access. Libraries like these are .NET for C/C++, UDDI4J and JAXR for Java, UDDI4Py for Python and many others.

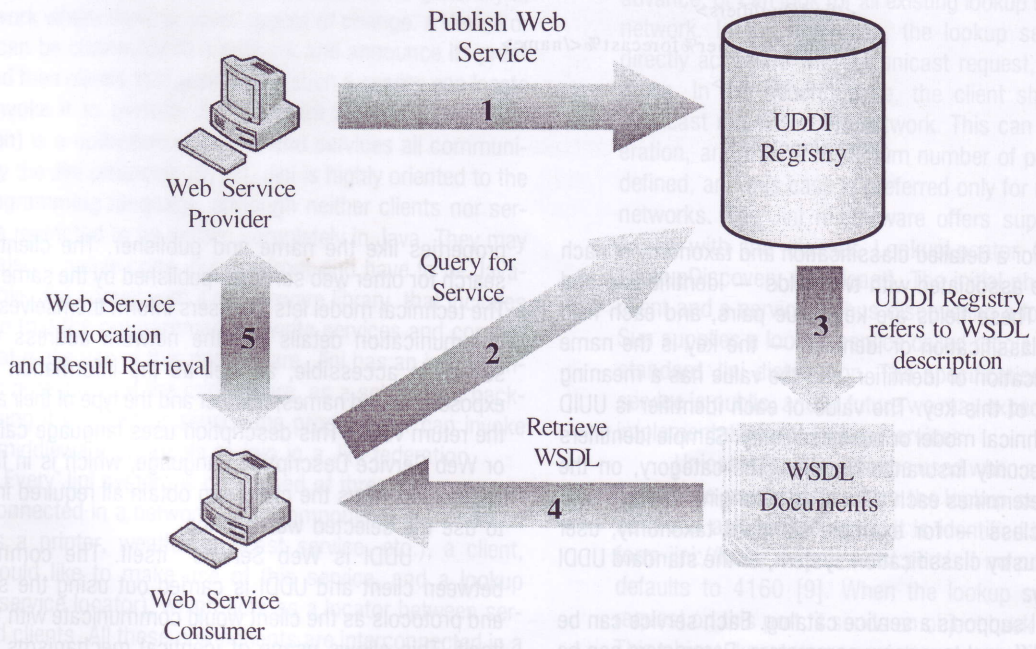


Figure 3. UDDI Service Discovery

UDDI service description model includes Business Entities, Services and Technical Models. Each business entity specifies company or person publishing web services. Each business entity has a unique identifier, called UUID, and can define any relationship with other business entities. Business entities publish one or more Web Services that describe particular family of technical services. Finally each service has a set of Binding Templates, specifying service entry point and implementation details, which allow clients to connect and communicate with the service. Service description however is physically separated from the business entity. This separation is realized with the usage of Technical Models (shortly tModels) [6]. Each tModel describes one service and its properties and characteristics. Each tModel has also a unique identifier, and can be referenced from one or more business entity, service or template.

The tModel, besides a name and service description in one or more languages, refers also to external WSDL description file. This external file is referenced in the form of URI, and thus facilitates the publisher to change the description if necessary. For a detailed classification and taxonomy of each tModel can be associated with two fields – identifierBag and categoryBag. These fields are key-value pairs, and each field defines one classification or identifier – the key is the name of this classification or identifier, and the value has a meaning in the context of this key. The value of each identifier is UUID of another technical model or business entity. Sample identifiers could be a Security Insurance Number. The category, on the other hand, determines each tModel as belonging to some kind of semantic class – for example standard taxonomy, user keywords, industry classification system, or the standard UDDI type taxonomy.

UDDI supports a service catalog. Each service can be classified by different taxonomy parameters. Parameters can be geographic location (with pre-defined index), an industry code, or user attributes and keywords. Every client can search for a service, specifying a service name (or just beginning of the name), or additional conditions for narowing the result set with different taxonomy parameters.

The following example is a SOAP message [8], addressed to UDDI registry (for example http://uddi.ibm.com/beta/inquiryapi) that will query about all registered Weather Forecast services. If business Key attribute refers to valid Business Entity, then the search would be only for services published by this Business Entity. When it is omitted (or an empty string), the query will search all services no matter who their publisher is. We specify the name of the service using a wildcard character, thus „Weather%forecast%" will match any service name that begins with „Weather" and contains the characters „forecast" anywhere to the right of the characters „Weather". The find qualifier in the example will return the result sorted alphabetically. Maximum count of matched services can be controlled with maxRows attribute – this would allow the client to put some limitation on the result number and keep the response message at an acceptable length. UDDI registry itself will also automatically limit the maximum result count to some predefined value [6]:

Once a suitable Web Service is found in the registry, UDDI makes it possible for the user to download its technical and user description. This includes the service's tModel and properties like the name and publisher. The client can also search for other web services, published by the same publisher. The technical model lets the users inform themselves about the communication details like the network address where the service is accessible, as well as the information about the exposed method names, number and the type of their arguments, the return value. This description uses language called WSDL, or Web Service Description Language, which is in fact a XML application. Thus the client can obtain all required information to use the selected web service.

UDDI is Web Service itself. The communication between client and UDDI is carried out using the same rulas and protocols as the client would communicate with the service itself. This allows usage of identical mechanisms to access both the service and the exposed UDDI functionality. UDDI uses standard, open, existing and proven protocols from the Internet – transport protocol HTTP, communication protocols – SOAP and XML. This allows easy communication between different applications that do not know each other in advance, through using the existing Internet standards like XML and HTTP. As a

SOAP Message for Service Query

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
<Body>
   <find_service businessKey="" generic="2.0" xmlns="urn:uddi-org:api_v2">
   <findQualifiers>
         <findQualifier>sortByNameAsc</findQualifier>
   </findQualifiers>
   <name>Weather%forecast%</name>
   </find_service>
</Body>
</Envelope>
```

consequence of this, each system, able to communicate via the Internet standards can communicate with any Web Service, without any importance of a particular program language, hardware platform, operation system, etc.

UDDI makes clear separation between business and technical aspects of the web services, and between service publishers and the services themselves. Web service suppliers (publishers) are grouped in a dedicated group, and the services – in another group. Each service is bound to its own provider. This model allows accurate description of the business relationships between suppliers that is not affected by the technical web service interaction. UDDI supports different business relations like peer-to-peer, parent and derived, etc. On the other hand, each service is physically separated from its communication details. Business entities take the topmost place in the UDDI data model hierarchy, and contain all the services. Thus UDDI accents over the business aspect (providers, companies, business organizations), and the technical details take the second place. UDDI is not oriented to the mobile and dynamic networks, where different devices and nodes join and leave often the network. UDDI is rather oriented to relatively static networks and static „players". It plays a great role in the service discovery in the global network Internet.

5. JINI

Jini is another distributed computing environment that offer „network plug and play" [9]. Jini can be used for mobile computing tasks where a service may only be connected to a network for a short time, but it can be used more generally in any network where there is some degree of change. A device or service can be connected to a network and announce its presence, and then clients that wish to use such a service can locate it and invoke it to perform tasks. A Jini system (called also federation) is a collection of clients and services all communicating by the Jini protocols [9,12]. Jini is highly oriented to the Java programming language, although neither clients nor services are restricted to be written completely in Java. They may include also a native code, but the front-end have to be Java-compatible. Jini comes with a middleware library, that includes an API so that the programmer can write services and components that make use of this middleware. Jini has an implementation (in pure Java) of the middleware, as a set of Java packages. Using it in client or service the programmer can invoke the Jini middleware protocols to join in a Jini federation.

Every Jini system is comprised of three main components, connected in a network [12]. Components are a service (such as a printer, weather forecast service, etc.), a client, which would like to make use of this service, and a lookup service (service locator), which acts as a locator between services and clients. All these components are interconnected in a network, and Jini specification is independent of the network protocol used, although currently only TCP/IP implementation exists [9,12]. To accomplish the task of remote control, Jini uses server proxies. The proxy is downloadable at a client side object, which exposes the interface with the remote service. The proxy takes care of the service provider communication, probably using RMI, so the client only has to call proxy's methods to finish his job. The proxy is the part of the service that is visible to clients, but its function will be to pass method calls back to the rest of the objects that form the total implementation of the service.

A special service, called the „lookup service", must always exist in one Jini network [12, 9]. This service will act like a repository for all published services – the providers will register their services into the lookup service, and the clients will query it when they need to use a service. Such a service will usually have been started by some independent mechanism. In fact, the lookup service is just another Jini service, but it is one that is specialized to store services and pass them on to clients looking for them. In a Jini network, several lookup services can exist simultaneously, and each one of them has its own set of registered services. A LAN may run many lookup services to provide redundancy in case one of them crashes [12]. Services can be semantically combined in groups depending on any user defined property or classification. Each lookup service can attend any number of groups in the network. For example, a company may have Engineering and Public groups, serving the Engineering Department and all other departments, and a dedicated lookup service for any group. Lookup services in a network can overlap and bring information redundancy in order to raise the reliability in case one lookup service fails or crashes. When a client needs a web service, the first thing it should do is to find a lookup service. The client may know the address of the lookup service in advance, or can look for all existing lookup services in the network. In the first case, the lookup service can be directly accessed with an unicast request, using its address. In the second case, the client should make a multicast request in the network. This can be heavy operation, and thus a maximum number of packet hops is defined, and this case is preferred only for usage in local networks. The Jini middleware offers support for both cases with the classes LookupLocator (unicast) and LookupDiscovery (multicast). The initial phase of both a client and a service is thus discovering a lookup service. Sun supplies a lookup service called reggie as part of the standard Jini distribution. The specification of a lookup service is public, and in future we may expect to see other implementations of lookup services.

Unicast discovery can be used when a client already knows the machine on which the lookup service resides, so it can ask for it directly. It is identified by URL in the form jini://host/ or jini://host:port/. If no port is given, it defaults to 4160 [9]. When the lookup service gets a request on this port, it sends an object back to the server. This object, known as a registrar, acts as a proxy to the lookup service, and runs in the service's JVM (Java Virtual Machine). Any requests that the service provider needs to make of the lookup service are made through this proxy registrar. Any suitable protocol may be used to do this, but in practice implementations will probably use RMI. A summary of the whole picture, along with sample

Java code for unicast search is shown in *figure 4*.

If the location of a lookup service is unknown, it is necessary to make a broadcast search for one. UDP supports a multicast mechanism, which the current implementations of Jini use. Since multicast is expensive in terms of network requirements, most routers block multicast packets. This usually restricts broadcast to a local area network, although this depends on the network configuration and the time-to-live (TTL) of the multicast packets.

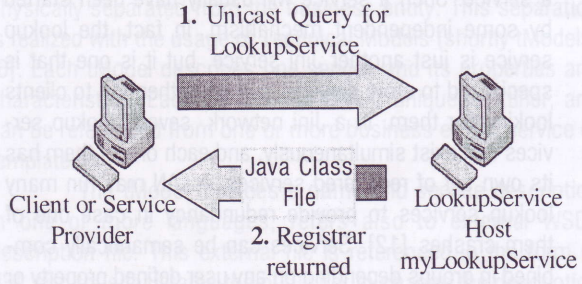


Figure 4 . Unicast Lookup Service Search

Simple Java code illustrating unicast LookupService search:

```
try {
      LookupLocator lookup = new LookupLocator("jini://myLookupService");
      ServiceRegistrar registrar = lookup.getRegistrar();
      System.out.println("Service registrar at " + registrar.getLocator().getHost());
}
catch(Exception e) { // handle exception...
}
```

There may be any number of lookup services running on the network accessible to broadcast search. In a small network, such as a home network, there may be just a single lookup service, but in a large network there may be many – perhaps one or two per department. Each one of these may choose to reply to a broadcast request. Multicast search sends out UDP packets (to address 224.0.1.84, port 4160) [12,9]. In Jini, this is done with LookupDiscovery class, registering an application-defined listener with the DiscoveryListener object. When receiving the search request, every LookupService in the network that attends the requested group, will reply back and the framework will call the discovered() method of DiscoveryListener. Thus, application will have called its discovered() method for each LookupService in interest. Service providers could register their services to all LookupServices, and clients can query them for services they need:

The process involves also copying the remote ServiceRegistrar object locally to the client [12,9]. Jini internally ships compiled class-files from LookupService, and makes them alive through serialization. To transfer data, Jini uses a TCP socket connection. The local object, called Registrar, runs as a java-object in the application's address space, and the application makes normal method calls to it. When needed, it communicates back to its lookup service. For application it acts like a proxy to the LookupService – it does not cache any information on the application side, but gets „live"' information from the lookup service as needed. The Registrar is used by service providers to register their services, as well as by clients to discover services. When the service is registered, its provider passes the service object to the lookup service. Often, this object is another proxy object that communicates back to the service provider, but it can also be a stand-alone object that implements the whole service. The service object gets cached in Lookup Service's repository, together with additional service attributes. When a client requests this service, the Lookup Service sends back the service object to it. Having this object, the client can execute methods on it, and when it is a proxy object, it will delegate these invocations to the service provider using some remote invocation scheme(usually JavaRMI). Services are registered for a specified time period, called Lease, and they regularly update the lease to show they are alive. When the lease time runs out, the service is automatically unregistered. This allows safe removal of services whose provider crashed. The concept is depicted below in *figure 5*.

Every service has its own Service ID, which is a „universally unique identifier", and it is unique over time and space with respect to all other service IDs generated by all lookup services. Every newly registered service obtains its own Service ID. This could be used by the client and LookupService to quickly discover the service in need. Alternatively, a client may want to find a service satisfying a number of interface requirements at once. For example, a client may look for a service that implements both Clock and Alarm. Finally, the client can specify a set of attributes that must be satisfied by each service. Each attribute required by the client is taken in turn and matched against the set offered by the service. For example, in addition to requesting a Clock with an Alarm, a client entry may specify a specific time format. All this information is stored in an object of class ServiceTemplate, and sent to the LookupService. On the other hand, when services are registered the provider sends all information about the service in ServiceItem object. These objects are what LookupServices store about each service it knows of.

For each registered service the LookupService stores an instance of a service class along with a set of attribute entries. A client can search for suitable services either by asking for an instance of a specific class, or by asking for an instance of more general class with the additional information that it can support particular characteristics and attribute values.

The Entry class allows services to advertise their capabilities in flexible and simple ways. Their primary intention is to provide extra information about services so that clients can decide whether or not they are the services they want to use [9]. For example, suppose a printer capable of handling a number

Simple Java code illustrating multicast LookupService search for all groups:

```
try {
   LookupDiscovery discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);

   discover.addDiscoveryListener(new DiscoveryListener() {
      public void discovered(DiscoveryEvent evt) { // LookupService discovered

         ServiceRegistrar[] registrars = evt.getRegistrars();

         for(ServiceRegistrar registrar : registrars) {
            System.out.println("Found registrar at " + registrar.getLocator().getHost());
   }
  }
  public void discarded(DiscoveryEvent evt) { // LookupService discarded

  }
});
}
catch(Exception e ... // handle exception...
```

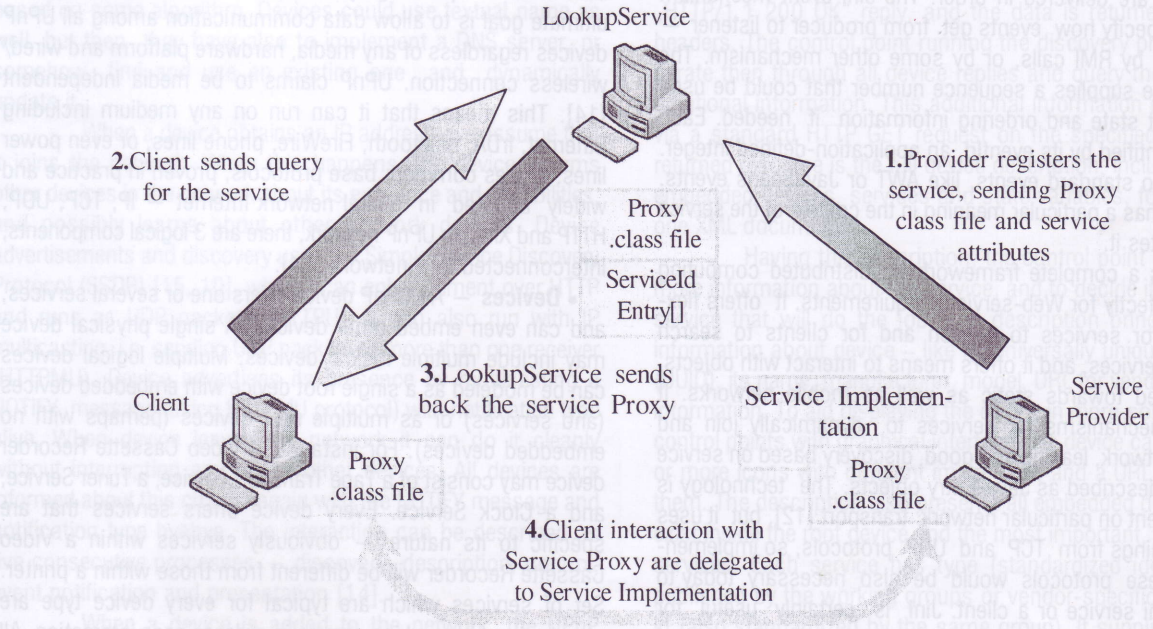


Figure 5. Interaction between Client, Service Provider and Lookup Service in a Jini system

of input file formats. It could do so by exporting a service object implementing Printer along with an Entry object saying that it can handle plain text and another Entry object saying that it can handle PostScript files. The service implementation can just add more and more information about its capabilities without altering the basic interface. Every object that is exported as an attribute, has to implement the Entry interface. This empty interface is used only to distinguish the Entry objects. When a client wishes to find services that can handle particular attributes it uses Entry class to specify which fields it needs. For any particular Entry, the client specifies which fields must match exactly, and which fields it does not care about [12]. As a simplification, client is not allowed to perform quantative checks, i.e. client cannot search for printer whose speed is at least 5 pages per minute, nor any Boolean expressions. The client can only specify particular values to some of the fields and to ignore values of other fields. If a field has value of null, it will be ignored in matching. Thus primitive types (such as int or char) cannot be used as fields - they have to be wrapped by their corresponding Java-class (like Integer or Character). Matching services in repository to the requested service template is a subject to the following rules [12]:

- A service item (item) matches a service template (tmpl) if: item.serviceID equals tmpl.serviceID (or if tmpl.serviceID is null); and item.service is an instance of every type in tmpl.serviceTypes; and item.attributeSets contains at least one matching entry for each entry template in tmpl.attributeSet.
- An entry matches an entry template if the class of the template is the same as, or a superclass of, the class of the entry, and each non-null field in the template equals the corre-

sponding field of the entry. Each entry can be used to match more than one template. Note that in a service template, for serviceTypes and attributeSet, a null field is equivalent to an empty array; both represent a wildcard.

Jini objects may also be interested in changes in other Jini objects, and would like to be listeners for such changes. Jini supports remote events, but mostly the implementation is up to Java-programmer, as Jini offers only the concept [12,9,10]. The remove events architecture takes into account the nature of network communication and handles possible problems like unreliability of network delivery, time required to transfer an event message, and listeners could leave the network without removal from object they listen to (for example in case of service crash), so they are allowed time out like service's lease mechanism. A remote event is serializable object and can be moved around the network to its listeners. Jini makes no assumptions about guarantees of delivery, and does not even assume that events are delivered in order. The Jini event mechanism does not specify how events get from producer to listener – it could be by RMI calls, or by some other mechanism. The event source supplies a sequence number that could be used to construct state and ordering information if needed. Each event is identified by its eventId, an application-defined integer. There are no standard events, like AWT or JavaBeans events. Each event has a particular meaning in the context of the service that generates it.

Jini is a complete framework for distributed computing that fits perfectly for Web-services requirements. It offers flexible ways for services to publish and for clients to search published services, and it offers means to interact with objects. It is targeted towards static as well as mobile networks. It supports mechanisms for services to dynamically join and leave the network, leasing and good discovery based on service attributes, described as Java-Entry objects. The technology is not dependent on particular network transport [12], but it uses not a few things from TCP and UDP protocols, so implementation of these protocols would be also necessary today to create a Jini service or a client. Jini is certainly useful for the home environment, as long as interfaces for the desired devices are being developed. This is one of the drawbacks: much is still left unspecified, interfaces for certain devices still have to be implemented, while they are already available for consumer electronics in other service discovery solutions [9]. Because of Java language used in, every device or computer that wants to use Jini, needs to have a Java Virtual Machine, making it hard for embedded and poor-resource systems. However, currently more and more embedded and mobile devices are equipped with Java Micro-Edition, so maybe in future Jini will take part not only in conventional networks but also in mobile and ad-hoc networks.

6. Universal Plug and Play

Universal Plug and Play (UPnP) is more than just a simple extension of Plug and Play technology [14,10]. It is designed to support a zero-configuration, invisible networking, and an automatic discovery for a variety of devices. The scope of UPnP encompasses many existing, as well as new and exciting scenarios including home automation, printing and imaging, audio and video entertainment, kitchen appliances and others [14]. The supported devices are classified into standard categories, and special groups are working towards complete standardization for wide range of vendors. The goal is to enable the emergence of easily connected devices and to simplify the implementation of networks in home and corporate environments, by defining and publishing UPnP device and service descriptions built on open, Internet-based communication standards. Since UPnP is distributed, open network architecture, defined by the protocols used, it is independent of any particular operating system, programming language, or physical medium [14,9]. UPnP does not specify the APIs applications will use, allowing operating system vendors to create the APIs that will meet their customer needs. Currently, more and more devices (mostly routers) are shipped with UPnP support, and support is built in some operating systems, like Windows XP [17]. The ultimate goal is to allow data communication among all UPnP devices regardless of any media, hardware platform and wired/wireless connection. UPnP claims to be media independent [14]. This means that it can run on any medium including Ethernet, IrDA, Bluetooth, FireWire, phone lines, or even power lines. It uses common base protocols, proven in practice and widely adopted in global network Internet – IP, TCP, UDP, HTTP and XML. In UPnP network, there are 3 logical components, interconnected in a network [14]:

- **Devices** – An UPnP device offers one or several services, and can even embed other devices. A single physical device may include multiple logical devices. Multiple logical devices can be modeled as a single root device with embedded devices (and services) or as multiple root devices (perhaps with no embedded devices). For instance, a Video Cassette Recorder device may consist of a Tape Transport Service, a Tuner Service, and a Clock Service. Every device offers services that are specific to its nature – obviously services within a Video Cassette Recorder will be different from those within a printer. Set of services which are typical for every device type are standardized by working groups in order to ease interaction. All information is described in a XML device description document for each device.
- **Services**. A service offers some functionality over the network. It exposes it as actions and models its state with state variables. For instance, a clock service could expose the action SetTime and have a state variable called CurrentTime. Using the actions clients can control the service, or query it for information. Services are contained within devices, and one device may contain as many services as necessary.
- **Control Points**. A control point in an UPnP network is a component that discovers and controls devices. It can query devices for services in interest, invoke actions on services, and subscribe to service's events. Anytime the state of the service changes, an event will be sent to the control point. Normally devices in the network will act also as control points, thus enabling true peer-to-peer networking.

A service in an UPnP device consists of a state table, a control server and an event server. The state table models the state of the service through state variables and updates them

when the state changes. The control server receives action requests (such as SetTime), executes them, updates the state table and returns responses. The event server publishes events to the interested subscribers whenever the state of the service changes. This architecture allows all control points to be informed about the state of any device in the network, and at the same time to keep low network traffic.

Although UPnP claims to be completely media independent, the addressing scheme is based on IP addresses and TCP/UDP [9, 14]. Other devices connected in different networks like HAVi, CAN or X10 can also participate in the UPnP network through an UPnP bridge or proxy [14]. Each device in UPnP network is identified by unique IP address. When a device joins the network, it first looks for a DHCP server using IP multicast (this automatically realizes the necessity of DHCP client implementation in each device). For managed networks where a DHCP server exists, it will assign IP address to the device. Otherwise, the device chooses IP address randomly based on some algorithm. Devices could use textual name as well, but then they have also to implement a DNS server, or somehow find and use an existing one and dynamically update it.

When a device obtains an IP address, we assume that it joins the network. When this happens, the device informs other devices in the network about its existence and capabilities, and possibly learns about other network devices. Device advertisements and discovery uses the Simple Service Discovery Protocol (SSDP) [15, 10], which is an enhancement over HTTP and runs as UDP packets (HTTPU). It can also run with IP multicasting, i.e. sending UDP packets to more than one receiver (HTTPMU). Device advertises its presence by multicasting a NOTIFY message (using HTTPMU protocol) with notification type alive. When device leaves the network it can do it cleanly without interrupting any of the other devices. All devices are informed about this change again with the NOTIFY message and notification type byebye. The interaction can be described as five consecutive processes – discovery, description, control, event notification and presentation [14].

When a device is added to the network, the UPnP discovery protocol allows that device to advertise its services to control points on the network. Similarly, when a control point is added to the network, the UPnP discovery protocol allows that control point to search for devices of interest on the network. The fundamental exchange in both cases is a discovery message containing a few, essential specifics about the device or one of its services – its type, universal service identifier, and a URL to more detailed information.

Service discovery in UPnP does not need a central repository or lookup service – service query is multicasted in the network, and all participants reply back when they meet the query criteria. This is a two-step process – first all devices are multicast searching for an abstract service type. Devices and services are discovered by using SSDP's M-SEARCH messages over HTTPMU [14,15]. At this stage, the control point can refine its criteria to one of the following:

- Search for all device and services (search target is „ssdp:all").
- Search for root-level devices only, and does not search for embedded devices or services (search target is „upnp:rootdevice").
- Search for a particular device, knowing its universal ID (UUID). Device UUID is standardized and is specified by UPnP vendor (search target is „uuid:device-UUID").
- Search for any device of a particular type. Device types are standard and are defined by the UPnP Forum working committee (search target is „urn:schemas-upnp-org: device:device Type: version").
- Search for any service of a particular type. Service types are standard and are defined by UPnP Forum working committee (search target is „urn: schemas-upnp-org:service:service Type: version").

Then, after receiving the IP multicast, all matching devices return back small piece of information about themselves, like type, ID and URL for more information. At this stage they do not return more information, like attributes, name, provider and capabilities. The information is returned with standard „HTTP/1.1 200 OK" reply, and the data is returned in HTTP headers. The control point running the discovery process can iterate then through all device replies and query their URL for additional information. This additional information is retrieved via a standard HTTP GET request on the specified URL. The returned resource is the full device description, including all its embedded devices, services and state variables, formatted in one XML document.

Having this description, the control point can obtain more information about the service, and to decide if this is the service that will do the job. The description brings detailed information about device – like its universally unique identifier (UUID), its name, manufacturer, model, URL for vendor-specific information. To aid displaying the device in more-sophisticated control points with graphical interface, the device can offer one or more icons with different image size, and a URL to retrieve them. The description also presents all embedded devices that exist inside the root device and the most important, the offered services. Each service has type (standardized identifier, as published by the working groups or vendor-specific type) and id (also standardized by the same group). It supplies 3 URLs – Service Control Protocol Definition, control and event URLs. The first refers to another XML file with description of actions offered by the service along with description of arguments they take – a name, a meaning and a type, as well as the state variables that this service has. The second URL can be used to control the service by activating its actions. For remote calls UPnP uses the SOAP protocol – all actions with their parameters and return value are executed with messages formatted according SOAP specification [8].

Note that the control point can request service description XML, but for standard services it is not obligatory to do it. Since all actions, services and devices are standardized, the control point can assume that some actions for particular device types are present, as well as it can know actions' parameters and their return type. Thus it is not necessary to add additional network traffic for standard service descriptions [14].

Each control call, performed on a service, can change its internal state. This internal state can also be changed during its work, due to the external circumstances or service's algorithm. In UPnP, the state is modeled with non-empty set of state

variables [14]. Each variable has a name and a data type. The change of variable's value is considered as an event and can be signaled to all control points that are interested. State variables like these are evented. Particular state variables can also be specified as non-evented, for example one whose value changes too rapidly, or that contain a value too big for eventing. To determine the current value for such non-evented variables, control points must poll explicitly the service. Each control point can subscribe for changes in state variables. A subscriber is sent all event messages from the service, and all subscribers receive equal information about service's status change. Event messages are XML messages according the GENA standard [16], sent with IP unicast to all subscribers with HTTPU protocol.

Service control via SOAP messages [8] is not the only way to control the service. For humans, UPnP offers additional mechanism for visualization and control of the service – the presentation URL [14]. This URL refers to HTML page that shows service's status and allows basic or complete control over it. This allows a user with standard Web browser to observe and supervise the service. The specification does not say how this HTML page will look like and what possibilities it will offer – this is up to the device manufacturer.

Universal Plug and Play is a technology promising that can make true pervasive networking in near future. It has all

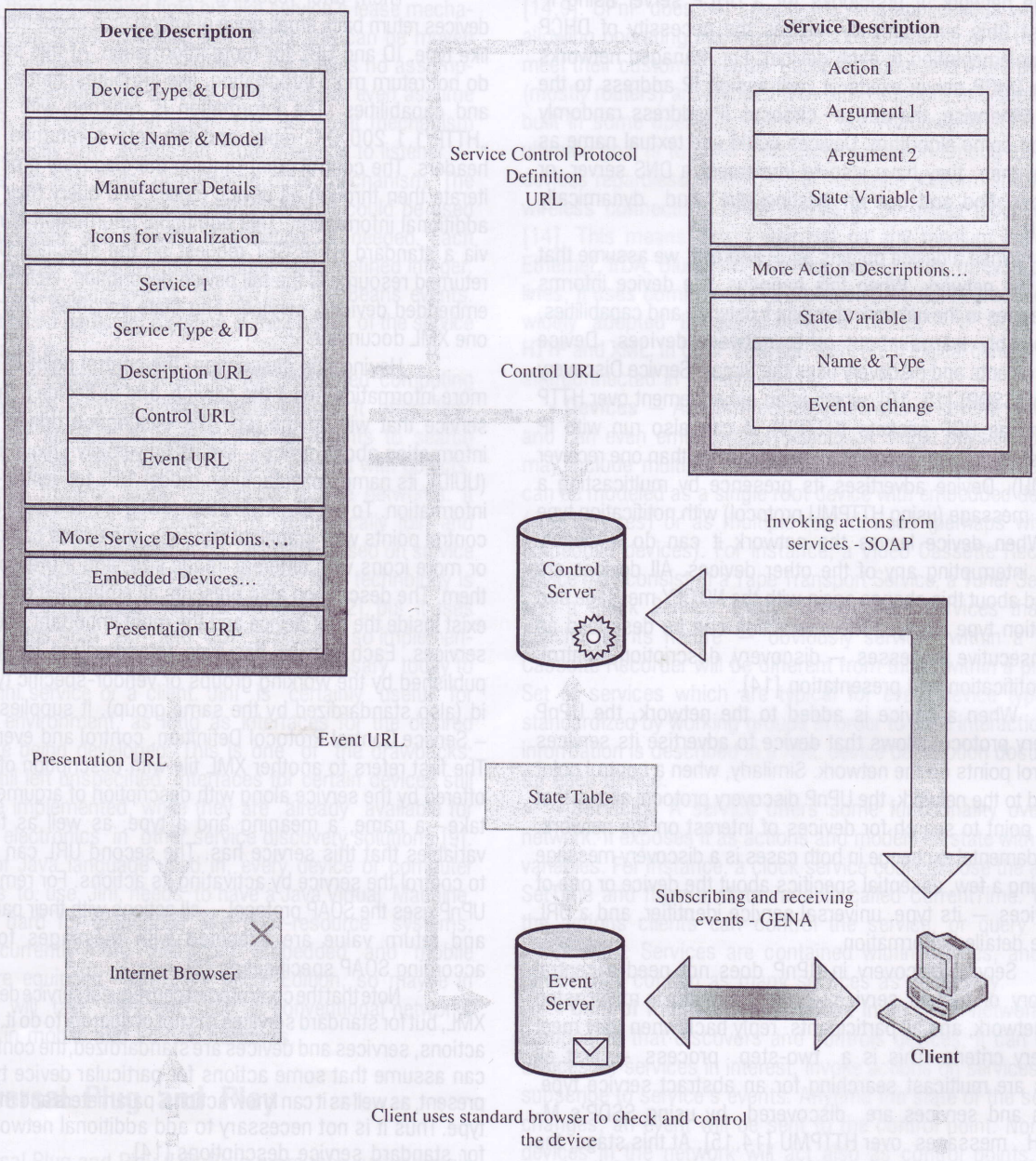


Figure 6. Discovery, Description, Control, Eventing and Presentation in UPnP

that is required to achieve its goal. Foundation on standard and proven protocols helps UPnP to extent easily [14]. An important feature of UPnP is automatic configuration of IP addresses of devices being plugged in. UPnP can work with no central directory of addresses or service repository. UPnP comes with predefined standard templates for existing home and office appliances and hence it takes a high position among the service discovery solutions in home environments [9,11,10]. But among all these advantages UPnP has two major problems – first is the requirement for security [11], and the second is the necessity for implementation of all mandatory technologies – HTTP server, DHCP client, TCP/IP, HTTPU and HTTPDU, XML parsers and others [14]. This is surely easy for desktop computers, but it may be a problem for embedded devices with limited resources. Nevertheless, UPnP is a promising technology that surely will be a player in service discovery.

7. Bluetooth

Bluetooth is a wireless, short-range ad-hoc network, offering instant connectivity between different devices. It uses the globally available 2.4-GHz ISM (industrial, scientific, and medical) frequency band and targets mobile devices as well as desktop computers and periphery. The operating range is typically around 10 meters (but there are also wider standards), and Bluetooth offers smaller price compared with other wireless technologies because of the integrated single-chip radio [18]. Mobile devices usually have limited resources, so Bluetooth protocol stack is designed to be as simple as possible according device resources and capabilities. It includes radio frequency protocol (RF), the Baseband protocol, Link Manager Protocol, a Host Controller Interface, Logical Link Control and Adaptation Protocol, RFCOMM (for cable replacement) and finally Service Discovery Protocol (SDP).

Bluetooth supports two types of connections – point-to-point connections and piconets [18]. When a device initiates communication with other devices, it can form a new piconet. Each piconet has a master device (the one that initiated the communication) and up to seven slave devices. Any slave device communicates always with the master device, and never with another slave device. Several piconets can occupy a common physical area without mutual disturbing - this is achieved through a dynamic change of the radio frequency in small steps, called hopping. A single device can participate in more than one piconet, and thus forming networking structures called scatternets, but it can be master device in maximum one piconet. All piconets inside a scatternet can work independently and simultaneously. Standard mechanisms are provided for a device to change dynamically its role as a master or a slave.

Bluetooth specification offers standard approach for discovering services [18,19]. It is called Bluetooth SDP and is optimized for ad-hoc networks and resource-constrained devices. It is a simple protocol that allows client applications to discover the existence of services provided by server applications in the vicinity of the user, as well as the attributes of those services. The attributes of a service include the type or class of service offered and the mechanism or protocol information needed to utilize the service. Having located available services, a user or client may choose to use any of them.

SDP uses a request/response model where each transaction consists of one request message (called Protocol Data Unit, PDU) and one response PDU. The specification defines standard type of PDUs – each one has a header with PDU and transaction identifiers, and a variable-size payload part with parameters for a request or response. Generally, each type of request PDU has a corresponding type of response PDU, but if server finds that the request is not formatted as it is expected to be, it can respond with an error PDU [19].

Each service in terms of the Bluetooth SDP has a 128-bit universally unique identifier (UUID) that defines the service functionality. Each service can also be associated with one or more attributes. Each service attribute describes a single characteristic of a service. A service is always an instance of particular service class. The service class is standardized and

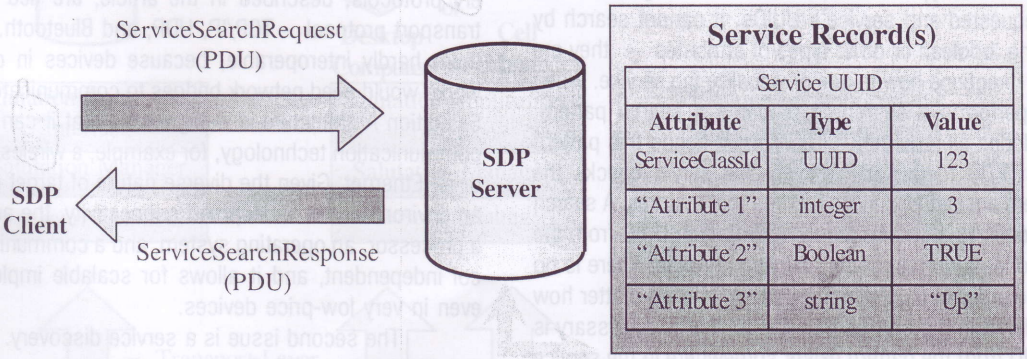


Figure 7. Service Discovery in Bluetooth SDP

describes the attributes that the service possesses. A service class can be subclassed and the derived class inherits all attributes from the base class, as well as new specific attributes. A service can be an instance of more than one service class as well. Attributes can be standard ones, common for all services, or specific to the concrete service. In either case, every attribute has its own UUID, and additionally name and data type. For standard or universal attributes, the data type and usage directions are defined and standardized. Data types for all attributes can be null value, unsigned integer, signed integer, UUID, string, boolean value, URL, etc.

Each service contains at least one attribute - the service class attribute. This attribute is mandatory and presents a list of all service classes UUIDs that this service is instance of. The attribute is called ServiceClassIDList and the UUIDs are ordered from most particular to the most common class. Service discovery in a piconet is based on SDP servers and SDP clients. It is decentralized, meaning that no central repository or service locator is used – the job is done by SDP servers and clients. These are software components that communicate in between and carry out the discovery process. A SDP client sends a discovery query to SDP server, and the server replies back to it. Every SDP server maintains a directory with all services offered by the device that owns it, and is responsible for requests reply. Each Bluetooth device has at most one SDP server, but can act as a SDP client to many servers. All of the information about a service is maintained by an SDP server within a single service record. The service record consists entirely of a service record handle and a list of service attributes. SDP allows the following set of service inquires:

- Search for services by service class – a SDP client supplies the service UUID.
- Search for services by service attributes – a SDP client supplies the service UUID as well as a set of attribute UUID that should match the service description.
- Service browsing – SDP client can browse for all services in a particular group.

The process of service discovering uses its own channel in a point-to-point communication, and allows the client to narrow its requirement by specifying different attributes that the service should contain. SDP supports searching only by attributes whose data type is UUID, and the searching uses exact match of requested and service's UUIDs. It cannot search by integer, string, boolean or other types of attributes – they are intended for directions how to use or visualize the service. When SDP client performs a search, it prepares a search pattern, where it includes all required UUIDs. Then it sends this pattern in a request PDU to the SDP server. The server checks the search pattern against all service records that it stores. A search pattern matches to a service record only if each UUID from the pattern matches a corresponding service attribute. There is no matter of the order of UUIDs in the pattern, and no matter how they are ordered in the service record - all that is necessary is that any UUID from the pattern exists somewhere in the service record's attributes. Each search pattern contains at least one UUID and this could be the UUID of the service class (recall the ServiceClassIDList attribute).

Bluetooth specification allows each service to attend to one or more logical groups. Groups can be hierarchically organized, and each one has its own UUID. Each group is described by a special service of class ServiceGroupDescriptior with particular value of its group identifier. When another service wants to attend to a group, it sets its BrowseGroupList attribute to the identifier or identifiers to the group. Thus, browsing for all services in particular group gets easy – the SDP client just needs to include only the group's UUID in the search pattern. As a result it will receive responses from all devices that attend the group (because all they have the specified group UUID in their BrowseGroupList attribute). Generally, all services attend the common group called PublicBrowseRoot, and thus any other device can browse them.

Bluetooth is not a simple technology, but it uses a simple service discovery protocol [19]. It does not offer any means to notify other network participants about new device that joins the network, or device that leaves it. There is no event notification when services become unavailable. In a dynamic ad-hoc network, it is expected that devices often join and leave the network, so the lack of this mechanism is a serious disadvantage. Despite of this, Bluetooth SDP offers implementation-simple and at the same time quite powerful discovery mechanism, suited for mobile and resource-constrained devices. To compensate the lack of means to access services, notifications, service advertisement and registration one can use other service discovery protocol above Bluetooth SDP, like Salutation [10].

8. Salutation

The Salutation Architecture aims at solving the problems of interoperability between a broad set of devicesand equipment and in an environment of widespread connectivity and mobility [20]. It provides a standardmethod for applications, services and devices to describe and to advertise their capabilities to other applications and devices, to find out their capabilities. The architecture takes in mind that customers' networks are not made up of a single manufacturer's pieces of equipment, and all they do not work in the same manner. Thus, Salutation tries to offer a single, versatile protocol for service discovery and utilization [20].

The first issue is the network diversity. All service discovery protocols, described in the article, are tied to particular transport protocol –TCP/IP, UDP, and Bluetooth. This makes them hardly interoperable, because devices in different networks would need network bridges to communicate in between. Salutation Architecture is designed so that it can sit atop any communication technology, for example, a wireless connection or an Ethernet. Given the diverse nature of target equipment in an environment of widespread connectivity, the architecture is a processor, an operating system, and a communication protocol independent, and it allows for scalable implementations, even in very low-price devices.

The second issue is a service discovery. Even though Jini, Bluetooth and UPnP all have a discovery protocol, the three protocols are incompatible. A Jini device cannot communicate with a device that uses only the Bluetooth service discovery protocol. However, the Salutation Discovery Protocol offers com-

mon means and mechanisms where devices can use same protocol, and they would at least be able to determine the capabilities of the other devices in the network. Additionally, since same software could be used with a Java- or Bluetooth-enabled device, or one using some other network protocol, Salutation may be the most cost-effective way to develop applications for any networked device. The architecture enables applications, services and devices to search services or devices for a particular capability, and to request and establish connections and use their capabilities.

The software that manages the Salutation protocol is called the Salutation manager [20,9,10]. It offers a set of APIs for client applications and acts as a service registry, discovers what services are on the network and whether they are available, and manages the session. It uses common transport manager, responsible for network communication. The Salutation Manager contains Registry to hold infor–mation about Services. Optionally, the Salutation Manager Registry may store information about Services that are registered in other Salutation Managers. All requests by other equipment for Salutation resources would be directed toward other Salutation Managers, which would respond accordingly. The Salutation Manager can discover other remote Salutation Managers and can determine the Services registered there. Service Discovery is performed by comparing the required service types, as specified by the local Salutation Manager, with the service types available on a remote Salutation Manager.

A device can use the Salutation discovery protocol to ask other devices on its network about their capabilities. The inquiry passes from the Salutation manager to the transport manager, which prepares the inquiry to run over the transport protocol used by the network. It makes its way over the network to another Salutation-compatible equipment, which supplies the information, and also learns about the initiating device. This combination of communicating devices could also be connected to a different device by a second network running a different transport protocol. The device in the middle needs two transport managers, one for each network protocol, but only one Salutation protocol. The Salutation application interface offers standard API and allows service consumers to interact with the Salutation protocol. Salutation Architecture allows software developers to write one application that works with several existing Salutation environments – primarily in office automation equipment. It could be the bridge between Bluetooth and Jini devices, as well as to other platforms and networks. Salutation is non-proprietary and thus the specifications open and available for third party development and since Salutation chooses a middle way between autonomy (Jini) and standardization (UPnP), it is easy for vendors to adapt to the specifications [9].

9. Comparison and Conclusion

The paper described some of the existing technologies for service discovery. An obvious conclusion is that there are far too many standards at this time. Many protocols are well suited for a range of problems, but are weak in other areas. Currently there is no universal protocol fitting perfectly into all the requirements like acceptable resource demands, network traffic, and technology completeness. Common comparison of described protocols is shown below (Salutation is not shown in comparison because it is not examined independently):

A key architectural issue is the way in which the information about existing services is stored in the network. Discovery protocols can be separated into two major groups – one with central Directory (Repository) and another where devices search for services by broadcasting messages into the network. The directory itself can be mobile or static. Protocols like DNS-SD, SLP, UDDI and JINI maintain a centralized directory that is able to serve any client with information they need. An example of mobile repositories is Directory Agents in SLP [4] and Lookup Service in JINI [13]. DNS-SD and UDDI on the other hand, have static repositories [6]. Others, like UPnP and Bluetooth,

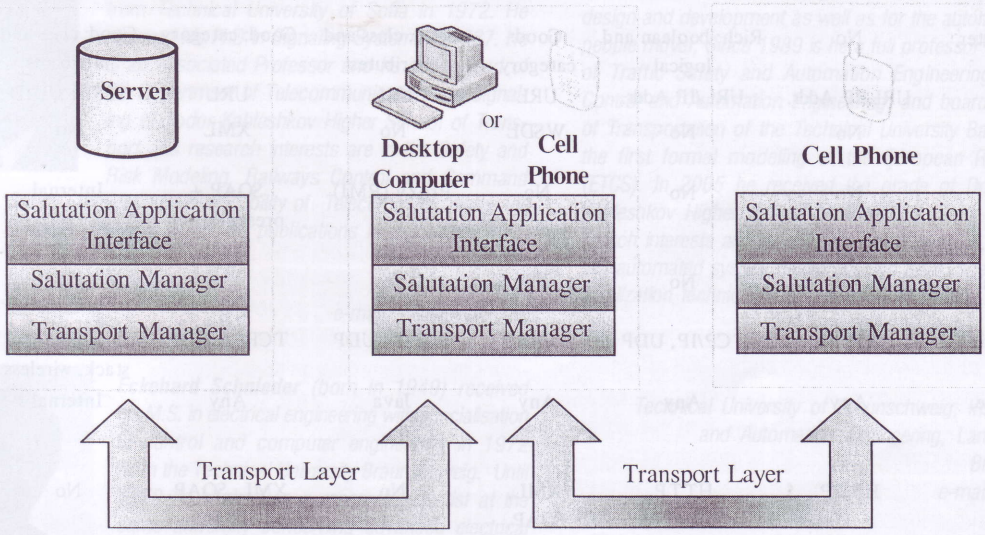


Figure 8. Salutation in action

use network broadcasting and rely that every suitable service will reply when a request is broadcasted into the network. This solution gives a great level of dynamism for services, and allows flexible and automatic service/client configuration. Maintaining state and discovery requests may involve quite a big traffic in the network for the later case. On the other hand, the lapse of centralized directory makes the network more robust and secured against directory service failures.

The protocols presented offer means how to find and interact with particular service. Most of discovery protocols examined in the paper are not fully complete. Some of them, like DNS-SD, SLP and Bluetooth measure up to retrieval of service address, and do not even specify how to invoke and perform an action over the service. UDDI and UPnP go further and offer description via WSDL or XML [6,14]. JINI probably is the most complete technology from above, because it realizes the remote invocation mechanism by a standard programmer's framework for transparent service activation through proxies [13].

To ease the interoperation between devices, a programming framework is highly recommended. It should offer an API for discovery, addressing and transparent activation of services. Having standard interface would allow services to register themselves and to be invoked by clients by minimum effort both for the service supplier and for service client. Also, the underlying details of communication, if it is SOAP, XML, WSDL or other will not be so important for applications. The framework can use the best one, fitted to its requirements, or even use several possibilities to enable bridging between different protocols. DNS-SD, SLP, UDDI, JINI, UPnP and Bluetooth protocols are independent between each other and cannot work in cooperation. Most of them can work together with others inside the same physical network without mutual disturbance, but do not offer interconnection. Many of these protocols are logically

Property	DNS-SD	SLP	UDDI	JINI	UPnP	Bluetooth
Discovery Model	Centralized (DNS-Server)	Centralized (Directory Agent), possible Peer-to-Peer	Centralized	Centralized, Lookup Service	Peer-to-Peer, Search by Multicast	Peer-to-Peer
Mobility	Static supplier	Yes	Static directory	.Yes	Yes	Yes
Service auto-configuration	No	No	No	No	Yes	Yes
Service Grouping	By subnet	Yes (scopes)	No	Yes	No	Yes, hierarchical
Service Registration	Manually by administrator	Auto-advertise	Auto-advertise Web-interface	Auto-advertise	Auto-advertise	Auto-advertise
Service Browsing	Yes	Yes	No	Yes	Yes	Yes
Search and Filter	No	Rich: boolean and logical	Good: category	Good: class and attributes	Good: category	Good: class and attributes
Addressing	URL/IP Addr.	URL/IP Addr.	URL	URL	URL	128bit UUID
Activation Description	No	No	WSDL	No	XML	No
Activation, Framework	No	No	No	Proxy (RMI)	SOAP + presentation URL	Internal
Remove Events	No	No	No	Yes	Yes	No
Network/Media	TCP/IP	TCP/IP, UDP	TCP/IP	TCP/IP,UDP	TCP/IP,UDP	Own protocol stack, wireless
Progr.language, platform	Any	Any	Any	Java	Any	Internal
Additional technologies	HTTP	HTTP	XML, SOAP, HTTP	No	XML, SOAP, HTTP	No
In use today	Rarely	Yes	Rarely	Yes	Yes	Yes

compatible [11], and can be mapped and bridged between with additional tools and services. Another possibility is to use hybrid solutions as a combination of more than one technology. An example could be the simplicity of Bluetooth's discovery with another richer discovery, like Salutation [10]. Actually, discovery really should be universal, and it should not be necessary to implement an array of equivalent protocols, or to have multi-protocol proxies [11]. Devices participating in ambient networks not always can offer enough computing power and storage to implement heavy and complex discovery protocols, as well as containing a code to support different protocols and mappings between them.

References

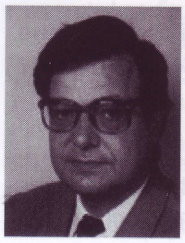
1. DNS-Based Service Discovery. http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt.
2. DNS SRV (RFC 2782) Service Types. http://www.ietf.org/rfc/rfc2782.txt.
3. Multicast DNS. http://files.multicastdns.org/draft-cheshire-dnsext multicastdns.txt.
4. RFC 2608. Service Location Protocol, Version 2,http://www.ietf.org/rfc/rfc2608.txt.
5. RFC 2609. Service Templates and Service:Schemes.http://www.ietf.org/rfc/rfc2609.txt.
6. OASIS. UDDI Version 2 Specifications. http://www.uddi.org/specification.html.
7. Piergiorgio Cremonese. Veronica Vanni. UDDI4m:UDDI in Mobile Ad Hoc Network. Second Annual Conference on Wireless On-demand Network Systems and Services (WONS'05).
8. W3C. Soap Version 1.2. http://www.w3c.org/TR/soap.
9. Stefan Fischer.Service Discovery in Home Environments. http://www.ibr.cs.tu-bs.de/courses/ws0203/skm/articles/hsd5.pdf.
10.Choonhwa, Lee, Sumi Helal.Protocols For Service Discovery In Dynamic And Mobile Networks.
11.McGrath, Robert E.Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing.
12. SUN Microsystems. Jini Architecture Specification. http://www.sun.com/jini.
13. Jan Newmarch. Jan Newmarch's Guide to Jini Technologies. http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml.
14. UPnP Forum. UPnP Specification. http://www.upnp.org.
15. UPnP Forum. Simple Service Discovery Protocol (SSDP). http://www.upnp.org/resources/specifications.asp.
16. UPnP Forum, General Event Notification Architecture (GENA). http://www.upnp.org/resources/specifications.asp.
17. Stephen J. Bigelow. Universal Plug and Play: Networking Made Easy. http://xml.coverpages.org xmlPapers200308.html# BigelowUPnP.
18. Bluetooth SIG. Specification of the Bluetooth System. Volume I:Core Specification. http://www.bluetooth.com/dev/specifications.asp.
19. Eugene A. Gryazin. |Service Discovery in Bluetooth. http://www.cs.hut.fi/~gryazin/SD_in_Bluetooth.pdf.
20. Salutation Consortium. http://www.salutation.org.
21. Peer Hasselmeyer , On Service Discovery Process Types. http://www.hasselmeyer.com/pdf.isco 05.pdf.

Manuscript received on 25.07.2006

Eng. Nikolay Kalaydjiev *was born in 1979. He graduated the Technical University in 2002 as a computer engineer. Since 2004 he is a PhD student at the Institute of Computer and Communication Systems at the Bulgarian Academy of Sciences. His scientific interests include computer networks, communication, addressing and discovery of services in ambient networks in pervasive and embedded computing environments.*

Contacts:
nik@iccs.bas.bg

continuation from 30



Emil Ivanov *was born in Sofia in 1949. He received the MSc degree in Electrical Engineering from Technical University of Sofia in 1972. He received his PhD in Signaling Systems in 1987. He is an Associated Professor and he is the Head of the Department of Telecommunication and Signaling of Todor Kableshkov Higher School of Transport. His research interests are in the Safety and Risk Modeling, Railways Control and Command Systems, Safety Certification, Dependability of Telecommunications and Signaling Systems. He has about 80 publications in this area.*

Contacts:
e-mail: eivanov@vtu.bg

Eckehard Schnieder *(born in 1949) received his M.S. in electrical engineering with specialisation in control and computer engineering in 1972 from the Technical University Braunschweig. Until 1979 he worked as a research scientist at the same university concerning advanced electrical drive control systems simulation. He received his Ph.D. in 1978. From 1979 until 1989 he joined Siemens Transportation Systems in Braunschweig, where he was responsible for the German maglev TRANSRAPID operation control system's design and development as well as for the automatic control of Siemens people mover. Since 1989 is he a full professor and head of the Institute of Traffic Safety and Automation Engineering, formerly Institute of Control and Automation Engineering, and board member of the Centre of Transportation of the Technical University Baunschweig. He directed the first formal modeling of the European Railway Control System (ETCS). In 2005 he received the grade of Dr. h. c. from the Todor Kableshkov Higher School of Transport, Sofia, Bulgaria. His main research interests are the discrete event systems, control system synthesis, automated system design, design tools, operations control systems, localization techniques, e.g. satellite navigation.*

Contacts:
Technical University of Braunschweig, Institute for Traffic Safety and Automation Engineering, Langer Kamp 8, D-38106 Braunschweig, Germany,
e-mail: e.schnieder@tu-bs.de