INDUSTRY SURVEY

Service Advertisement and Discovery:

Enabling Universal Device Cooperation

Golden G. Richard III
University of New Orleans

Service advertisement and discovery technologies enable device cooperation and reduce configuration hassles, a necessity in today's increasingly mobile computing environments. This article surveys five competing but similar "service discovery suites" and looks at efforts to bridge the technologies.

Computer users increasingly face the management of many computing devices. One reason is the expansion of computing environments in the home and office, as printers, scanners, digital cameras, and other peripherals are integrated into networked environments. Another reason is the proliferation of mobile devices such as laptop and palm-sized computers, cellular phones, and pagers. Because these devices trade functionality for suitable form factors and low power consumption, they are necessarily "peripheral-poor" and must therefore establish connections to neighboring devices for storage, faxing, high-speed network access, and printing.

It is easy to become frustrated when dealing with the configuration and interaction of such a multitude of devices. Service discovery technologies were developed to reduce this frustration and to simplify the use of mobile devices in a network by allowing them to be "discovered," configured, and used by other devices with a minimum of manual effort.

This article briefly surveys five of the leading technologies in this area. Table 1 lists the features of each technology. Although most of these "service discovery suites" promise similar functionality—namely, reduced configuration hassles, improved device cooperation, and automated discovery of required services—they come at the problem from different philosophical and technical approaches. Since none of these technologies is a superset of the others and none is mature enough to dominate the market, interoperation among them will require bridging mechanisms. The survey concludes with a review of some developments in this area.

BLUETOOTH: PICONETS FOR WIRELESS DEVICES

Bluetooth is a low-power, short-range, wireless radio system being developed by the Bluetooth Special Interest Group, an industry consortium whose member companies include Ericsson, Nokia, and IBM. The radio has a range of 10 meters and provides up to seven 1–megabit-per-second links to other Bluetooth devices. Bluetooth operates in the 2.4-GHz indus-

 1089-7801/00/$10.00©2000 IEEE

trial scientific and medical (ISM) band to maximize international acceptance and employs a frequency-hopping system to minimize interference. The low-level communications are detailed in the Bluetooth specification.[1]

Bluetooth has a small form factor; complete systems can be as small as 2-cm square. The technology supports both isochronous and asynchronous services. A simple isochronous application might link a cellular phone and wireless headset, where the headset and base are both Bluetooth devices. More complicated applications include automatic discovery of wireless network connections and automatic synchronization of data between several Bluetooth devices.

Table 1. Features of the five leading service discovery suites.

Feature	Bluetooth	Jini	Salutation	UPnP	SLP
Service discovery	✔	✔	✔	✔	✔
Service announcement		✔	✔	✔	✔
Service registry		✔	✔		✔
Interoperability	✔	✔	✔		✔
Security	✔	✔			✔

Figure 1 shows the Bluetooth protocol stack. At the bottom, the radio and baseband layers provide the short-range, frequency-hopping radio platform. The link manager protocol (LMP) handles data link setup and provides authentication and encryption services. The logical link control and adaptation protocol (L2CAP) supports multiplexed connectionless and connection-oriented communication over the LMP layer. L2CAP is proprietary, but other network protocols, such as IP, can be built on top of it. L2CAP is also used by higher level protocols. For example, Figure 1 shows links to the Hayes-compatible AT (ATtention) protocol, which provides a standard interface for controlling remote cellular phones and modems; RFComm, which emulates an RS-232 serial interface; a simple object exchange protocol (OBEX), which enhances Bluetooth's interoperability with IrDA; and Bluetooth's service discovery protocol (SDP).

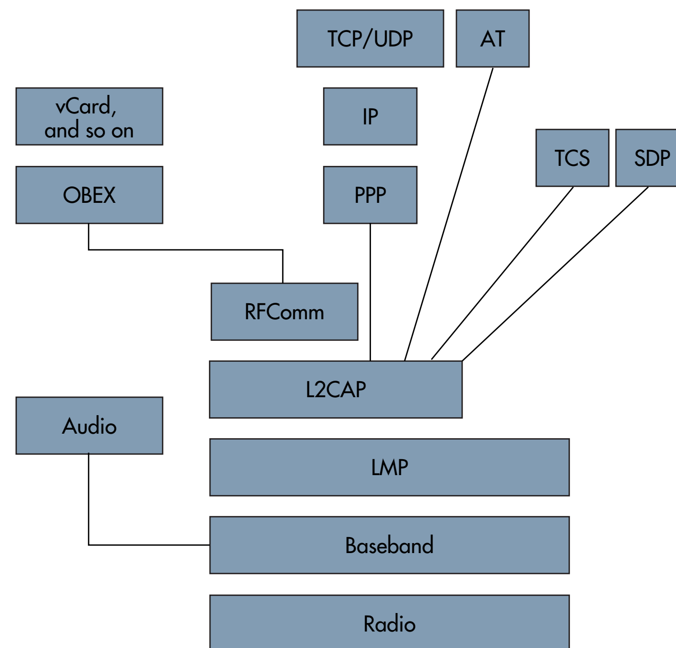

Figure 1. Bluetooth protocol stack. The link manager protocol (LMP) controls link setup and provides encryption and authentication services. The proprietary logical link control and adaptation protocol (L2CAP) provides multiplexed communication over LMP to higher level layers.

Groups of up to eight Bluetooth devices can form ad hoc networks called *piconets* to communicate, share services, and synchronize data. In each piconet, a master device coordinates the other Bluetooth devices (including setting the 1,600–hops-per-second frequency-hopping pattern). Individual devices can participate in more than one piconet at a time and can be in one of several states:

- *Standby*—the device is conserving power and waiting to connect to another Bluetooth device.
- *Inquire*—the device is searching for nearby Bluetooth devices.
- *Page*—the device is connecting to another Bluetooth device.
- *Connected*—the device is connected to another Bluetooth device.
- *Hold* and *park*—the device is participating in a piconet with varying degrees of power savings.

The Bluetooth SDP provides a simple API for enumerating the devices in range and browsing available services. It also supports *stop rules* that limit the duration of searches or the number of devices returned. Client applications use the API to search for available services either by service

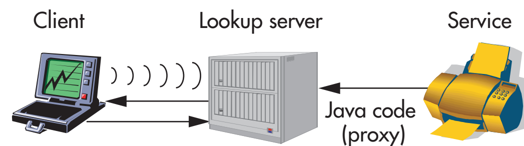

Figure 2. Jini service discovery entities: clients, lookup servers, and services. In this example, a printer service registers a *proxy* object with a lookup server, which will serve as a remote control for clients that use the service.

classes, which uniquely identify types of devices (such as printers or storage devices), or by matching attributes (such as a model number or supported protocol). Attributes that describe the services offered by a Bluetooth device are stored as a service record and are maintained by the device's SDP server.

Jini requires each device either to run a Java virtual machine or to associate itself with a device that can execute a JVM on its behalf.

The distinction between service classes and descriptive attributes is not well defined, but service classes generally define broad device categories, such as Printer, ColorPrinter, and PostScriptPrinter, while attributes allow a finer level of description. Manufacturers must eventually standardize these service classes for maximal interoperability between Bluetooth devices.

Unlike higher level service discovery technologies such as Jini, Bluetooth's SDP does not provide a mechanism for using discovered services—specific actions required to use a service must be provided by a higher level protocol. However, it does define a standard attribute ProtocolDescriptorList, which enumerates appropriate protocols for communicating with a service.

Bluetooth devices provide data security through unique 48-bit identifiers, 128-bit authentication keys, and 8- to 128-bit encryption keys. Strong authentication is possible because no international restrictions prevent it, but Bluetooth devices must negotiate encryption strength to comply with laws restricting encryption. Note that Bluetooth devices must be paired to provide them with matching secret keys that will support authentication. Once paired, Bluetooth devices can authenticate each other and protect sensitive data from snooping. Regardless of encryption strength, Bluetooth's fast frequency-hopping scheme makes snooping difficult.

JINI: MOBILE JAVA CODE

Jini is a service discovery and advertisement system that relies on mobile code and leverages the platform independence of the Java language.[2] The current Jini implementation is based on TCP and UDP, but implementations based on other network protocols are certainly possible. The major requirements are reliable, stream-oriented communication and a multicast facility. Jini's language-centric approach allows a flexible definition of service; for example, a service can be implemented entirely in software and, after discovery, can be downloaded and executed entirely on the client. Examples of such algorithmic services might include an implementation of a proprietary algorithm for shading a polygon or formatting a document to meet an organizational standard. On the other hand, Jini also requires each device either to run a Java virtual machine or to associate itself with a device that can execute a JVM on its behalf. For example, a Jini "device chassis" might Jini-enable a number of "dumb" devices, making their services available to Jini clients.

Jini entities consist of *services*, *lookup servers* that catalog available services, and *clients* that require services. A service can also be a client; for example, a telescope might provide pictures to a PDA as a service and look for printing services as a client. All service advertisements and requests go through a lookup server. Figure 2 illustrates the discovery and registration process for Jini clients and services.

To register service availability or to discover services, a service or client must first locate one or more lookup servers by using a *multicast request protocol*. This request protocol terminates with the invocation of a *unicast discovery protocol*, which clients and services use to communicate with a specific lookup server. The unicast protocol culmi-

nates in the transfer of an instance of the ServiceRegistrar class, a "remote control" for the lookup server. A lookup server can use the *multicast announcement protocol* to announce its presence on the network. When a lookup server invokes this protocol, clients and services that have registered interest in receiving announcements of new lookup services are notified.

These three protocols are encapsulated in a set of Jini classes. For example, to find lookup services, a client or service need only create an instance of LookupDiscovery.

Jini uses Java's remote method invocation (RMI) facility for all interactions between either a client or a service and the lookup server (after the initial discovery of the lookup server). Once a lookup server has been discovered and an instance of ServiceRegistrar is available, services can register their availability, and clients can search for needed services by invoking ServiceRegistrar methods.

Jini associates a proxy, or *remote control object*, with each service instance. A service advertises its availability by registering its object in one or more lookup servers via the register() method. This method takes several arguments, including an instance of ServiceItem, which contains a universally unique identifier for the service, its attribute set, and its remote control object. This object may either implement the service entirely (in the case of an algorithmic service such as the implementation of a polygon-shading algorithm), or provide methods for accessing the service over the network. The leaseduration parameter of register() specifies the service's intended lifetime. The service is responsible for renewing the lease within the time specified to maintain its listing. The lookup server is free to adjust the lease time, which is returned in a ServiceRegistration object.

When a service first contacts a lookup server, the server generates a unique identifier for it; the service uses this ID in all future registrations. The service identifier lets clients request a specific service explicitly and recognize when services reported by different lookup servers are identical.

To use a service, a device must first secure an instance of the proxy object for it. From a client point of view, the location of the service proxied by this remote control object is unimportant, because the object encapsulates the location of the service and the protocol necessary to operate it.

Clients use the lookup() method in ServiceRegistrar to discover services. This method takes a single argument, an instance of ServiceTemplate. The ServiceTemplate constructor takes several arguments. The first is the service identifier. If the service identifier is null, then arrays of types (Java classes, typically interfaces) and attributes (attribute objects) are used to match services. A service matches if its class matches one of the classes in the types array and if, for each of the attribute objects, all non-null members match one of the service's registered attributes. The return value from lookup() is an instance of ServiceMatches, which contains an array of remote control objects for the services that match. Finally, the notify() method allows a client to request an asynchronous notification when services matching a ServiceTemplate instance become available. This method uses Jini's distributed events mechanism, which extends Java's infrastructure for eventing across JVMs.

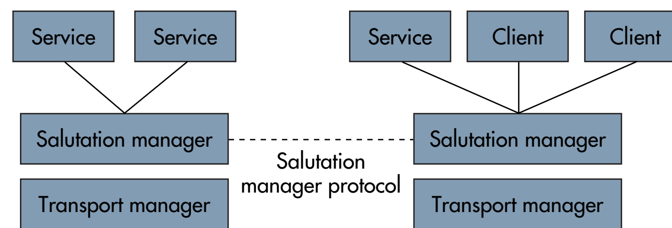

Figure 3. Salutation architecture. Salutation managers are service brokers, isolated by transport managers from the details of specific network transport protocols.

Jini depends on Java's security model, which provides tools like digital certificates, encryption, and control over mobile code activities such as opening and accepting socket connections, reading and writing to specific files, and using native methods. Systems administrators can establish different policies depending on where the Java code originated (for example, the local file system or a remote machine).

SALUTATION: A NETWORK-INDEPENDENT ARCHITECTURE

Salutation is an architecture for service discovery under development by the Salutation Consortium, which includes members from both industry and academia.[3] The consortium's goal is to build a royalty-free architecture for service advertisement and discovery that is independent of a particular network transport.

Figure 3 shows the three fundamental components in the Salutation architecture: *functional units,*

salutation managers, and *transport managers*. From a client's point of view, a functional unit defines a service. Functional units already specified or under consideration by the Salutation Consortium include printing, faxing, and document storage. There is also work on a functional unit specification to allow discovery of Hewlett-Packard JetSend-enabled devices. The specifications define attributes that characterize a service (for example, in the case of a printer, double-sided capability, color, and so on).

Salutation requires a network transport protocol that supports reliable, stream-oriented communication.

The functional unit Doc Storage defines file attributes that can be used to find information in temporary or long-term storage. For example, a client can search for operating system-specific drivers or software necessary to interact with a newly discovered device. The client simply queries a Salutation manager for the necessary Doc Storage functional unit, extracts the application or device driver, and installs it, thus providing limited code mobility.

Salutation managers function as service brokers; they help clients find needed services and let services register their availability. Services can register and unregister functional units with the local Salutation manager by using the API calls slmRegisterCapabilities() and slmUnregisterCapabilities(), respectively. A client can use the slmSearchCapability() call to determine if Salutation managers have registered specific functional units. Under the current version of the architecture, applications can query only the local Salutation manager. Future versions will allow remote Salutation managers to be specified. Once a functional unit is discovered, slmQueryCapability() can be used to verify that a functional unit has certain capabilities. The API also includes calls for initialization/version checking, availability checking, and communication between clients and services. (An API simulator is available at http://www.salutation.org/simulate.htm.)

Salutation managers fill a role similar to lookup servers in Jini, but they can also manage the connections between clients and services. A Salutation manager can operate in one of three "personalities":

- In *native* personality, Salutation managers are used only for discovery. They establish a connection between a client and service but perform no further operations on the data stream.
- The *emulated* personality is similar to the native personality in that Salutation managers set up the connection, but in this case they transfer native data packets encapsulated in Salutation manager protocol format, providing a bridge when no common message protocol exists between client and service. The Salutation manager is ignorant of the semantic content of the data stream between client and service.
- In *Salutation* personality, Salutation managers establish the connection between client and service, and they also mandate the specific format of the data transferred. The Salutation architecture defines the data formats.

A transport manager isolates the implementation of the Salutation manager from particular transport-layer protocols and thereby gives Salutation network transport independence. To support a new network transport requires a new transport manager to be written, but does not require modifications to the Salutation manager. Like Jini (and UPnP), Salutation requires a network transport protocol that supports reliable, stream-oriented communication. Initial implementations are based on IP and IrDA because of their widespread use.

Transport managers also locate the Salutation managers on their respective network segments via either multicast, static configuration, or reference to a centralized directory. Discovery of other Salutation managers allows a particular Salutation manager to determine which functional units have been registered and to allow clients access to these remote services. Communication between Salutation managers is based on remote procedure call (RPC). This interaction between remote Salutation managers contrasts with other registry-based service discovery mechanisms (for example, Jini and Service Location Protocol), where clients would be responsible for locating remote registries.

The Salutation specification currently does not address security issues.

A lightweight version of Salutation, called Salutation-Lite, has been developed for resource-limited devices. It is based primarily on IrDA to leverage the large number of infrared-capable devices. Salutation-Lite focuses primarily on service discovery. It uses the functional units OpEnvironment and Display to describe the operating system, processor

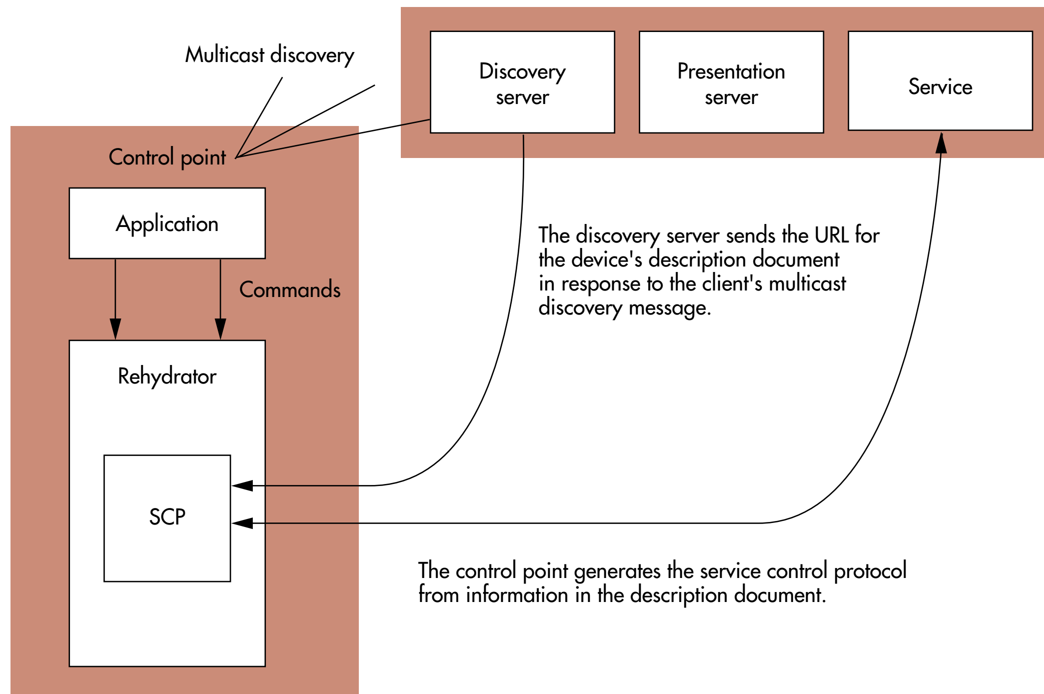

Figure 4. Interaction between a client (control point) and a service in UPnP. The control point discovers the device by sending a multicast message. The device responds with a URL pointing to its description document, which the control point can download for pertinent information, including a URL to which control messages can be sent and the protocol for interacting with the device through this control URL. The "rehydrator" converts generic commands into device-specific control messages.

class, amount of memory, and display characteristics of palm-sized devices. By noting the particular characteristics of the device, servers can provide appropriate drivers and software wirelessly.

Salutation-Lite implementations can be downloaded free from the Salutation website at http://www.salutation.org.

UPnP: XML FOR A WEB-BASED ARCHITECTURE

UPnP is a proposed architecture for service advertisement and discovery supported by the UPnP Forum, headed by Microsoft. Unlike Jini, which depends on mobile code, UPnP aims to standardize the protocols used by devices to communicate, using XML. The UPnP specification[4] is still in a preliminary stage; major issues like security have not yet been addressed.

UPnP's device model is hierarchical. In a compound device (for example, a VCR/TV combo), the *root device* is discoverable, and a client (called a *control point*) can address the individual subdevices (for example, a tuner) independently. Virtual Web servers in the device act as entry points for interacting with and controlling it. Devices that don't speak UPnP directly are called *bridged devices*. They can be integrated into a UPnP network in a manner similar to the integration in a Jini device chassis: A bridge maps between UPnP and device-native protocols.

The UPnP specification describes device addressing, service advertisement and discovery, device control, eventing, and presentation. The eventing facility allows clients to watch for significant changes in the state of a discovered service. It functions similarly to Jini's distributed event facility. Presentation allows a client to obtain a GUI for a discovered device through one of the device's virtual Web servers. Several protocols support these functions:

- AutoIP,[5] a simple protocol that allows devices to dynamically claim IP addresses in the absence of a DHCP server;
- Simple service discovery protocol (SSDP), the UPnP mechanism for service discovery and advertisement;

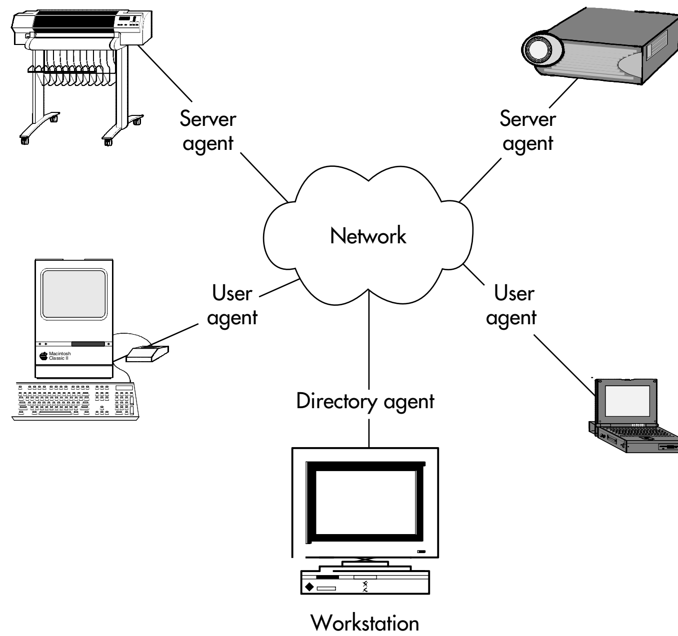

Figure 5. SLP entities: user agents, directory agents, and service agents. UAs discover services on behalf of applications, either via a DA or directly through an SA. In this example, a laptop and desktop are clients seeking services. A plotter and LCD projection system are services advertising their availability.

- Simple object access protocol (SOAP),[6] a protocol for remote procedure calls based on XML and HTTP that is used for device control after discovery; and
- Generic Event Notification Architecture (GENA), a UPnP subscription-based event notification service based on HTTP.

When devices are introduced into a network, they multicast "alive" messages to control points. When they wish to cancel availability of their services, they send "byebye" messages. In SSDP, each service has three associated IDs—service type, service name, and location—which are multicast when services are advertised. Any of these IDs can also be used to search for services.

To search, a control point sends a UDP multicast request to the network, as shown in Figure 4. Matching services send unicast responses to the client. These responses contain URLs, each pointing to an XML *description document* that describes a service. A description document contains several important items:

- A *presentation URL* allows entry to a device's root page, which provides a GUI for device control.
- A *control URL* is the entry point to the device's control server, which accepts device-specific commands to control the device.
- An *event subscription URL* can be used by clients to subscribe to the device's event service. The client provides an *event sink URL* in the subscription request. Significant state changes in the device result in a notification to the client's event sink URL.
- A *service control protocol definition* describes the protocol for interacting with the device.

The service control protocol (SCP) definition allows APIs to be converted to device-specific commands, shielding the application level from details of particular devices. After retrieving the description document, a UPnP component on the control point called the *rehydrator* is "plumbed" with a definition of the device's SCP. This component then sends device-specific commands via the device's control URL. SOAP is used for this interaction.

SSDP is similar to the Internet Engineering Task Force's service location protocol, but it lacks a query facility that can search for services by attributes. Further, SLP incorporates security measures and can interact with the IETF standards-track dynamic host configuration protocol (DHCP)[7] and the lightweight directory protocol (LDAP).[8] Finally, SSDP specifications currently limit discovery to a single subnet. Since UPnP does not use a registry, it is also likely to generate significantly more network traffic than SLP.

SLP: A PROPOSED IETF STANDARD

Service location protocol is an IETF protocol for service discovery and advertisement.[9] It is currently at the "proposed standard" stage along the IETF standards track. Unlike Jini, Salutation, and UPnP, which all aspire to some degree of transport-level independence, SLP is designed solely for IP-based networks. It provides a set of C and Java bindings that provide service discovery and advertisement functions to application software.

SLP comprises three entities: *service agents* (SAs), *user agents* (UAs), and *directory agents* (DAs). SAs advertise the location and attributes of available services, while UAs discover the location and attributes of services needed by client software. UAs can discover services by issuing a directory-like query to the network. DAs cache information about available services. Unlike Jini, SLP can operate without directory servers. The presence of one or more DAs can substantially improve perfor-

mance, however, by reducing the number of multicast messages and the amount of network bandwidth used. In fact, if DHCP is used to configure SLP agents with the location of DAs, then multicast is completely unnecessary. SLP also interoperates with LDAP, so services registered with an SLP DA can be automatically registered in an LDAP directory. This eliminates the need to reconfigure clients that already discover services using LDAP.

SLP has several mechanisms for discovering DAs:

- In passive discovery, SAs and UAs listen for multicast announcements from DAs, which periodically repeat these advertisements.
- In active discovery, SAs and UAs multicast SLP requests or use DHCP to discover DAs. When a DA is present, SAs and UAs use unicast communication to, respectively, register their services and find appropriate services.

In the absence of DAs, UAs multicast requests for service and receive unicast responses directly from the SAs that control matching services. This tends to increase bandwidth consumption, but provides a simpler model, appropriate for small networks (such as a home LAN).

SLP services are advertised through a service URL, which contains all information necessary to contact a service. Clients use the service URL to connect to the service. The protocol used between the client and server is outside the scope of the SLP specification. This separation is similar to Bluetooth, where the SDP does not specifically address how devices will communicate.

Service templates define an attribute set for each service type (a printer, for example).[10] The attributes include a specification of the attribute types and information about default and allowed values; they are used to differentiate between services of the same type and to communicate configuration information to UAs.

SLP doesn't define the protocols for communication between clients and services, and so its security model concentrates on preventing the malicious propagation of false information about service locations. SAs can include digital signatures when registering so DAs and UAs can verify their identity. Digital signatures can also be required when DAs advertise their availability, allowing UAs and SAs to avoid rogue DAs (that is, those without a proper signature). As with Jini, setting up the security features of SLP requires some configuration effort, but the effort can be well worth it, particularly in open environments.

BRIDGING THE TECHNOLOGIES

For service discovery to become pervasive, either a single service discovery technology must dominate or the most commonly used technologies must be made interoperable. Currently, bridging seems to be the most promising prospect for interoperability.

Implementations of certain low-level functions of service discovery (such as discovering registries) are interchangeable.

Implementations of certain low-level functions of service discovery (such as discovering registries) are interchangeable. For example, the Salutation Consortium uses SLP for service discovery beyond the local subnet. This lets the Salutation Manager search for SLP DAs, and then use SLP to register functional units and search for requested services.

A Jini-SLP bridge has also been developed, which allows services lacking a JVM to participate in Jini systems.[11] The heart of the Jini-SLP bridge is a special SLP UA that registers the availability of "Jini-capable" SLP SAs. To do this, Jini-capable SLP services advertise the availability of a Jini driver factory. The UA discovers all SAs with driver factories and registers them with one or more Jini lookup services. When a Jini client needs one of the registered SAs, it downloads the driver factory from the lookup server and uses it to instantiate a Java object to drive the service. Note that the SLP SAs are *not* required to host a Java virtual machine—the Java code installed on the SAs is static. Similar schemes are possible for the other technologies; for example, it should be possible to Jini-enable UPnP services in this way.

Miller and Pascoe[12] describe mapping Salutation to Bluetooth SDP to take advantage of Bluetooth's wireless capability. Two approaches are considered: The first maps the Salutation APIs to Bluetooth SDP by implementing Salutation on top of Bluetooth; the second uses a Bluetooth transport manager and essentially replaces Bluetooth SDP with Salutation. This approach will also

work with other schemes, like Jini. Bluetooth is a particularly attractive target for interoperability, primarily because of its wireless capability. Because of this, additional interoperability efforts between Bluetooth and other service discovery technologies seem inevitable.

Each service discovery technology has advantages and disadvantages. Currently, interoperability efforts are perhaps the most important force in service discovery, since it is very unlikely that device manufacturers will embrace multiple service discovery technologies on low-cost devices. ■

ACKNOWLEDGMENTS

Thanks to Sumi Helal of the University of Florida for sparking my interest in this area. Much of the material in this article is derived from a tutorial he invited me to create for IPCCC 2000 in Phoenix. Many thanks to Erik Guttman of Sun Microsystems for clarifying the differences between SLP and SSDP and going *way* beyond the call of duty in critiquing early versions of this article. David La Motta and Kirk Perilloux were kind enough to read early versions and offer suggestions. Finally, my personal editor (and mate) Christine Ciarmello-Richard was gracious enough to lend her critical eye, as always.

REFERENCES

1. *Specification of the Bluetooth System;* available at http://www.bluetooth.com/developer/specification/specification.asp.
2. K. Arnold et al., *The Jini Specification*, Addison-Wesley Longman, Reading, Mass., 1999.
3. *Salutation Architecture Specification;* available online at http://www.salutation.org/specordr.htm.
4. *Universal Plug and Play specification v1.0;* available online at http://www.upnp.org/.
5. R. Troll, "Automatically Choosing an IP Address in an Ad-Hoc IPv4 Network," IETF Internet draft, work in progress, Mar. 2000.
6. Simple Object Access Protocol (SOAP) 1.1, W3C Note; available online at http://www.w3.org/TR/SOAP.
7. R. Droms, "Dynamic Host Configuration Protocol," IETF RFC 2131, Mar. 1997; available online at http://www.dhcp.org/rfc2131.html.
8. M. Wahl, T. Howes, and S. Kille, "Lightweight Directory Access Protocol, version 3," IETF RFC 2251, Dec. 1997; available online at http://www.rfc-editor.org/rfc/rfc2251.txt.
9. E. Guttman, "Service Location Protocol: Automatic Discovery of IP Network Services," *IEEE Internet Computing*, vol. 3, no. 4, July/Aug. 1999, pp. 71-80.
10. E. Guttman, C. Perkins, and J. Kempf, "Service Templates and Service: Schemes," IETF RFC 2609, June 1999; available online at http://www.rfc-editor.org/rfc/rfc2609.txt.
11. E. Guttman and J. Kempf, "Automatic Discovery of Thin Servers: SLP, Jini and the SLP-Jini Bridge," *Proc. 25th Ann. Conf. IEEE Industrial Electronics Soc.* (IECON 99), IEEE Press, Piscataway, N.J., 1999.
12. B. Miller and R. Pascoe, "Mapping Salutation Architecture APIs to the Bluetooth Service Discovery Layer," white paper; available online at http://www.salutation.org/whitepaper/BtoothMapping.pdf.

Golden G. Richard III is an assistant professor of computer science at the University of New Orleans in Louisiana. His research interests include mobile computing, wireless networking, operating systems, and fault tolerance. He is on the executive committee of the IEEE Technical Committee on the Internet, a member of the IEEE and the ACM, and liaison to the University of New Orleans for Usenix's Educational Outreach Program.

Readers may contact the author at golden@cs.uno.edu.

How to Reach IC

Writers

We welcome submissions about Internet application technologies. For detailed instructions and information on peer review, *IEEE Internet Computing*'s author guidelines are available online at http://computer.org/internet/edguide.htm.

Letters to the Editor

Please send letters via e-mail to internet-computing@computer.org.

Reuse Permission

For permission to reprint an article published in *IC*, contact William J. Hagen, IEEE Copyrights and Trademarks Manager, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855-1331; w.hagen@ieee.org. Complete information is available at http://computer.org/permission.htm. To purchase reprints, visit http://computer.org/author/reprint.htm.