1 Mobile, Distributed, and Pervasive Computing

Michel Barbeau

Abstract

Pervasive computing can be defined as access to information and software applications anytime and anywhere. This form of computing is highly dynamic and disaggregated. Users are mobile and services are provided by collections of distributed components collaborating together. Recent advances in mobile computing, service discovery, and distributed computing are key technologies to support pervasive computing.

This chapter is about software technologies designed to address problems in mobile, distributed, and pervasive computing. Characteristics of pervasive computing applications are reviewed. Architecture of pervasive computing software is discussed. Key open communication technologies to support pervasive computing are presented in detail, namely, service discovery and distributed computing.

1.1 INTRODUCTION

Pervasive computing aims at availability and invisibility. On the one hand, pervasive computing can be defined as availability of software applications and information anywhere and anytime. On the other hand, pervasive computing also means that computers are hidden in numerous so-called information appliances which we use in our day-to-day life [Bir97], [Wei91], [Wei93]. Personal Digital Assistants (PDAs) and cell phones are the first widely available and used pervasive computing devices. Next generation devices are being designed. Several of them will be portable and even wearable, such as glass embedded displays, watch PDAs, and ring mouses.

Several pervasive computing devices and users are wireless and mobile. Devices and applications are continuously running and always available. From an architectural point of view, applications are non-monolithic, but rather made of collaborating parts spread over the network nodes. These parts are hereafter called distributed components. As devices and users move from one location to another, applications must adapt themselves to new environments. Applications must be able to discover services offered by distributed components in new environments and dynamically

reconfigure themselves to use these new service providers. From a more general point of view, pervasive computing applications are often *interaction transparent*, *context aware*, and *experience capture and reuse capable*. Interaction transparency means that the human user is not aware that there is a computer embedded in the tool or device that he or she is using. Context awareness means that the application knows, for instance, the current geographical location. An experience capture and reuse capable application can remember when, where, and why something was done and can use that information as input to solve new tasks.

Pervasive computing is characterized by a high degree of heterogeneity: devices and distributed components are from different vendors and sources. Support of mobility and distribution in such a context requires open distributed computing architectures and open protocols. Openness means that specifications of architectures and protocols are public documents developed by neutral organizations. Key specifications are required to handle mobility, service discovery, and distributed computing.

In this chapter, we review the main characteristics of applications of pervasive computing in Section 1.2, discuss the architecture of pervasive computing software in Section 1.3, and review key open protocols in Section 1.4.

1.2 PERVASIVE COMPUTING APPLICATIONS

Characteristics of pervasive computing applications have been identified as, namely, interaction transparency, context-awareness, and automated capture of experiences Abowd [Abo99].

Pervasive computing aims at non-intrusiveness. It contrasts with the actual non-transparency of current interactions with computers. Neither input-output devices nor user manipulations are natural. Input-output devices such as mouses, keyboards, and monitors are pure artifacts of computing. So are manipulations such as launching a browser, selecting elements in a Web page, setting up an audio or video encoding mechanism, and entering authentication information (e.g., a login and a password).

Biometrics security is a field aiming at making authentication of users natural. It removes the login and password intermediate between the user and the computer. To identify an individual, it exploits the difference between human bodies. Authentication is based on physical measurements. To be usable, however, the measurements must be non-invasive and fast. DNA analysis does not meet that criteria, but fingerprint identification does. Other alternatives include facial characteristics, voice printing, retinal, and typing rhythm recognition. Input biometric information hardware and software are being marketed. It is interesting to note that practical evaluations have reported that biometric input is often not recognized and needs to be accompanied by a bypass conventional authentication procedure (login and password) in case the biometric authentication fails [(Ed00].

Another example of interaction transparency is the electronic white-board project called Classroom 2000 [Abo99]. An electronic white-board has been designed that looks and feels like a white-board rather than a computer. With ideal transparency of

interaction, the writer would just pick up a marker and start writing with no plug in, no login, and no configuration.

To achieve transparency of interaction, advanced hardware and software tools are needed such as handwriting recognition, gesture recognition, speech recognition, free-form pen interaction, and tangible user interfaces (i.e. electronic information is manipulated using day to day physical objects).

Context awareness translates to adaptation of the behavior of an application as a function of its current environment. This environment can be characterized as a physical location, an orientation or a user profile. A context-aware application can sense the environment and interpret the events that occur within it. In a mobile and wireless computing environment, changes of location and orientation are frequent. With pervasive computing, a physical device can be a personal belonging, identified and long-term personalized to its user (such as a cell phone or a PDA) or shared among several users and personalized solely for the duration of a session (such as an electronic white-board).

The project Cyberguide [Abo99] is a pervasive computing application that exploits awareness of the current physical location. It mimics on a PDA the services provided by a human tour guide when visiting a new location.

Context-aware components can sense who you are, where you are, and what you are doing and use that information to adapt their services to your needs. Mobility and services on demand are greatly impacted by the location of the devices and the requested services. Examples range from relatively rudimentary device following services such as phone call forwarding to the location of the device, to more complex issues of detecting locations of available services and selecting the optimal location for obtaining the services, such as printing services.

The complexity of the problem increases when both the service users and the service devices are mobile. These problems require dynamic and on-the-fly system configuration. The dynamics of such system are complex because it requires not only system reconfiguration and low level configuration, e.g., multiple communication and security protocols, but also service detection and monitoring in order to provide the best available services.

Capture and storage of past experiences can be used to solve new problems in the future. Experiences are made of events and computers have the ability to record them automatically. Human users only have to recall that information from the computer when it is needed. For example, a context-aware electronic wallet could capture and store locations, times, and descriptions of payments made by a traveler. Back home, the traveler could use the recorded events to generate an expense report.

1.3 ARCHITECTURE OF PERVASIVE COMPUTING SOFTWARE

The engineering of pervasive computing software is discussed in Refs. [Abo99], [Abo96]. The software of pervasive computing applications is subject to the support of every day use and continuous execution. Robustness, reliability, and availability are there-

fore required. In the sequel, we focus on issues of software engineering for pervasive computing that have to do with mobility and distribution.

An important issue that has been addressed is the architecture of mobile user-interfaces [DRRD00]. Mobility, which most of the time implies wireless communication, brings additional issues, namely, narrow-bandwidth communications, limited processing power, and restricted input/output devices (e.g. stylus based input, small screens).

With pervasive computing, information pursues the user rather than the user pursuing the information as with traditional desktop computing. This has been addressed by a research system called Personal Information Everywhere (PIE) that has been developed by Carmeli, Cohen, and Wecker [CCW00] to provide information to mobile people within an organization. The architecture of this system consists of consumers of information, PDAs running a PIE specific small kernel, and a supplier of information, a central database server (written in XML). The consumer-to-supplier communication is wireless.

An interesting aspect of this project is the partitioning of the processing between a server and a PDA in order to cope with the light processing capabilities of the latter. The model is called Mobile Application Partitioning. The kernel on the PDA handles interaction with the user. A proxy runs on the server and handles the graphic rendering and user event handling. There is one proxy per PDA. The main logic loop is as follows. The proxy gets data from the database and prepares the layout of the screen. The proxy sends messages to the PDA to render the layout of the screen. Whenever a user-generated event occurs, a signal is sent from the PDA to the proxy. The signal contains the identity of the event and the identity of the object on which the event occurred. The handler of the event is the proxy. The result translates to updates of the screen layout prepared in the proxy and rendered on the PDA.

1.4 OPEN PROTOCOLS

Open protocols are required by pervasive computing for establishing communication and collaboration between distributed components in a global infrastructure-based manner as well as in an ad hoc manner. Mobility, service discovery, and distributed computing are issues that need to be addressed.

The problem of mobility of devices, from network to network, is not solved by plain IP. It is, however, addressed by the mobile Internet Protocols (IPs). Mobile IPs are discussed in more detail in Chapter 25 of this book. In the context of the current chapter, it is worth mentioning that IPv6 is a better candidate than IPv4 for pervasive computing [Pau01]. Indeed, pervasive computing puts enormous pressure on the demand of IP addresses. The number of devices will be high and they will be continuously running, hence there is little possibility of temporal sharing of IP addresses as does DHCP. The 128-bit addresses of IPv6 can support considerably more devices than IPv4 32-bit addresses of IPv4. There is a movement in the wireless industry towards IPv6. For instance, the Third-Generation Partnership

Project (3GPP) [Pro01] has adopted IPv6 for their next generation of wireless network specification.

In the subsequent subsection, we focus on application support protocols. Service discovery and distributed computing are discussed in more detail.

1.4.1 Service discovery

Service discovery protocols are a key technology of pervasive computing. They give to distributed components the capability to advertise and discover each other's services on a network. For instance, a PDA equipped with a service discovery protocol, once attached to a network, can automatically discover a laptop advertising an agenda synchronization service.

There are leading service discovery technologies: Service Discovery Protocol (SDP) of Bluetooth [Blu99], Jini [Mic99b], Salutation [Con99], Service Location Protocol (SLP) [GPVD99], and UpnP [Cor99].

In this subsection, we review access to services on an IP network and the highlights of SLP and Jini. We also explore two related issues, namely, service selection facilitation and security.

1.4.1.1 Access to services on an IP network To establish an association with a server process from machine to machine on the Internet, the client requires the IP address of the machine on which the server is running and the port number of the socket on which the server is listening. In addition, the client needs to learn and to run a protocol understood by the server. Services are often registered under human readable names. Service names need to be mapped to machine names and port numbers on which the services are offered. Often the name of the protocol understood by the server is implicit in the name of the server (e.g. WWW implies http).

A practice in IP networks for mapping service names to machine names on which services are offered consists of naming conventions for machines offering services (e.g. mailhost.scs.carleton.ca, ftp.scs.carleton.ca, www.scs.carleton.ca) and registered associations with a DNS of standard machine name and real machine address or name (e.g. www.scs.carleton.ca is mapped to fusion.scs.carleton.ca). A drawback of this approach is that only one machine per DNS can offer a service under a standardized name. A solution to this problem has been proposed [GV96]. It is called DNS SRV Resource Records. It allows the mapping of a service name (as defined in file `/etc/services`, e.g. FTP) to names of machines offering the service.

Machine names then have to be mapped to IP addresses. This translation relies both on a local name server (DNS) and a local file listing associations of machine name and IP address (file `/etc/hosts` on Unix).

For mapping service names to port numbers on which services are offered, port numbers are standardized. Associations of service name to port number and transport protocol name are stored in a local file (e.g. `/etc/services` on Unix).

This practice has an advantage. There are no requirements for special infrastructure for service location. That is, in-place naming services support service location.

This approach has, however, several disadvantages. It is not possible to advertise several service providers of the same service name (unless DNS SRV Resource Records are used). Clients must be aware of naming conventions. Search by values of attributes is not supported. Machine names and port numbers are discovered using different mechanisms. Updates need to be performed at two different places. The introduction of new types of services requires standardization of new names. Information may not be up-to-date. In other words, this solution lacks the generality and dynamism required by the heterogeneous hardware and distributed components of a pervasive computing environment. Service discovery protocols have been devised to facilitate association of clients to servers in a heterogeneous and dynamic environment.

1.4.1.2 Service Location Protocol A service may either be of hardware nature (e.g. a network access point) or of software nature (e.g. a CORBA server). SLP is a general protocol for the advertisement and discovery of network services at the scale of an enterprise network. The service discovery process is of type *yellow pages*, that is, services can be discovered by type name and by characteristics.

Characteristics of services are described by values given to named attributes. For instance, a network access point would be described by the name of the protocol it supports (e.g. IEEE 802.11), its speed (e.g. 11 Mbps), and encryption algorithm (e.g. WEP). A *service type* is a collection of services having a common nature (e.g. all the access points) and sharing the same kinds of attributes (e.g. protocol, speed, and encryption algorithm). In SLP, the information required by a client to establish an association with a server is called a *Service Access Point* (SAP). A SAP typically contains at least a protocol name and a machine name. It could also contain a port number and the path to an executable file. A *service advertisement* is a structure of information describing a service. It contains the service type name, values of attributes, and SAP.

SLP is a mechanism for facilitating the association of entities that have services to offer or that have needs for services. In the SLP model, there are three kinds of entities: User Agent (UA), Service Agent (SA), and Directory Agent (DA). A UA represents a consumer of services, an SA represents a provider of services, and a DA represents a database of service advertisements.

SLP proposes two alternative architectures. The first involves only SAs and UAs communicating directly with each other. With the aim of reducing network traffic, a second architecture involves SAs, UAs, and DAs acting as central sources of service advertisements in which SAs register services and UAs enquire about services.

A SAP is represented by a special type of URLs, called URLs of scheme *service:* [GPK99], or a generic URL [BLFIM98]. The scheme service: is discussed in more detail hereafter.

An URL of scheme service: is made of a service type (a name) and access information:

```
"service:" service-type ":" service-access-info
```

SLP has a notion of type of service with a two-level hierarchy and a notion of instance of service. SLP concepts of type of service, hierarchy, and instance are analogous to object-oriented concepts of class, inheritance, and object.

A two-level hierarchy consists of an abstract-type service at the top and one or several concrete-type services at the bottom. An abstract-type service groups several concrete-type services that provide the same function but through different protocols. For instance, a banking service provided by distributed components is a function that can be achieved by several different concrete remote method invocation protocols such as IIOP and RMI. In this case, the abstract-type service is banking and the associated concrete-type services are IIOP and RMI. A concrete-type name provides the name of a protocol to use by a client to call a method on a distributed component.

Names of services are subject to standardization in order to achieve uniformity from one system to another. An organization that standardizes service types and names is called a naming authority. The authority from which a service name is drawn can be explicitly specified. When it is unspecified, the default naming authority is Internet Assigned Numbers Authority (IANA). It that case, the specified service type must have been standardized by IANA, e.g http, ftp, and telnet. A naming authority, can have the scope of an enterprise. The naming authority is identified by the name of a company. Other conventions also apply. For example, authority name *test* is for non-standardized services under test.

The formal syntax of the naming part (`service-type`) of an URL of scheme service: encompassing the notions of abstract-type, concrete-type, and naming authority is as follows:

```
abstract-type [ "." naming-authority ] ":" concrete-type
```

Here is an example of two concrete types of service grouped under the same abstract type of service:

```
service:banking.demo:iiop
service:banking.demo:rmi
```

banking is the abstract-type name and *banking.demo:iiop* and *banking.demo:rmi* are concrete-type names. *demo* (stands for demonstration) is the naming authority. A UA could issue a request for the abstract-type of service *banking* and would receive replies with the aforementioned two names. Both services do the same function but through different protocols, i.e. either Internet Inter-ORB Protocol (IIOP) or Remote Method Invocation (RMI). It is also possible to request by full name, both abstract-type and concrete-type.

Organization of services in a two-level hierarchy is interesting because it makes possible the grouping of services that are of the same kind, but accessible through different protocols. If this flexibility is not required, a flat organization is possible as well. In that case, types of services are said to be simple. The name (`service-type`) of a simple-type of service is structured as follows:

```
simple-type [ "." naming-authority ]
```

Often, the part `simple-type` corresponds to the name of a protocol, e.g. *http*. Here is another example where `simple-type` is not a protocol name:

```
service:banking.demo
```

In that case, the name of the protocol that should be used to communicate with the service must by inferred by some means, e.g. using preset conventions.

A URL of scheme service: also contains information required by the UA to communicate with the SA, in addition to the protocol name. Access information essentially consists of an address of a machine where the service is offered, an optional path to a file (e.g. an executable program), and an optional list of attributes representing additional information required to be able to contact the SA. The formal syntax of a URL of scheme service: with access information is as follows:

```
"/" address-family "/" address-spec
   [ "/" [ url-path ] [ ";" attribute-list ] ]
```

The part *address-family* indicates the network protocol to be used. A double slash "//", i.e. the field is empty, is for IP, *at* for Appletalk, and *ipx* for IPX.

The part *address-spec* contains a host name or an IP address and, optionally, a port number.

The part *url-path* is specific to the protocol. For example, if the protocol is *http*, the url-path is the name of a file containing an HTML page. Here is an example:

```
service:http://fusion.scs.carleton.ca/index.html
```

It is the SAP for a simple-type service named *http*. The address family is IP and the address is *fusion.scs.carleton.ca*. The url path is *index.html*. The attribute list is empty. It is important to stress that with this naming scheme it is possible to advertise a second Web server in the domain *scs.carleton.ca* provided by a different SA, for instance:

```
service:www.test:http://apex.scs.carleton.ca/index.html
```

UAs request access to Web servers by the service-type name *http*. Two SAPs will be returned. It contrasts with a conventional Internet naming scheme where, by convention, the Web server in domain *scs.carleton.ca* is advertised under the name *www.scs.carleton.ca* and is mapped to a unique machine address.

The attribute-list provides additional information required to access a service. The attribute-list is made of pairs of attribute id and value according to the following syntax:

```
attribute-id "=" value
```

`attribute-id` is common to all SAs offering the same kind of service. `Value` is specific to every SA. For example, access to a secure banking service may require the client to have a knowledge of a Security Parameter Index (SPI) that determines an authentication key and algorithm to be used by the UA to contact the SA. This can be represented as follows:

```
service:banking.demo:iiop://some.where.net;SPI=19
```

The attribute SPI tells the UA to use an authentication key and algorithm associated with the number 19 (which is arbitrary for this example).

The information represented in service advertisements needs to be described precisely. SLP defines a structure of service location information [GPK99]. It is a model of data for the precise specification of elements of service advertisements (i.e. a type of service, attributes, and a SAP). Besides, it is extensible. That is, the introduction of new types of services is possible.

A service type is described by a structure called a *service type template*. The concept of template is analogous to the concept of structure in the C programming language. Each service type is described by a template written according to formal syntax rules. Each instance of the service is specified by assigning effective values to each attribute defined in the template and defining the SAP, which may have a service type specific syntax defined in the template.

The model of a service type template is pictured in Figure 1.1 using UML notation. A service type template has a name, a version, and a description. It contains zero or more attributes and, optionally, a URL syntax definition.

The purpose of the version field is to capture the evolution of a template. A template that is under development should have a number below 1.0 whereas standardized templates should be numbered from 1.0 and above. The description field is free format text for the purpose of documentation.

Each attribute has a name, a data type, default value(s), a descriptive text, and allowed values. Valid data types are boolean, integer, string, opaque, and keyword. A value of type opaque is an array of bytes. An attribute of type keyword has no value and is like a constant.

Valid flags are *O*, *M*, *L*, and *X*. Flag O means that the attribute is optional. Flag M means that the attribute is multi-valued. Flag L means that the attribute is a literal and its name cannot be translated into another language. Flag X means that a value for this attribute must be provided by a UA when a service request is formulated for that type of service.

An SA can omit providing values for attributes when registering a service with a DA. In that case default values apply. The range of values that can be assigned to an attribute can be restricted by specifying allowed values. The syntax of the URL is of scheme service: and is described using Augmented BNF. Here is a sample template:

```
template-type = printing
template-version = 0.1
template-description =
An template example for a printing service.
template-url-syntax =
color = BOOLEAN
FALSE
speed = INTEGER O
page-queue = INTEGER O
```

It is the template of a printing service called *printing*. It has no specific URL syntax, i.e. scheme service: applies. The template has three attributes. Attribute

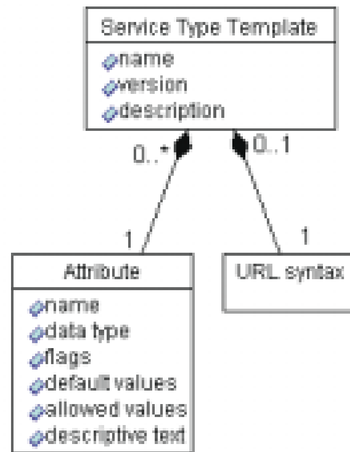

Fig. 1.1 Structure of SLP information.

color, modeling support of color printing, is mandatory and has default value false. Attribute *speed* specifies the speed of the printer, in pages per minute, and is optional. Attribute *page-queue* is also optional and indicates the current number of pages in the queue of the printer.

Interactions between DAs, SAs, and UAs are based on the following basic messages: Acknowledgment (SrvAck), Service Reply (SrvRply), Service Request (SrvRqst), and Service Registration (SrvReg).

There are two models of interaction: a model involving DAs and a model not involving DAs. Without DAs, UAs send using UDP multicast or broadcast message SrvRqst to SAs. SAs are listening and, when they find a match between a requested service and a service they offer, they reply to the UAs using unicast.

The model involving DAs is pictured in Figure 1.2. Message SrvReg is sent by an SA to a DA, using TCP or UDP unicast. The purpose of this message is registration of a new service. It contains a URL, a type of service name, and descriptive attributes. Message SrvAck is sent by a DA to an SA, using TCP or UDP unicast. Message SrvRqst is sent by a UA to a DA using UDP unicast or TCP unicast. TCP is selected when the reply can't stand in one UDP datagram. The purpose of this message is to look up for services. It contains a type of service name and a predicate which is a query evaluated over the attributes of registrations in a DA. Message SrvRply is sent by a DA to a UA using UDP unicast or TCP unicast to respond to SrvRqst. It contains URLs of SAs matching the query.

There are four different ways SAs and UAs can obtain the SAP of their DA: through a configuration file, a DHCP server, active discovery (multicast of requests by SAs and UAs), and passive discovery (multicast of advertisements by DAs).

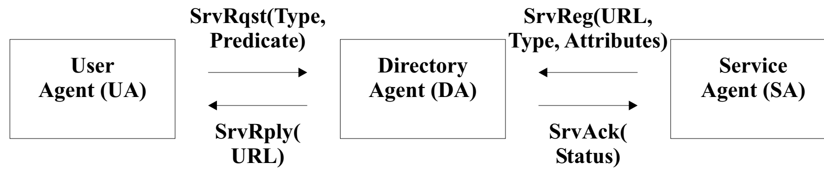

Fig. 1.2 Interaction between a DA, an SA, and a UA.

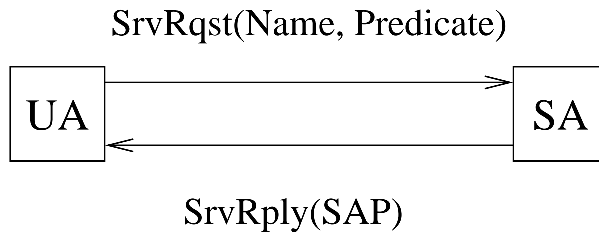

Fig. 1.3 UA and SA interaction.

Figure 1.3 pictures the operation of SLP in a DA-less architecture. A UA sends, using UDP above IP multicast or broadcast, a SrvRqst to SAs. The characteristics of the required service are specified in the SrvRqst as a service type name and a predicate over service descriptive attributes. When a listening SA finds a match between a requested service and a service it offers, it replies to the UA by sending a SrvRply using unicast. The SrvRply contains a SAP.

For ad hoc networks, the DA-less model may be more desirable. By definition, an ad hoc network doesn't rely on infrastructure.

During the process of discovery of SAs by UAs, when should the transmission of the SrvRqst, using multicast or broadcast, stop? How can the system provide with reasonable probability a complete set of available services while not waiting too long for SAs to respond?

To address this issue, SLP defines a multicast convergence algorithm. A SrvRqst is transmitted by a UA up to four times over a period of 15 seconds. Message SrvRqst contains a field called *previous responder list*. The list contains the IP addresses of the SAs that have returned SrvRplys so far in the execution of the multicast convergence algorithm. An SA listening and receiving a SrvRqst with its own IP address within the previous responder list of the message ignores the request and remains silent.

An important issue that must be addressed by a service discovery system is scalability. For instance, if the number of SAs matching a given request is high, the number of replies and amount of traffic will be high as well. To address scalability, SLP has a notion of scope. A scope is a group of UAs and SAs. DAs support scopes. UAs send SrvRqsts only to SAs and DAs supporting their scope. SAs send SrvRegs only to all the DAs supporting their scope. The concept of scope provides scalability

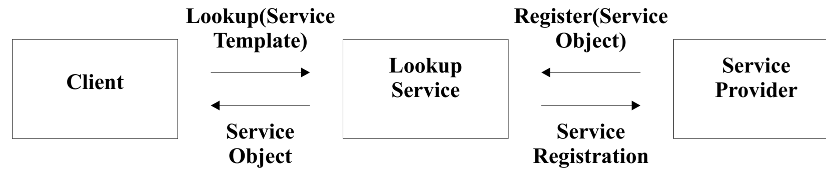

Fig. 1.4 Interaction between a Client, a Lookup Service, and a Service Provider.

limiting the network coverage of a request and the number of replies. Each scope is named. UAs and SAs can be members of several scopes. They can learn their scope name(s) by, for example, reading a configuration file.

1.4.1.3 Jini The architecture of a Jini system consists of Clients, Lookup Services, and Service Providers which are analogous to the concepts of UAs, DAs, and SAs of SLP. As SLP, Jini proposes two alternative architectures. The first architecture, with a mode of operation called peer lookup, consists of Service Providers and Clients with direct communication. The second architecture consists of Service Providers, Clients, and Lookup Services acting as central sources of information.

In Jini, a discovery protocol is used by a Service Provider or a Client to discover a Lookup Service. Thereafter, a Service Provider may register with the Lookup Service with a protocol called *Join*. A service may be located on the Lookup Service by a Client using a protocol called *Lookup*.

The discovery protocol goes as follows. A Service Provider or a Client sends a discovery request on the local network using multicast. Listening Lookup Services reply to the Service Provider or Client using unicast. Each Lookup Service returns a proxy object whose methods are used by the Service Provider or Client to contact the Lookup Service.

Following the discovery, protocol Join is used. The Service Provider calls method *register()* to load a service object (also called a proxy) into the Lookup Service (see Fig. 1.4). The service object consists of a Java interface to the service (i.e. signature of methods and descriptive attributes). The Lookup Service returns a service registration object that will be used by the Service Provider to maintain its offer of service.

Protocol Lookup is used by a Client to enquire for Service Providers with a Lookup Service. Location is by type of Java interface or by values of attributes of the service. The requirements of the Client are specified with a service template.

When a Service Provider is located, the corresponding service object is taken from the Lookup Service and loaded into the Client. Afterwards, the Client communicates through the service object (which acts as a proxy) with the Service Provider. The communication protocol used between the service object and service provider is not in the scope of Jini and is said to be private. RMI can be used for that purpose.

As SLP, presence of a Lookup Service is not mandatory. For location of a service when there are no Lookup Services, a Client can apply a technique called *peer lookup*.

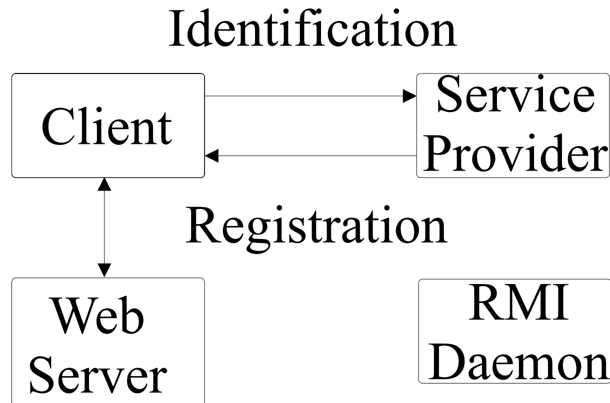

Fig. 1.5 Peer lookup.

The Client sends a message called identification to request registration messages from Service Providers (the identification message is normally sent by a Lookup Service). The Client then receives registration messages from Service Providers among which one can be selected.

In addition to components Client, Lookup Service, and Service Provider, Jini requires some elements of infrastructure. When a proxy or a service object is downloaded, only the data members are obtained. The implementation class must be down loaded separately. A Web server is required for downloading the code.

Jini is based on Remote Method Invocation (RMI) [Mic99a]. A RMI activation daemon is required to start a Java server object on demand such as the Jini Lookup Service.

In contrast to SLP, Jini needs a Java virtual machine at the Client as well as the Service Provider nodes. The Java virtual machine, for Jini as well as the technologies such as RMI and object serialization on which it depends, may not fit in a small footprint memory pervasive computing device such as a PDA or cell phone. It has also been demonstrated that Jini is substantially more chatty than SLP (for equivalent functions) [Bar00] which is something undesirable for wirelessly connected devices. Jini is better in terms of facilitation of communication because service discovery and service usage can take place in the same environment, the Java RMI environment. To reconcile the physical constraints of small devices with communication capabilities, a lightweight version of Jini called JiniLite has been created by Chen [Che00].

With JiniLite, the client is made lighter. First, the Remote Method Invocation (RMI) API is simplified. Simplification is obtained at the expense of limitations on methods parameters (e.g. only parameters of simple pre-defined types) and communication model (clients can’t be called back). Second, service objects are not loaded dynamically in the client. Clients are pre-loaded with service object stubs to provide services for which usage is foreseen. Third, service objects themselves are run on a gateway (a proxy server). Service object stubs in light clients communicate

Fig. 1.6 The JiniLite model.

with service objects in gateways through RMILite. Clients are configured with the address of their gateway. In the gateway, service objects are stored in a service registry on which clients can invoke a lookup method. When a service is provided to a client, a copy of the service object is taken from the registry and run within the gateway.

1.4.1.4 Service Selection Facilitation When a UA requests instances of a type of resource, the selection of any instance of that resource type will often not satisfy the needs. For example, given the need to fax a document, several fax machines can be discovered. There is, however, most probably one of them that is more appropriate than all the others because of physical proximity. For instance, two faxes may be discovered but, for a user located on the first floor, the one situated on the first floor is more attractive then the one situated on the twentieth floor. An issue is how can the selection of the most appropriate service to fulfill a certain need be facilitated? Service selection can be facilitated with the help of tools. Some approaches are discussed hereafter.

Selection of a service can be facilitated by using a service browser. Such a browser is presented in Ref. [HMBB00]. It provides to the user a view of the available services on the network. Figure 1.7 illustrates a view in which SAPs are listed (upper area) and descriptive attributes of a service are posted (lower area). Using the browser, a user queries the network for a service and selects one of the found services by visual inspection of the listed SAPs and attributes. The user then selects manually the service to be used.

Service selection transparency can be achieved. McCormack has developed a mechanism, called service recommendation, that ranks services with respect to one another [McC00], [HMBB01]. An SLP SrvRqst includes a predicate expressing a condition on the values of the attributes of a sought service. Predicates are limited to attribute comparative expressions. Service recommendation extends the predicate syntax with ranking functions. A ranking function takes an expression over the attributes as argument. When the ranking function is evaluated by a DA, a numerical value is returned, thus ranking a service advertisement relative to the other service advertisements registered in the DA that satisfy the predicate in the SrvRqst. The ranking function is formulated by the user or predefined in the application. It is a model of the desirability of a service. Evaluations of the ranking function on all the service advertisements matching a predicate in a SrvRqst are performed simultane-

Fig. 1.7 SAP and attribute view of the SLP Service Browser.

ously. The service with the highest/lowest rank is recommended, according to the ranking function. It is called service recommendation because the DA recommends to the UAs a service advertisement that has the highest/lowest rank according to a user-specific ranking scheme. With such a mechanism it is therefore possible to delegate to a DA the selection, for instance, of a printer with the highest printing speed and shortest queue.

Contextual information about the UAs and SAs can be used to take a service selection decision. Physical location, because of its relevance, is a type of contextual information that can be used to facilitate service selection. Physical location often amounts to physical proximity of the user and service, such as in the same office, same floor or same building. Location tracking solutions based on networks of sensors or triangulation may not suitable in an ad hoc network environment because of the infrastructure required.

Close proximity can be detected as follows. User and service devices may be equipped with infrared ports and use successful establishment of communication through the infrared ports as a confirmation of proximity.

Figure 1.8 pictures integration of a service discovery protocol with a close proximity-based selection protocol. There are a UA, a near SA, and a far SA. They are all within RF reach of each other. The UA sends using broadcast a SrvRqst. It is received by both SAs and they both send using unicast a SrvRply. This completes the service discovery phase. To achieve close proximity selection, the UA sends using multicast a message called SetIrdLink through the infrared port. This

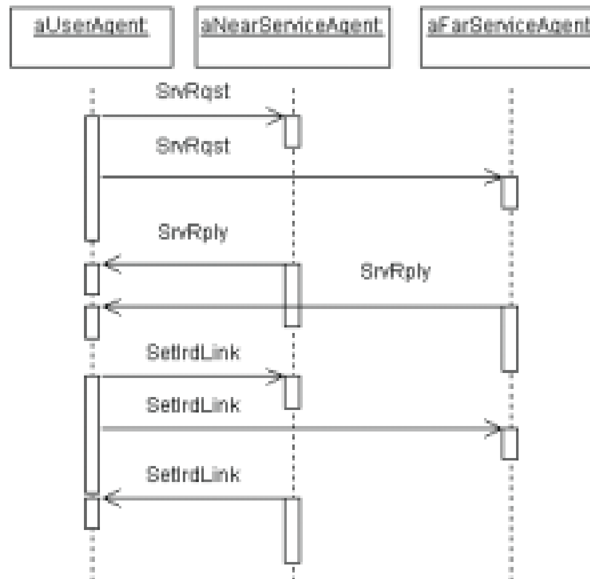

Fig. 1.8 Service discovery and close proximity based selection.

message is, however, received only by the near SA which replies with a message called SetIrdLinkConf. This completes the service selection phase.

1.4.1.5 Security Theft of service is the actual number one security problem in cellular networks [Rie00]. A similar problem exists with computer network services. Solutions devised for cellular telephony can be applied.

Control of access to services relies on a form of identification. Either a user or a device may be identified. The most desirable form, in the context of service access control, is user identification because it is independent of the device utilized by the user to access the network.

Identification of a user may be done with an identification number entered by the user before a service is accessed. Further automation can be achieved by using instead a fingerprint captured by a biometrics sensor integrated to the device. However, a number or a fingerprint should not be transmitted unprotected because these identifiers can be copied by malicious listeners. Encryption can be used for that purpose and it is supported by most of the service discovery protocols.

Device identification may be considered equivalent to user identification in cases where the device is a personal belonging of the user. Indeed, in contrast to a desktop which can be shared by several members of a family, a PDA is a personal assistant. Identification of the palmtop means as well identification of its user. Each Bluetooth device has a 48-bit identifier that can be used for that purpose.

Secret key authentication can also be used to identify users or devices. Authentication is supported by most of the service discovery protocols.

RF fingerprinting can be used as well to identify a device (more exactly its air interface). It has been observed that radio transmitters that are built according to the same specifications all exhibit unique signal characteristics. The characteristics are obtained by measuring characteristics of the signal, e.g. the time-frequency relation of the signal at the start of the transmission. Most of the RF fingerprinting technology is proprietary or subject to patents.

1.4.2 Distributed computing

A distributed system includes resources, resource managers, and clients. A resource may correspond for instance to a printer, a window on a software application or a data element. Telecommunications networks are the infrastructure on which rely distributed systems. Concretely, each resource is located on a network node and can be used remotely from other nodes using telecommunications. A resource manager is a piece of software responsible for the administration of a type of resource. It has a telecommunications interface through which users access and update the resources. A manager also enforces access policies associated with each type of resource.

The concept of component is based on the concept of object. As an object, a component is a logical entity containing information and capable of executing operations on it. A subset of the operations is accessible to the environment of a component and constitutes its interface. A call to an operation by a client of a component, a process or another component, is achieved through the transmission of a message intercepted by the interface that dispatches the request to a method associated with the operation. The method eventually returns a response to the caller.

A component deserves a new term because it is more than a normal object. An object is a unit of software reusable, without pain, as long as the hosting software is written in the same language, is on the same platform, and is co-located with the object. A distributed component infrastructure facilitates the reuse of software units, called components, across programming languages, operating systems, and network nodes.

According to the distributed component model, resources, local or remote, are abstracted as components. A uniform syntax is used to call the components, whether or not they are in the same program, process or network node. This is called access transparency.

In contrast to a client-server model, in which the client talks to a server process, in the distributed component model the client talks to a remote object that exists within a container process. That container process can embed several objects (see Figure 1.9).

Every component has a unique identity. A component can be mobile, i.e. its host can change, to improve the performance of fault tolerance. When the location of a component changes, its identity is invariant. This is called transparency of migration. Moreover, in contrast to a client-server model, the naming scheme is uniform and doesn’t change from of type of resource to another.

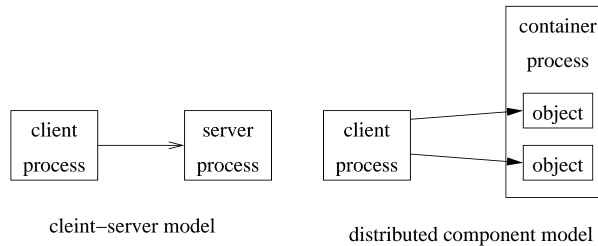

Fig. 1.9 The client-server model versus the distributed component model.

The entity responsible for the management of a component is called a component manager. The manager of the component is normally co-located with the component.

Fault transparency can be provided through the notion of service. A collection of components distributed over different nodes can supply the same services. Clients of the services select any supplier.

The Common Object Request Broker Architecture (CORBA) [Gro99] is a realization of the distributed component model. For communication between clients and distributed components, CORBA has a notion of Object Request Broker (ORB). It transmits client requests, i.e. operations calls, to components. Clients and components can be on different nodes, run on different operating systems, and be programmed in different languages. An ORB has the capability to forward requests over the network, from one operating system to another, and from one programming language to another.

When a caller and a callee are not co-located, there are two acting ORBs: an ORB co-located with the caller that encodes and sends the request on the network and an ORB co-located with the callee that receives the request from the network, decodes it, and dispatches it to the component. Inter-ORB communication is done through the Internet Inter-ORB Protocol (IIOP). A request, sent from one location to another, is encapsulated within a packet containing the identity of the target component, an operation name and parameters. CORBA is a middleware. i.e. a software that goes between parts of distributed applications.

CORBA clearly separates the notion of interface from the notion of implementation of the interface. The implementation is changeable and, behind an interface, can hide several different implementations. This provides flexibility. The interface is specified with a CORBA-specific language called the Interface Definition Language (IDL). Implementations can be programmed, however, is several different languages such as C, C++, and Java.

The development of an implementation is done as follows. The interface is written in the IDL. The IDL specification is processed by a translator that generates a representation in a target implementation language. The programmer writes in that target language methods associated to the operations of the interface. The component is compiled. The code of the component contains the elements required for its registration with an ORB when it is launched.

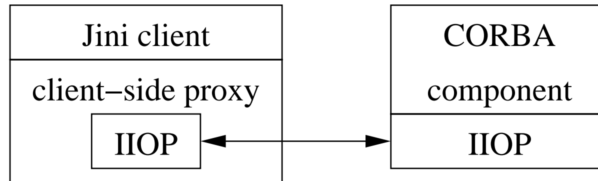

Fig. 1.10 The coexistence of CORBA and Jini.

There is a CORBA implementation for a popular PDA operating system called Palm OS. It is a port of an open source implementation for CORBA called Mico [PR00]. Mico for CORBA [Pud99] provides an API for creating CORBA clients on Palm OS. Servers cannot run on Palm OS. The capability to run servers on a PDA is a promising development. Indeed, a PDA could abstract its databases, such as address book and agenda, as CORBA objects and make them available to other applications on their PDAs. There are no IDL compilers for Palm OS hence the client stubs have to be written by the programmer. Recently, commercial versions of CORBA for PDAs have been announced [Ver00].

CORBA can coexist with a service discovery protocol such as Jini and this issue has been addressed before by Jai et al. [JOR00]. A client-side proxy, associated with a CORBA component, is registered with the lookup service, by some entity (see Figure 1.10). Having the same interface as the CORBA component, the proxy, after it has been discovered, is downloaded and co-located with the client. The proxy hides the protocol, i.e. IIOP, for the communication with the the CORBA component.

1.5 SUMMARY

Characteristics of pervasive computing applications have been discussed in Section 1.2. Interaction transparency means that human-to-computer interaction is natural and based on ordinary life objects and operations. Context awareness means that applications can sense and exploit information about the physical environment in which they are running. Automated capture of experiences exploits knowledge about actions performed in the past bound to contextual information to assist and make the resolution of new problems easier and faster.

Issues of architectures of pervasive computing that have to do with mobility and distribution were reviewed in Section 1.3. Pervasive computing platforms may be characterized by relatively narrow-bandwidth channels, slow processing power, and limited input/output capabilities. To cope with these issues, some tasks can be delegated by a pervasive computing device to a server. This approach is called application partitioning. The component-based distributed computing model is well suited to the design of such applications.

We raised the need of open protocols to enable inter-operability between the elements of pervasive computing. Service discovery protocols and distributed components architectures were addressed in more detail. Service discovery protocols, such as SLP and Jini, provide mechanisms with which distributed components can discover what each has to offer to other in terms of services. With an open distributed computing architecture, components can collaborate together using a common communication language. CORBA is an open distributed components architecture that achieves location transparency, programming language independence, and platform independence of service providers. Other open protocols not discussed in this chapter, such as mobile Internet protocols and ad hoc networking protocols, are also required to support mobile and distributed pervasive computing.

Acknowledgments

The author would like to thank the following persons for many fruitful discussions about the issues discussed in the chapter: Victor Azondekon, Francis Bordeleau, Bogdan Gheorghe, Javier Govea, Evan Hughes, David McCormack, and Ramiro Liscano.

References

Abo96. Gregory Abowd. Software engineering and programming language considerations for ubiquitous computing. *ACM Comput. Surv.*, 28(4es), December 1996. Article 190.

Abo99. Gregory D. Abowd. Software engineering issues for ubiquitous computing. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 5 – 84, 1999.

Bar00. Michel Barbeau. Bandwidth usage analysis of Service Location Protocol. In *Proceedings of Workshop on Pervasive Computing, International Conference on Parallel Processing*, pages 51–56, Toronto, August 2000. The International Association for Computers and Communications (IACC).

Bir97. Joel Birnbaum. Pervasive information systems. *Communications of the ACM*, 40(2):40–41, February 1997.

BLFIM98. T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. IETF Request for Comments: 2396, August 1998.

Blu99. Bluetooth. Specification of the Bluetooth system. www.bluetooth.com, December 1999.

CCW00. Boaz Carmeli, Benjamin Cohen, and Alan J. Wecker. Personal information everywhere (PIE). In *Proceedings of the eleventh ACM on Hypertext and Hypermedia*, pages 252–253, 2000.

Che00. Mike Chen. JiniLite white paper. www.cs.berkeley.edu/s̃ilkworm/jinilite/whitepaper.html, October 2000.

Con99. Salutation Consortium. Salutation architecture specification. www.salutation.org/specordr.htm, 1999.

Cor99. Microsoft Corporation. Universal plug and play: Background. www.upnp.org/resources/UPnPbkgnd.htm, 1999.

DRRD00. Alan Dix, Devina Ramduny, Tom Rodden, and Nigel Davies. Places to stay on the move - software architectures for mobile user interfaces. *Personal Technologies*, 4(2), 2000.

(Ed00. David A. Finck (Ed.). Biometrics security - body language. *Laptop Buyer's Guide and Handbook*, pages 94, 96, April 2000.

GPK99. E. Guttman, C. Perkins, and J. Kempf. Service templates and service: schemes. IETF Request for Comments: 2609, June 1999.

GPVD99. E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. IETF Request for Comments: 2608, June 1999.

Gro99. Object Management Group. The Common Object Request Broker: Architecture and specification. ftp.omg.org, 1999.

GV96. A. Gulbrandsen and P. Vixie. A DNS RR for specifying the location of services (dns srv). IETF Request for Comments: 2052, October 1996.

HMBB00. Evan Hughes, David McCormack, Michel Barbeau, and Francis Bordeleau. An application for discovery, configuration, and installation of SLP services. MICON 2000. Available at : www.scs.carleton.ca/b̃arbeau, 2000.

HMBB01. Evan Hughes, David McCormack, Michel Barbeau, and Francis Bordeleau. Service recommendation using SLP. In *Submitted to International Conference on Telecommunications 2001*, 2001.

JOR00. Benchiao Jai, Michael Ogg, and Aleta Ricciardi. Effortless software interoperability with Jini Connection Technology. *Bell Technical Journal*, pages 88–101, April-June 2000.

McC00. David McCormack. Service recommendation in SLP. Report for Honours Project, School of Computer Science, Carleton University, (Available at: www.scs.carleton.ca/b̃arbeau), 2000.

Mic99a. Sun Microsystems. Java remote method invocation specification, December 1999.

Mic99b. Sun Microsystems. Jini architecture specification, November 1999.

Pau01. Linda Dailey Paulson. Will wireless be IPv6's killer app? *Communications of the ACM*, 34(1):28–29, January 2001.

PR00. Arno Puder and Kay Romer. *Mico: An Open Source CORBA Implementation*. Morgan Kaufmann Publishers, 2000.

Pro01. Third-Generation Partneship Project. 3GPP - a global initiative. http://www.3gpp.org, 2001.

Pud99. Arno Puder. Mico for the Palm Pilot. http://diamant-atm.vsb.cs.uni-frankfurt.de/ mico/pilot/, 1999.

Rie00. M. J. Riezenma. Cellular security: Better, but foes still lurk. *IEEE Spectrum*, 37(6):39–42, June 2000.

Ver00. Vertel. Vertel launches next-generation CORBA for Palm OS-first ever-wireless CORBA. http://www.vertel.com, April 2000.

Wei91. M. Weiser. The computer of the 21st century. *Scientific American*, 265(3):66–75, September 1991.

Wei93. Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75 – 84, Jully 1993.