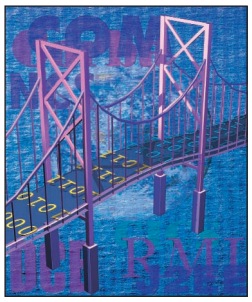
Service Discovery 101



Steve Vinoski • *IONA Technologies* • *vinoski@ieee.org*

I recently had to replace the furnace in my home. I'd never had to do that before, so I had to figure out whom to hire to get good quality at a fair price. Unfortunately, none of my friends or coworkers could offer any first-hand advice. I did a few online searches, found some suitable furnace service companies in my area, and requested estimates from each of them. I finally chose a service provider based on the type of furnace I wanted and the cost estimate the provider gave me. (I'm happy to report that the job went according to plan.)

In the big picture, my little furnace anecdote is rather unremarkable. In our day-to-day lives, each of us frequently requires services on our homes, mechanical equipment, lawns, swimming pools, and even our persons, such as with hair care and dentistry. We find suitable suppliers for each service by networking with family, friends, and acquaintances or by using search services such as telephone book or online directories to decide who can best provide the service with the qualities we desire.

A key aspect of the service-oriented architecture approach[1] for middleware is that services advertise themselves using directory or lookup services so that prospective clients can find them. The service location and discovery abstractions required to support this aren't much different from those we use to conduct business with other people. As a result, we can gain insights into how distributed service discovery systems work by comparing them to everyday human-oriented service discovery approaches.

Back in the Old Days

Years ago, people didn't have much choice when it came to service providers. In the old American West, for example, if your horse needed shoe work, you went to the blacksmith. If you needed medical attention, you sent word to the doctor. Not a blacksmith or a doctor, but *the* blacksmith or *the* doctor — people possessing such skills were few and far between, and communities were lucky if they had such professionals available within close proximity. If these people were away or otherwise unavailable when you required their services, you were out of luck.

Before the 1990s, many distributed systems were a lot like this, typically single instances hard-wired together. Service consumer applications required knowledge of low-level network addressing details, such as numeric network addresses and port numbers, to talk to service provider applications. If those addressing details changed, the consumers also had to change and be redeployed. If a provider application did not perform well, the consumer applications had to either work around it or suffer through it. If the network became disconnected or if the service provider stopped running, the consumer applications were generally unable to continue operating. In such systems, there was no service discovery to speak of.

The American West has advanced well beyond its one-blacksmith and one-doctor days. Similarly, distributed systems architectures have advanced in terms of scale, availability, and uptime, with service discovery normally considered a critical part of any distributed computing system. But despite these advances in understanding and capabilities, a surprising number of distributed systems designed and deployed today do not use service discovery.

This lack of foresight is usually due to designers thinking of service discovery as overkill or as unnecessary performance overhead. They think, "My system will never need to be extended to the point that it needs service discovery." Often, such systems either wither on the vine, quickly failing and winding up on the compost heap, or they become victims of their own success and require continuous costly maintenance just to keep them up and running.

 1089-7801/03/$17.00©2003 IEEE Published by the IEEE Computer Society

Naming

Early telephone users would pick up a receiver and crank a handle. An operator then would ask them whom they were trying to contact. Callers supplied the name of the person or business they were trying to reach, and the operator created the connection for them. Later, as the phone system grew and systems of phone numbers stabilized, callers could still ask the operator to connect them to a specific person, or they could ask them to connect to a number.

In software development, we're fond of solving problems by adding levels of indirection. Much like a telephone operator, a naming service provides a level of indirection that isolates service consumers from the addressing and location details of service providers. Consumers need know only the name of the service they wish to find. The naming service stores an association between a name and the address and location details for the named service.

Like a caller using telephone white pages, a consumer application looks up the desired service name in the naming service, which returns the associated service provider information. The consumer then uses this information to interact with the provider. Performing a name lookup on an implementation of the Java Naming and Directory Interface, for example, returns a Java object that the caller can use to invoke the named service.

One difference between most naming services today and the telephone operators of yesteryear is that a naming service typically does not connect a caller to the service it's requesting. The reason for this is the same as one of the reasons telephone operators are used relatively less frequently than they once were: scalability.

A naming service implementation that routes all messages between callers and services will become a bottleneck as the number of callers and services increases. Telephone white pages effectively distribute lookup processing among subscribers, thereby significantly reducing workload for centralized operators. Similarly, naming services supply enough information to let consumers interact directly with services and stay out of the way otherwise, thus avoiding becoming centralized bottlenecks.

Such rating systems are simply the Web version of traditional "word of mouth" networking used to find the best mechanic, carpenter, and so on.

Trading

If a pipe were to burst in your home, you'd probably want to call a plumber as quickly as possible. Unless you happen to have a plumber as a personal friend, it's unlikely that you could use the white pages to find one. Instead, you'd turn to the yellow pages, look up the category "Plumbers," and then choose one from among those listed. Perhaps you'd narrow your choice by first considering only those plumbers in the same city as you, then narrow it further by considering only those who offer 24-hour emergency service, and finally choose the first one from the remaining list.

In distributed systems, a discovery service that provides lookup based on service properties is generally referred to as a trader. A consumer first decides what properties it desires from the service it's seeking. The trading service matches those properties against the properties of the services registered with it, returning to the consumer details for those services that match. Matching algorithms are usually nontrivial, tend to involve mandatory and optional properties and multiple property types (such as Boolean, integer, and string), and usually let prospective consumers specify the tolerances, ranges, and operations to be applied to determine matching.

A common trader example is one in which an application searches for a printing service. The application requires a color Postscript printer that can print on A4 paper at a rate of 6 to 8 ppm, so it expresses these properties as part of its lookup operation on the trader. Based on these properties, the trader returns a list of printers that provide the qualities the application seeks.

This list not only contains information on how to interact with each printer, but it also contains all the properties exported by each printer. The application might use these additional properties to further narrow the selection. For example, each printer might have a "location" property that the application displays in its GUI to let the user select the printer from among those returned by the trader.

You can find many trader-like systems on the Web. For example, imagine that you're looking for a vendor for a particular type of item. There are sites that let people rate online vendors so that prospective buyers can use the ratings to choose the best vendors and avoid those who provide poor-quality service. Such rating systems are simply the Web version of traditional "word of mouth" networking that friends and families use to find the best mechanic, carpenter, and so on. Without rating or evaluation systems, suppliers could make virtually any claims they wanted. The accuracy of their service properties would thus be questionable, and would, as a result, be ultimately useless to prospective customers.

The much-hyped world of Web services in which applications trade for other applications' services over the

Internet requires trader capabilities but, unfortunately, few trader systems actually incorporate any form of rating or feedback. The OMG Trader,[2] though, does support dynamic properties, which are evaluated dynamically when a lookup operation is performed, not statically when the service registers with the trader.

While dynamic properties provide greater service description accuracy—for example, they could be used to implement a load-balancing service by having consumers look up services based on lightest dynamic load—they're still service-controlled and, thus, do not enable consumers to use them to rate the services they use. UDDI[3] provides no dynamic properties. Dynamic discovery and use of Web services on an Internet scale in business settings cannot work unless adequate rating services are in place.

Flexible Discovery

Given that service discovery depends on discovery services, how does an application actually find a discovery service? When it comes to discovering services in our everyday lives, we know to use the phone book, call the telephone directory service, ask friends or family, or use online rating sites. Applications, however, generally do not have the luxury of the richness and stability of human communications available to them. For applications, references or handles to discovery services are often supplied through configuration mechanisms that load the information at application startup.

Relying on static configuration data to bootstrap an application with respect to discovery services works, but it's loaded with potential problems. For example, what happens if the naming service configured for a collection of applications goes down? Configuration can supply information for backup services as well, but what if the backups fail? There's a definite limit to the amount of configuration data that administrators are willing to maintain manually for enterprise-scale distributed systems. They want to specify the bare minimum amount of bootstrapping configuration data required to get the system up and running, and then they want the system alone to discover and keep track of everything else. Imagine if you had to manually specify information to your phone about how to connect itself into the telephone system every time you wanted to use it.

Techniques based on network multicast or broadcast provide an elegant solution to the problem of discovering discovery services. Such techniques effectively add yet another level of indirection. Rather than attempting to contact a naming service at a particular address hard-coded into a configuration or properties file, for example, an application might send a request for the service to a broadcast or multicast address, then wait for naming service implementations to respond with information about how to request services from them using regular point-to-point style communications. Note that such solutions also handily address issues related to discovery service failure. Should an application find that the discovery service it's using has failed, it need only send another broadcast or multicast message to find a new one.

I've recently become intrigued by Apple's Rendezvous service discovery protocol.[4] While Rendezvous is directed more at easily connecting computers and devices through multicast-based and trader-like mechanisms, it applies to software as well. Apple uses it for its iChat instant messenger system, for example, as well as for file sharing.

Imagine a middleware world in which all kinds of services — for example, naming services, traders, event and notification services, load balancers, and system monitors and managers — could discover each other and effectively use each other's services without the hassle of administrative setup or maintenance. Jini (see www.sun.com/software/jini) proved that this approach is workable, at least for Java-centric distributed systems. Just as the dynamic host configuration protocol (DHCP) lets me easily configure my laptop into most networks wherever my travels take me, protocols such as Rendezvous will push middleware systems toward supporting self-configuration and dynamic service discovery, making them truly adaptive to service and network topology changes.

Techniques based on network multicast or broadcast provide an elegant discovery solution.

References

1. S. Vinoski, "Web Services Interaction Models—Part 1: Current Practice," *IEEE Internet Computing,* vol. 6, no. 3, May/June 2002, pp. 89–91.
2. Object Management Group (OMG), "Trading Object Service Specification," OMG document formal/00-06-27, version 1.0, May 2000, available at http://cgi.omg.org/docs/formal/00-06-27.pdf.
3. Organization for the Advancement of Structured Information Standards (OASIS), "Universal Discovery, Description, and Integration, Version 3.0," 19 July 2002, available at http://cgi.omg.org/docs/formal/00-06-27.pdf.
4. "Mac OS X v10.2 Technologies: Rendezvous," white paper, Apple Computer, Oct. 2002, available at www.apple.com/macosx/pdfs/Rendezvous_TB.pdf.

Steve Vinoski is vice president of platform technologies and chief architect for IONA Technologies. He is coauthor of *Advanced CORBA Programming with C++* (Addison Wesley Longman, 1999). Vinoski serves as IONA's alternate representative to the W3C's Web Services Architecture working group.