Using the ServiceDiscoveryManager

Topics in This Chapter

- Basics of the `ServiceDiscoveryManager`
- Common programming idioms
- Creating `LookupCaches`
- Soliciting events

Chapter 10

In the previous chapter, you learned how to use the `ServiceRegistrar` interface. This interface, common to all lookup service implementations, is the "lowest common denominator" API for accessing lookup functionality. You can count on the basic interfaces for `lookup()` and `register()` being available on every lookup service you encounter.

Now, in this chapter, you'll learn about how to use the `ServiceDiscoveryManager` class. This class is built using the "raw" `ServiceRegistrar` interface, and provides higher level functionality on top of it. In this chapter, you'll learn how to use the `ServiceDiscoveryManager` to maintain client-side caches of services, search for services, and receive event notifications when services appear, disappear, or are modified. While you can certainly do all of these functions directly using `ServiceRegistrar`, in most cases, the `ServiceDiscoveryManager` will provide a simpler programming model and more functionality.

I'll talk about the API provided by this important class, and cover the most common programming "idioms" that it supports. As you'll see, the `ServiceDiscoveryManager` is a complicated class that can support a number of styles of interaction. Beginners are often confused by the behavior of this class because they don't understand the different ways to use it, and how those programming styles interact. In this chapter, I'll cover each of these styles, and provide example programs that show you practical details about how to use them.

Higher Level APIs for Client Lookup

You've no doubt noticed that many of the clients you've seen so far exhibit many of the same behaviors—they discover lookup services, search for desired services, and solicit events to be informed of changes in the available services. In the examples in this book, I've typically done some bookkeeping to keep track of the lookup services and service proxies that I'm using. Certainly much of the `ServiceInfoWatcher` and `ServiceInfoSearcher` code from the last chapter is devoted to these sorts of chores.

There are a number of good reasons for this bookkeeping. First, by holding on to the proxies for lookup services, you don't have to reinitiate the costly discovery process if you need to do future queries or event solicitations. Second, by holding on to the services found at these lookup services, you can very quickly iterate over the available services, perhaps to present them to a user. Keeping information about the service IDs of lookup—and other—services allows you to recognize duplicate registrations, which is a useful thing to do.

Of course, not *all* clients will, or should, do this bookkeeping. Limited memory clients, for instance, may be willing to pay the time penalty required to query lookup services again and again, rather than keeping references to many service proxies around. But, many clients will follow the "cache and bookkeep" paradigm. (And, now is probably a good time to point out that many services will themselves be clients of other services. So, they are likely to follow many of these same patterns and have the same requirements for bookkeeping.) Sun realized this common behavior and created a class—called the `ServiceDiscoveryManager`—to support it in Jini 1.1.

The ServiceDiscoveryManager

The `ServiceDiscoveryManager` is an incredibly flexible class. It can serve as a "substitute" for using the raw `ServiceRegistrar` methods for doing lookup. It can also create a cache of services based on some search criteria. This cache can be polled by a client application that wishes to determine what services are available, and the cache can also deliver events to the client when a service is added, removed, or changed. One of its benefits is that it largely hides the entire notion of lookup services and the `ServiceRegistrar` API. You only deal with services—the `ServiceDiscoveryManager` takes care of finding lookup services, eliminating duplicate services, and so on. Most client applications written to Jini 1.1 or later will use the `ServiceDiscoveryManager` class as their sole interface for service location.

Here is the declaration of the class:

```
package net.jini.lookup;

public class ServiceDiscoveryManager {
    public ServiceDiscoveryManager(
                         DiscoveryManagement discoveryMgr,
                         LeaseRenewalManager leaseMgr)
        throws IOException;

    public LookupCache createLookupCache(
                        ServiceTemplate tmpl,
                        ServiceItemFilter filter,
                        ServiceDiscoveryListener listener)
        throws RemoteException;

    public ServiceItem lookup(ServiceTemplate tmpl,
                             ServiceItemFilter filter);
    public ServiceItem lookup(ServiceTemplate tmpl,
                             ServiceItemFilter filter,
                             long waitDur)
        throws InterruptedException, RemoteException;
    public ServiceItem[] lookup(ServiceTemplate tmpl,
                               int maxMatches,
                               ServiceItemFilter filter);
    public ServiceItem[] lookup(ServiceTemplate tmpl,
                               int minMatches,
                               int maxMatches,
                               ServiceItemFilter filter,
                               long waitDur)
        throws InterruptedException, RemoteException;

    public DiscoveryManagement getDiscoveryManager();
    public LeaseRenewalManager getLeaseRenewalManager();
    public void terminate();
}
```

I'll cover how the constructor for this class works before jumping in to the other methods on the class and how they're used.

The constructor for ServiceDiscoveryManager takes as arguments an object implementing the DiscoveryManagement interface, and a LeaseRenewalManager. The semantics are much the same as for the service-side JoinManager: You can pass in any object that implements DiscoveryManagement (typically a LookupDiscoveryManager) to have control over the discovery process. You can also pass in your own `LeaseRenewalManager` if you wish to reuse an instance of this class that you may already have "sitting around" in your program. If you pass null for either of these parameters, a default implementation will be created (a `LookupDiscoveryManager` initialized to find lookup services in the public group, and a "fresh" `LeaseRenewalManager`).

Notice that the constructor is declared as raising `IOException`. This is because the constructor may have to initiate the multicast discovery protocols, which can cause this exception.

A Word on Usage Patterns

The `ServiceDiscoveryManager` is a commonly misunderstood class, not only because it provides a lot of functionality, but because it supports three broad patterns of usage. Before I continue with the description of the other methods in the class, I should say a few words about these patterns, how they work, and how they relate to one another.

In the first pattern, there is no need to create a `LookupCache` at all. In this style of use, you only need to create a `ServiceDiscoveryManager` and then invoke whichever version of `lookup()`—on the `ServiceDiscoveryManager` itself—that is applicable to the needs of your application.

In the other two usage patterns, after creating a `ServiceDiscoveryManager`, you use that object to create one or more `LookupCaches`. From that point on, you *typically* interact only with the `LookupCache`, not the `ServiceDiscoveryManager` that you initially created. Of course, you are free to invoke methods on `ServiceDiscoveryManager` at any time, even if you've created a `LookupCache`, although this style of programming isn't common in the two patterns that involve `LookupCaches`.

Once a `LookupCache` has been created, if you need to query for a particular service, you will typically invoke one of the versions of `lookup()` provided by the `LookupCache`, not the `ServiceDiscoveryManager`. This is the second usage pattern.

The third and final usage pattern involves the event mechanisms of the `LookupCache`. That is, if you wish to be notified of the arrival, departure, or modification of services, you must register for events with the `LookupCache`, not with the `ServiceDiscoveryManger` or the underlying `ServiceRegistrar`.

These three patterns will become more clear as I walk through the rest of the methods on the class. But it's important to keep these in mind in the discussions below, so that you can see the "big picture."

Creating Lookup Caches

Perhaps the most important method on this class, because it's used in the two most common usage patterns, is `createLookupCache()`. Clients use this method to instruct the `ServiceDiscoveryManager` to create a cache of all of the available services that match some search criteria. The `ServiceDiscoveryManager` will discover lookup services and then search them to find the desired services. It will also solicit events from these lookup services to ensure that the cache is kept relatively up-to-date—as new matching services appear in the community, they will also appear in the cache.

The cache that is returned by this method is a local repository of all matching services, and is "back-filled" by the `ServiceDiscoveryManager` (meaning that the set of services it contains may change, even after you've acquired a reference to the cache). You should be sure that you understand this point: The proxies and attributes of any matching services will be stored locally by the cache. So, if your client is running in a memory-constrained environment, you should take great care when using caches.

Using methods that you'll see below, you can use the cache to return all matching services, and can ask the cache to inform you about changes in the status of matched services.

The `createLookupCache()` method takes three arguments. The first is an instance of `ServiceTemplate`, which works just as in the earlier examples of the previous chapter: The template is used to query lookup services to find desired services, using the normal template-matching semantics. The second argument is a `ServiceItemFilter`. This is a *client-side* filtering mechanism that lets you have fine-grained control over which services show up in the cache.

I talked in the last chapter about how `ServiceTemplates`, while powerful and simple-to-use, don't always give you all of the control over service matching that you may like. They don't, for instance, allow you to do any comparisons over numeric attributes other than simple equality. The `ServiceItemFilter` used here allows you greater control over matching. Any `ServiceItems` that match the template are returned to the client where they are evaluated by the filter. The filter has a method that can return true or false to indicate whether the service is considered a match or not.

Core Tip: Optimizing your searches

Be sure you understand the relationship between `ServiceTemplates` *and* `ServiceItemFilters`. `ServiceTemplates` *are sent "over the wire" to lookup services. The matching of services against* `ServiceTemplates` *happens inside the* *lookup service, using the fast (but relatively coarse) rules I outlined before. Any matched services are then returned to the client, where* `ServiceItemFilters`*—which execute completely in the client—have an opportunity to further prune the results.*

Usually, you will look for a design that partitions a search between these two classes in a way that minimizes the amount of data returned from the lookup service. Since sending a lot of data over a network is a relatively expensive operation, you should use `ServiceTemplates` *that, as closely as possible, cull out only the services that you're interested in. Once this small set of possibly interesting services is returned, you can use* `ServiceFilters` *to refine the result set.*

The final argument to the constructor is an object that implements the `ServiceDiscoveryListener` interface. This argument allows you to supply a listener to the cache that will be called whenever the cache updates its state to reflect the addition, removal, or change of a service. The cache will deliver a `ServiceDiscoveryEvent` that encapsulates this information. Unlike `ServiceEvents`, which are sent from lookup services to reflect their changes in state, `ServiceDiscoveryEvents` are purely *local* events; that is, they originate in the cache in the client's VM, rather than some remote VM.

There are some other important differences between `ServiceDiscoveryEvents` and `ServiceEvents` in addition to just the local versus remote distinction. In particular, the cache will try to *coalesce* multiple remote `ServiceEvents` into a single `ServiceDiscoveryEvent`. I'll talk more about this in the section on `ServiceDiscoveryListeners` a bit later.

I'll continue walking through the rest of the methods in `ServiceDiscoveryManager`, and then talk about the supporting classes such as `LookupCache`, `ServiceDiscoveryEvent`, and so on.

New Versions of Lookup()

The `ServiceDiscoveryManager`, in addition to supporting the ability to create caches of services, also provides "wrapper" implementations of the lookup method that appears in the `ServiceRegistrar` interface. I call these implementations "wrappers" because they allow clients to essentially search a whole *set* of lookup services at once. That is, while the `ServiceRegistrar` version searches only one particular lookup service, these versions are "front ends" that provide a slightly richer API and can search all of the lookup services that have been discovered so far.

All versions of the `lookup()` method take a `ServiceTemplate` and a `ServiceItemFilter` as arguments; these parameters are used in exactly the same way as described above. The `ServiceDiscoveryManager` uses the template to get an initial set of matching services from the lookup services it has discovered, and then uses the filter to further cull this list.

The first two versions of `lookup()` return only a single `ServiceItem`. If the template and filter together match multiple services, an arbitrary one is selected and returned. If no matching services are available at the lookup services (that is, if the template query returns no matches), then the filter will not be run and a null will be returned to the caller.

The second two versions return an array of all matching service items. Note that these versions of `lookup()` take extra parameters describing how many matches should be returned. If no matches are found, then an empty array will be returned to callers.

One single-valued version of `lookup()` and one multi-valued version of `lookup()` are also defined to be *blocking*. This means that they will wait for some specified amount of time until the requested number of matches is found. The versions of `lookup()` that take a wait duration parameter have this behavior.

If the wait duration expires without the requested number of services being found (either one service for the single-valued version, or `minMatches` for the multi-valued version), then the blocking methods will return an empty result. Otherwise, they complete successfully and return the service items they have found.

Contrast this to the non-blocking versions (the versions *without* the wait duration parameter), which return a result immediately. Be aware that if these versions of `lookup()` report that no matching services have been found, it may be because there actually are no matching services in the community, or it may be because no lookup services have been discovered yet.

This is a common mistake when using this class! Callers create a `ServiceDiscoveryManager` and immediately use the non-blocking version of `lookup()`, which will often return null. The error isn't that the service isn't available—it's just that the lookup service holding it may not have been discovered yet.

Note that the blocking methods may raise `java.lang.InterruptedException`. This is because, sometimes, a caller may wish to interrupt a thread that is blocked executing one of these methods. If you interrupt a thread in this way (by calling `Thread.interrupt()`), then the method will raise an `InterruptedException`.

The blocking methods provide you with an easy-to-use way to allot a fixed period of time to try to find a service.

Core Tip: Know when to use lookup(), and when to use a cache

The cache feature of `ServiceDiscoveryManager` *is typically used by clients that need frequent access to a whole set of services (such as browsers), or clients that are running on high-resource platforms.*

The `lookup()` *method, on the other hand, is most commonly used by clients that need access to only one (or a few) particular services, or by clients running in resource-constrained environments.*

Miscellaneous Methods

After the `lookup()` methods, there are a handful of useful utility methods on `ServiceDiscoveryManager`. The `getDiscoveryManager()` and `getLeaseRenewalManager()` methods return the `DiscoveryManagement` object and `LeaseRenewalManager` object in use, respectively. If you explicitly passed in instances in the constructor, these methods will return these same objects; otherwise, they will return the implicit objects created by the `ServiceDiscoveryManager`.

The `terminate()` method is used to shut down the `ServiceDiscoveryManager`. Specifically, this means that all `DiscoveryListeners` and `RemoteEventListeners` will be removed, and any leases on event registrations will be cancelled. All internal threads will also be stopped.

Much as in the case of `JoinManager`, calling `terminate()` is an irreversible operation. Calling any method on `ServiceDiscoveryManager` after calling `terminate()` will result in a `RuntimeException.`

Receiving Remote Events

You've no doubt noticed that some of the methods of this class raise `RemoteExceptions` while others do not. This is because the methods that raise `RemoteExceptions` (including `createLookupCache()` and the blocking versions of `lookup()`) register remote event listeners with lookup services so that they can be informed of changes in the set of available services.

This fact isn't a mere curiosity of the implementation, though—it puts requirements on you to ensure that your client program correctly exports the listener code that lookup services will use to call back to the `ServiceDiscoveryManager`.

Remember that remote events are delivered from one JVM to a listener in another. The actual mechanics of how this happens are that the listener is defined as an RMI remote object, typically extending `java.rmi.Unicast-RemoteOb-`

`ject`. The *stub* for this object is automatically downloaded into the caller—the program that will *generate* the remote event—so that it can know how to connect back to the receiving listener.

Just like any remote event listener, `ServiceDiscoveryManager` comes with a listener class that will receive events, and an `rmic`-generated stub class that is meant to be downloaded into callers. Even though the stub comes as a part of the standard Jini client libraries, *it is your responsibility* to provide the facilities for this stub to be downloaded by lookup services.

The common way to do this is to extract the stub class along with the classes it uses from the Jini JAR files and bundle them into a separate JAR file that will be served by the client's HTTP server. You can, of course, bundle these into the JAR file that contains any other client-exported code so that you have everything in one place. After doing this, you must set a codebase that tells the lookup service where the needed code can be found.

The stub class is called `net.jini.lookup.ServiceDiscoveryManager$LookupCacheImpl$LookupListener_Stub.class`, and it lives in `jini-ext.jar`. This class depends only on `net.jini.core.event.RemoteEventListener.class`, from `jini-core.jar`. While technically you don't need to export the class file for the `RemoteEventListener` interface—because any lookup service should already have it—it's a good idea to keep all necessary code together, just in case.

Remember that you *must* make some provision for exporting these class files, or the methods on `ServiceDiscoveryManager` that use events—which are the most powerful and useful methods—will not be useable by you!

In the interest of keeping the descriptions of how to run the example programs short, I will take the expedient approach of simply placing these class files in the client's downloadable code directory, where they will be served by the "normal" HTTP server that exports the rest of the client's code. I would recommend producing a single JAR file of all downloadable code exported by the client for a "production" system, though.

Supporting Classes

As you saw above, there are a number of supporting classes used alongside the `ServiceDiscoveryManager`.

LookupCache

For many clients, the most common idiom of using the `ServiceDiscoveryManager` will be to create one or more `LookupCaches`, each of which reflects a different pool of services that the client is interested in. `LookupCache` is actually an interface, which allows future implementations of `ServiceDiscoveryManager` to provide different sorts of caches, perhaps with different performance characteristics.

The cache provides a handful of methods that allows clients to retrieve services out of the cache, and ask the cache to notify them upon changes.

```
package net.jini.lookup;

public interface LookupCache {
    public ServiceItem lookup(ServiceItemFilter filter);
    public ServiceItem[] lookup(ServiceItemFilter filter,
                                int maxMatches);

    public void addListener(ServiceDiscoveryListener l);
    public void removeListener(ServiceDiscoveryListener l);

    public void discard(Object serviceReference);
    public void terminate();
}
```

The `lookup()` methods work similarly to the versions in `ServiceDiscoveryManager` that you saw before, with a couple of important differences. First, these methods do not take the `ServiceTemplate` parameter. This is because they do not actually cause any network traffic to go between the client and the lookup service. Instead, the query is answered completely from the services that happen to be contained in the cache at the time the call to `lookup()` is made. Recall that the contents of the cache will eventually be all of the services that match both the template and the filter specified when the cache was set up (I say "eventually" because it may take some time for the cache to fill). The `ServiceItemFilter` provided here allows you to further refine the services you select from the cache.

A second difference is that both versions of `lookup()` here are non-blocking. The first will return a matching `ServiceItem` if it exists in the cache, or null otherwise, but will return immediately. Likewise, the second will return a matching array of `ServiceItems`, or an empty array if no matches exist, but will still return immediately. Again, this behavior is because both of these methods involve no remote calls—they go directly to the cache and try to satisfy the requests based on the information available locally.

While the `lookup()` calls allow you to "poll" the cache to find out if desired services are available, sometimes you may want the cache to asynchronously notify you when services are available. Thus, the `LookupCache` provides a way for you to install and remove listeners for service-related events. The `addListener()` and `removeListener()` methods let you install and uninstall `ServiceDiscoveryListeners` that will be called when services are added, removed, or changed. (See below for details on `ServiceDiscoveryListeners` and the `ServiceDiscoveryEvents` that they will receive.)

Finally, the last two methods are used for housekeeping in the cache. The `terminate()` method simply shuts down all of the activities of the `LookupCache`: It halts all threads and cancels any event registrations with lookup services. This method is irreversible, and is typically called when the client is no longer interested in the contents of the cache.

The `discard()` method is used to drop a service from the cache. The argument here should be the service's proxy object. Once a proxy has been dropped, all references to it in the cache are removed, and the cache's `ServiceDiscoveryListeners` will be notified. This operation is often done if a service seems to have failed or has become unreachable—you will usually detect this because the service's proxy will begin to raise `RemoteExceptions`. Discarding the service means that the service may be rediscovered later, if it recovers and registers itself with lookup services.

Core Note: Understanding the semantics of discard

As mentioned, calling `discard()` *causes a reference to a service to be dropped from the cache so that it can be rediscovered. Generally, the only time you will explicitly discard a service is when it seems to have failed—meaning that attempts to use its proxy result in* `RemoteException` *being thrown.*

If the service has genuinely crashed or been shut down then, over time, its registrations with lookup services will expire. When the service comes back online, it will reregister with these lookup services, the lookup cache will detect this fact, and the service will be rediscovered and added to the cache once again. Any listeners registered with the cache will be notified.

There is, however, an insidious condition which the `LookupCache` *must deal with. What happens if the service hasn't actually crashed, but merely become unreachable because of some network partition? In this case, your attempts to use it will certainly result in*

`RemoteExceptions` *being raised. But it may be possible, depending on the partition, that the service is still in perfect communication with all of the lookup services with which it is registered!*

In this case, the service will never "reappear" in the lookup service, since it will never have left it. Does this mean that the `LookupCache` *will never rediscover the service, and that it is lost forever?*

Fortunately, no. The algorithm that the `LookupCache` *uses to determine when a service is to be rediscovered is a bit more complex than you might first think; it can be helpful to understand how this works in case you're debugging problems with service discovery.*

When you first call `discard()` *on a service, the* `LookupCache` *removes the reference to the service from the cache. But it also keeps a copy of this reference in a separate "limbo" storage area. It then waits for a some amount of time to pass; typically this will be some span that is longer than the typical lease duration with a lookup service. If, during this interval, the service disappears from the lookup services, then the cache assumes that the service has in fact crashed, and removes its reference from the limbo area. If the service returns later, it will then be rediscovered.*

However, if this timeout elapses and the service is still registered with the lookup services, the cache assumes that the service has not actually crashed—since it is apparently renewing its leases with the lookup services. In this case, the service is then "rediscovered" by the cache, and its reference is moved from limbo back into the cache's regular storage space. Any listeners are informed of this fact. This mechanism provides a way for the cache to recover from network partitions and "rediscover" inaccessible services, even if those services have never left the community.

The timeout period can be controlled by the property `com.sun.jini.sdm.discardWait`*, and is set to 10 minutes by default (which is twice the maximum lease granted by reggie).*

ServiceItemFilter

The `ServiceItemFilter` interface provides a way to do client-side "filtering" of service query results. The basic idea is that you provide a class that implements the `ServiceItemFilter` interface to calls that create caches or do a `lookup()`. Any results returned by matching a `ServiceTemplate` are returned to your client, where your filter gets a chance to veto whether a service should be returned.

```
package net.jini.lookup;

public interface ServiceItemFilter {
    public boolean check(ServiceItem item);
}
```

The `check()` method will be called to evaluate whether a service is considered to match the filter. The filter should return true if the service matches, or false otherwise.

Filtering provides you with a way to do searches that are impossible using the standard `ServiceTemplate` semantics. For example, you can write filters that apply numerical comparison tests to attributes.

ServiceDiscoveryListener

The `ServiceDiscoveryListener` interface is used by clients that wish to be asynchronously notified of changes in the available services known to a `LookupCache`. You can install a `ServiceDiscoveryListener` when you first create a cache via a call to `createLookupCache()`, and you can also add listeners after the fact by calling `addListener()` directly on the `LookupCache`.

The interface itself is fairly simple:

```
package net.jini.lookup;

public interface ServiceDiscoveryListener {
    public void serviceAdded(ServiceDiscoveryEvent event);
    public void serviceChanged(ServiceDiscoveryEvent event);
    public void serviceRemoved(ServiceDiscoveryEvent event);
}
```

In many ways, the information available through this interface is analogous to the information available through the low-level `ServiceEvents` that are sent from lookup services. The primary differences are that the `ServiceDiscoveryListener` methods are locally invoked from the cache (rather than remotely by the lookup service), and correspond to changes in the cache rather than just one particular lookup service. This means that the `LookupCache` will try to *coalesce* multiple `ServiceEvents` into single `ServiceDiscoveryEvents`.

Why is this? Think about how services are registered in Jini. Typically, a service will start up and join any and all lookup services that it finds that are “relevant” (meaning that they are members of a group, or named by a locator, that the service is searching for). So any given service will usually be registered with *many* lookup services. The `LookupCache` will register with *each* of these to receive notifications in changes in the set of services each provides.

But of course, when you’re interested in a service, you typically just want to acquire *a* reference to it, not *all* references to it. So the `LookupCache`, even though it receives multiple `ServiceEvents` about a given service, will typically generate only a single corresponding event to its listeners.

This is done in the following way:

- The `serviceAdded()` method is called when a cache receives notification that a service has been registered *for the first time* in a community. This means that even if a service registers itself with multiple lookup services, the cache will notice these multiple registrations, determine that the multiple registrations are for the same service, and generate only one invocation of `serviceAdded()`.
- The `serviceRemoved()` method is called when a service disappears from *all* of the lookup services that it knows about. The method will not be called if a service is dropped merely from one of a set of lookup services with which it is registered.
- The `serviceChanged()` method is invoked once for each *distinct* change in attributes it detects. That is, the cache keeps one “canonical” set of attributes associated with each service in the cache. When the attributes change at one lookup service, the cache updates its record of the service’s attributes and generates an event. Future notifications of updates do not generate events, if those updates produce attributes that are the same as those already known to the cache. Typically, when a service changes its attributes, it does so by updating each lookup service with which it is registered. The first such change will be detected by the cache and will result in an event. As the other lookup services are updated to the same attribute set, further events are not generated by the cache.
- Finally, the `serviceChanged()` method is invoked when a unique change occurs in the service’s proxy object.

Be sure to note that the `serviceChanged()` method, in addition to being called when an attribute on a service changes, will also be called when the *proxy* for a service changes. This may happen if a service updates or revises its published proxy. But, two proxies *for the same service* that are registered at different lookup services may not be "equal" if you don't take special precautions. If you write your own services, you should be aware that proxies that have inconsistent equality comparisons with each other can cause the `LookupCache` to report that a proxy has changed.

Core Alert: Controlling proxy changes

As a service writer, you will often want to have control over whether the proxies for your service are considered equal to, or different from, each other. The best way to do this—and a good rule of thumb to follow in any case—is to override the `equals()` *and* `hashCode()` *methods on your service proxies.*

Think about what happens if your service registers itself at multiple lookup services. Each registration will result in a new copy of the proxy being stored at each of those lookup services. If your service doesn't override `equals()` *and* `hashCode()`*, you've effectively given up control over how the* `LookupCache` *will determine if these proxies are the same or not. And, without doing anything explicit, the default implementation of these methods is likely to report that each proxy is different from all others, resulting in many* `serviceChanged()` *invocations.*

(If your proxies are simply RMI stubs, then you'll get the correct behavior—since stubs override `equals()` *and* `hashCode()` *to be based on whether the remote objects referred to by the stubs are equal to each other. If, as is commonly the case, you're using smart proxies that contain a single remote reference to a back-end server, a common implementation for* `equals()` *is to simply invoke the* `equals()` *method on the remote reference.)*

The rule of thumb is to always provide "smart" implementations of these two methods on your service proxies. Remember that if two objects are equal to each other, they must return the same hash code.

ServiceDiscoveryEvent

The `ServiceDiscoveryEvent` encapsulates the information about service changes, additions, and removals that the `LookupCache` produces.

```
package net.jini.lookup;

public class ServiceDiscoveryEvent
    extends java.util.EventObject {
```

```
    // ... constructor elided ...
    public ServiceItem getPostEventServiceItem();
    public ServiceItem getPreEventServiceItem();
}
```

The declaration here doesn't show the constructor for the class, since it's only of interest to the implementors of the `LookupCache` class. The two methods of interest here are `getPostEventServiceItem()`, which returns the `ServiceItem` for the changed service *after* the change has taken place, and `getPreEventServiceItem()`, which returns the `ServiceItem` for the changed service *before* the change has taken place.

If the event is being sent because a service was removed, then `getPostEventServiceItem()` will return null. Likewise, if the event is sent because a service was added, `getPreEventServiceItem()` will return null. If the service merely changed (it wasn't added or removed), then neither will return null—the pre-event method returns the state before the change and the post-event method returns the state after the change.

Neither of these methods actually copy the cached `ServiceItem` before returning it, since this can be a potentially expensive operation. So you should take care to not modify the `ServiceItems` returned by these methods, or you will seriously corrupt your `LookupCache`.

The only other method of interest on this class is `getSource()`, which is inherited from `java.util.EventObject`. This method will return the `LookupCache` that generated the event.

Core Tip: Troubleshooting multiple service events

One common problem symptom when you use the event facilities in the `ServiceDiscoveryManager` *is that you may receive mulitple, repeated* `ServiceDiscoveryEvents` *for the same service. Often these will show the service constantly appearing and disappearing rapidly.*

Such problems are typically the result of one of two errors. The first, and easiest to fix, is that you are not correctly exporting the remote event listener that the `ServiceDiscoveryManager` *uses to receive events. Without this working, the* `ServiceDiscoveryManager` *will be unable to correctly determine when services come or go after initial discovery time See the section, Receiving Remote Events for details.*

The second, and somewhat more insidious cause, results from services that do not properly override `equals()` *and* `hashCode()`

on their proxies. This means that each proxy registration is likely to look like a new and different proxy version to the `ServiceDiscoveryManager`*, resulting in unnecessary events. See the section on* `ServiceDiscoveryListener` *for details.*

If you're experience this symptom, check both of these potential causes.

A Basic Example

In this example, you'll see how to use the `ServiceDiscoveryManager` in its most straightforward settings. The example demonstrates how to use both the blocking and non-blocking forms of lookup, as well as the `LookupCache`. You should pay special attention to the output of this program! Misunderstanding how the `ServiceDiscoveryManager` reports its results is a common source of errors.

Look at the code in Listing 10-1:

Listing 10–1 ClientLookupExample.java

```
// Explore the ServiceDiscoveryManager

package corejini.chapter10;

import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.lookup.LookupCache;
import net.jini.lookup.ServiceItemFilter;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lease.LeaseRenewalManager;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import corejini.chapter9.ServiceInfoSearcher;

public class ClientLookupExample  {
    protected LookupDiscoveryManager discoveryMgr;
    protected LeaseRenewalManager leaseMgr;
    protected ServiceDiscoveryManager lookupMgr;

    public ClientLookupExample() throws IOException {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(
                    new RMISecurityManager());
        }

        lookupMgr =
            new ServiceDiscoveryManager(null, null);
    }
```

Listing 10–1 ClientLookupExample.java (continued)

```
    // Subclasses may override...
    public ServiceDiscoveryListener getListener() {
        return null;
    }

    // subclasses may override...
    public ServiceItemFilter getFilter() {
        return null;
    }

    // Do a bit of work here...
    public void runTests() {
        ServiceTemplate tmpl =
             new ServiceTemplate(null, null, null);
        ServiceItem service = null;

        service = lookupMgr.lookup(tmpl, null);

        if (service == null) {
            System.out.println("Non-blocking lookup " +
                               found no services.");
        } else {
           System.out.print("Non-blocking lookup found: ");
            ServiceInfoSearcher.printServiceInfo(service);
        }

        try {
            service = lookupMgr.lookup(tmpl, null, 20000);
        } catch (RemoteException ex) {
            System.out.println("Error: " + ex);
        } catch (InterruptedException ex) {
        }

        if (service == null) {
            System.out.println("Blocking lookup found " +
                               "no services.");
        } else {
            System.out.print("Blocking lookup found: ");
            ServiceInfoSearcher.printServiceInfo(service);
        }

        LookupCache cache = null;
```

Listing 10–1 ClientLookupExample.java (continued)

```
        try {
            cache = lookupMgr.createLookupCache(tmpl,
                                               getFilter(),
                                               getListener());
        } catch (RemoteException ex) {
            System.out.println("Error: " + ex);
        }

        service = cache.lookup(null);

        if (service == null) {
            System.out.println("Cache lookup found " +
                               "no services.");
        } else {
            System.out.print("Cache lookup found: ");
            ServiceInfoSearcher.printServiceInfo(service);
        }

        System.out.println("Pausing...");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
        }

        System.out.println("Trying again...");
        service = cache.lookup(null);
        if (service == null) {
            System.out.println("Cache lookup found " +
                               "no services.");
        } else {
            System.out.print("Cache lookup found: ");
            ServiceInfoSearcher.printServiceInfo(service);
        }

        System.out.println("Getting all cached services.");
        ServiceItem[] services = cache.lookup(null,
                                           Integer.MAX_VALUE);
```

Listing 10–1 ClientLookupExample.java (continued)

```
        if (services == null || services.length == 0) {
            System.out.println("No services in cache");
        } else {
            for (int i=0 ; i<services.length ; i++) {
                System.out.print("[" + i + "] ");
                ServiceInfoSearcher.printServiceInfo(
                                             services[i]);
            }
        }

        cache.terminate();
    }

    public static void main(String[] args) {
        try {
            ClientLookupExample cle =
                new ClientLookupExample();
            cle.runTests();
        } catch (IOException ex) {
            System.err.println(ex.toString());
        }

        System.exit(0);
    }
}
```

The `main()` for this example creates an instance of `ClientLookupExample` and then calls `runTests()`, which exercises the `ServiceDiscoveryManager`. Note that the constructor for the example installs a security manager—which it must do in order to download code, just the same as all the rest of the examples in this book—and creates a new `ServiceDiscoveryManager`, passing in nulls as arguments. This causes the `ServiceDiscoveryManager` to create its own `LookupDiscoveryManager` and `LeaseRenewalManager`.

Most of the work here happens in `runTests()`. The code exercises the `ServiceDiscoveryManager` in several ways. First, it calls `lookup()` directly, using the non-blocking APIs. In this example, the program always uses a template that matches all services. After this, it tries the blocking version of `lookup()` with a wait duration of 20000 milliseconds (20 seconds). It then creates a `LookupCache`, and invokes `lookup()` on it to fetch a service. Note that the code calls `getFilter()` and `getListener()` to get a `ServiceItemFil-`

ter and a `ServiceDiscoveryListener` to use when creating the cache. In this example, these methods always return null; later examples will override these to test filtering and event processing. The code then sleeps for a few seconds, does another lookup, and—finally—prints out the entire contents of the cache. Any services that are found are printed by calling the `printServiceInfo()` method from the earlier `ServiceInfoSearcher` program.

While this example may seem to be going all over the map, invoking `lookup()` over and over again, first on the `ServiceDiscoveryManager` and then on the `LookupCache`, it's important to understand the different behaviors that these calls can produce. So, the next step is to move on and compile the example and see what it does.

Compiling and Running the Example

Compile this example using the standard operating procedure. Remember that the client will need to export downloadable code to services. For this to work, you must ensure that you're running an HTTP server to export the client's downloadable code. Refer back to the client examples in Chapter 5 if you haven't started this Web server.

On Windows:

```
javac -classpath C:\files;
                 C:\jini1_1\lib\jini-core.jar;
                 C:\jini1_1\lib\jini-ext.jar;
                 C:\jini1_1\lib\sun-util.jar;
                 C:\client
    -d C:\client
C:\files\corejini\chapter10\ClientLookupExample.java
```

The one extra step you must take is to ensure that the class files for the `ServiceDiscoveryManager`'s remote event listener are also exported correctly! Here, I'm taking the quick-and-dirty approach of extracting the needed files from the Jini JARs directly into the client's download directory. This makes them available from the same codebase, and uses the same HTTP server, as the rest of the client's code.

The syntax to extract a single file from a JAR is `jar xvf <jar_file> <file_to_extract>`. Make sure you're in the `client-dl` directory when you run the extraction commands.

```
cd C:\client-dl
jar xvf C:\jini1_1\lib\jini-ext.jar net\jini\lookup\Service-
DiscoveryManager$LookupCacheImpl$LookupListener_Stub.class
```

```
jar xvf C:\jini1_1\lib\jini-core.jar
   net\jini\core\event\RemoteEventListener.class
```

If you look at the contents of the `client-dl` directory, you should see that `jar` created a `net` directory, as well as subdirectories, to contain the class files.

```
java -cp C:\jini1_1\lib\jini-core.jar;
        C:\jini1_1\lib\jini-ext.jar;
        C:\jini1_1\lib\sun-util.jar;
        C:\client
    -Djava.security.policy=C:\files\policy
    -Djava.rmi.server.codebase=http://myhost:8086/
corejini.chapter10.ClientLookupExample
```

On UNIX:

```
javac -classpath /files:
                /files/jini1_1/lib/jini-core.jar:
                /files/jini1_1/lib/jini-ext.jar:
                /files/jini1_1/lib/sun-util.jar:
                /files/client
    -d /files/client
/files/corejini/chapter8/ClientLookupExample.java
  C:\files\corejini\chapter10\ClientLookupExample.java
```

The one extra step you must take is to ensure that the class files for the `ServiceDiscoveryManager`'s remote event listener are also exported correctly! Here, I'm taking the quick-and-dirty approach of extracting the needed files from the Jini JARs directly into the client's download directory. This makes them available from the same codebase, and uses the same HTTP server, as the rest of the client's code.

The syntax to extract a single file from a JAR is `jar xvf <jar_file> <file_to_extract>`. Make sure you're in the `client-dl` directory when you run the extraction commands. On most UNIX shells, you will have to preface the dollar sign character by a backslash to "escape" it.

```
cd /files/client-dl
jar xvf /files/jini1_1/lib/jini-ext.jar net/jini/lookup/Ser-
viceDiscoveryManager\$LookupCache-
Impl\$LookupListener_Stub.class
jar xvf /files/jini1_1/lib/jini-core.jar
   net/jini/core/event/RemoteEventListener.class
```

If you look at the contents of the `client-dl` directory, you should see that `jar` created a `net` directory, as well as subdirectories, to contain the class files.

```
java -cp /files/jini1_1/lib/jini-core.jar:
         /files/jini1_1/lib/jini-ext.jar:
         /files/jini1_1/lib/sun-util.jar:
         /files/client
     -Djava.security.policy=/files/policy
     -Djava.rmi.server.codebase=http://myhost:8086/
corejini.chapter10.ClientLookupExample
```

The results that you see will actually vary depending on the performance of the machine you run the example on, and the performance of the machines running lookup services in your community. Here's a sample run, though:

```
  Non-blocking lookup found no services.
  Blocking lookup found:   Name = Lookup
   Manufacturer = Sun Microsystems, Inc.
   Vendor = Sun Microsystems, Inc.
   Version = 1.1 alpha
   Model =
   Serial Number =
   Proxy is com.sun.jini.reggie.RegistrarProxy@af330156
  Cache lookup found no services.
  Pausing...
  Trying again...
  Cache lookup found:   Name = Address Book Service
   Manufacturer = Xerox PARC
   Vendor = Xerox PARC
   Version = 1.0
   Model = null
   Serial Number = null
   Proxy is com.xerox.dispatch.addressbook.AddressBookServiceProxy@0
  Getting all cached services...
  [0]   Name = Address Book Service
   Manufacturer = Xerox PARC
   Vendor = Xerox PARC
   Version = 1.0
   Model =
   Serial Number =
  Proxy is com.xerox.dispatch.addressbook.AddressBookServiceProxy@0
  [1]   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@f16dfd46
[2]   Name = Authentication Service
  Manufacturer = Xerox PARC
  Vendor = Xerox PARC
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.xerox.dispatch.authenticator.AuthenticatorProxy@0
[3]   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.1 alpha
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@af330156
```

Note that, here, the first call to `lookup()` returns null while the second finds a valid service! Remember that invoking `lookup()` on the `ServiceDiscoveryManager` only searches for services on the lookup services that have been discovered so far. In this run, the first call to `lookup()` happens before any lookup services have been found, so it returns null. The second blocks for a few seconds, and then gives a chance for some lookup services to be discovered, so it returns a service. These two similar calls produce different results, and you should make sure you understand why.

After this, the program creates and exercises the `LookupCache`. The first operation is to call `lookup()` on this cache. In the example run you see here, notice that the cache returns no service. This is because calling `lookup()` on the cache only returns results that have already been stored in the cache. In this case, the cache has not yet been filled with any services and so nothing is returned. After sleeping for a bit, the next call to `lookup()` returns a service. Again, it's important to notice the differences here. Even though prior calls to `lookup()` on the `ServiceDiscoveryManager` itself may return results, here the call to `lookup()` on the cache returns nothing. A later call, made after a pause, returns a service because the cache has had a chance to fill.

Finally, the example dumps out all of the services in the cache; in this example, there are several—they've just taken a few moments to show up.

Be sure you understand the differences in behavior between the "raw" `ServiceRegistrar` version of `lookup()` and the versions here. The versions here have timing dependencies that do not exist in the `ServiceRegistrar` version—and these timing dependencies are a common source of confusion when using the `ServiceDiscoveryManager`. Even though you must understand these differences, most users will still prefer to use the `LookupCache`, since it provides so many benefits (the ability to search for services across many lookup services, the fact that it essentially "hides" the existence of lookup services at all, the ability to run client-side filtering predicates, and so on).

Using Filters

Listing 10-2 demonstrates how to use `ServiceItemFilters`. In this program, you'll see three filters, and learn how to combine them together to extract specific services in ways that are impossible using just `ServiceTemplates`.

The first filter shows how you can do substring matching of attributes. This filter allows you to find all services that contain a given string in the `vendor` field of their `ServiceInfo` attributes.

Listing 10–2 ServiceInfoVendorFilter.java

```
// A filter that matches services whose ServiceInfo.vendor
// contains a given substring.

package corejini.chapter10;

import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceItemFilter;
import net.jini.lookup.entry.ServiceInfo;
import net.jini.core.entry.Entry;

public class ServiceInfoVendorFilter
        implements ServiceItemFilter {
    protected String substring;

    public ServiceInfoVendorFilter(String substring) {
        this.substring = substring;
    }

    public boolean check(ServiceItem item) {
        Entry[] attrs = item.attributeSets;
        for (int i=0 ; i<attrs.length ; i++) {
            // If it's a ServiceInfo or subclass
            if (ServiceInfo.class.isAssignableFrom(
                                   attrs[i].getClass())) {
                return ((ServiceInfo)
                 attrs[i]).vendor.indexOf(substring) != -1;
            }
        }

        return false;
    }
}
```

The `ServiceInfoVendorFilter` is pretty simple. Instances of the class are constructed with a string to search for. The `check()` method iterates over the attribute set on the service item, looking for `ServiceInfo` attributes or its subclasses. When one is found, the method simply looks to see if the parameter occurs as a substring in the vendor string. If it does, the method returns true.

The second filter is a little more complicated. It lets you match services that have a certain version number in the `version` fields of their `ServiceInfo` attributes. Using this filter, you can find services that are exactly at a given version (say, version "1.0.1"), or services that are *greater* than a given version (version "1.2.7" is greater than version "1.2" or version "1.2.6").

Version numbering is a black art at best, and this particular filter (Listing 10-3) only makes a best effort attempt at comparing versions. But, it shows the power of being able to use arbitrary filters to prune the set of matched services.

Listing 10–3 ServiceInfoVersionFilter.java

```
// A filter that matches services whose ServiceInfo.version
// matches a search parameter

package corejini.chapter10;

import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceItemFilter;
import net.jini.lookup.entry.ServiceInfo;
import net.jini.core.entry.Entry;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class ServiceInfoVersionFilter
        implements ServiceItemFilter {
    public static final int EQUAL = 1;
    public static final int EQUAL_OR_GREATER = 2;

    protected String test;
    protected String[] testTokens;
    protected int cond;

    public ServiceInfoVersionFilter(String test, int cond){
        this.test = test;
        this.cond = cond;
        testTokens = makeVersionTokens(test);
    }
```

Listing 10–3 ServiceInfoVersionFilter.java(continued)

```
    // Version numbering is an inexact science, but
    // this covers many cases.
    public boolean check(ServiceItem item) {
        Entry[] attrs = item.attributeSets;
        for (int i=0 ; i<attrs.length ; i++) {
            // If it’s a ServiceInfo or subclass
            if (ServiceInfo.class.isAssignableFrom(
                    attrs[i].getClass())) {
                String target = ((ServiceInfo)
                                  attrs[i]).version;

                if (target.equals(test)) {
                    return true;
                }

                switch (cond) {
                case EQUAL:
                    return false;
                case EQUAL_OR_GREATER:
                    String[] targetTokens =
                        makeVersionTokens(target);

                    // Cycle through each.  If any element
                    // in target is less than the
                    // corresponding element in test,
                    // report false.
                    int min = Math.min(testTokens.length,
                                       targetTokens.length);
```

Listing 10–3 ServiceInfoVersionFilter.java(continued)

```
                    for (int j=0 ; j<min ; j++) {
                        try {
                            int testVal =
                           Integer.parseInt(testTokens[j]);
                            int targetVal =
                          Integer.parseInt(targetTokens[j]);

                            if (targetVal < testVal) {
                                return false;
                            }
                       } catch (NumberFormatException ex) {
                            System.err.println(
                                   “Unexpected format: “ +
                                   testTokens[i] + “, “ +
                                   targetTokens[i]);
                            ex.printStackTrace();
                        }
                    }

                    return testTokens.length <=
                           targetTokens.length;
                }
            }
        }

        return false;
    }
```

Listing 10–3 ServiceInfoVersionFilter.java(continued)

```
    // Convert a string of the form 1.2.2.1 into a
    // string array.
    static String[] makeVersionTokens(String v) {
        String str = v;
        // Jettison everything after a space.
        if (str.indexOf(‘ ‘) != -1) {
            str = str.substring(0, str.indexOf(‘ ‘));
        }
        StringTokenizer t = new StringTokenizer(str, “.”);
        ArrayList arr = new ArrayList();
        while (t.hasMoreTokens()) {
            arr.add(t.nextToken());
        }
        return (String[]) arr.toArray(new String[0]);
    }
}
```

This filter is quite a bit more complicated than the previous one. Instances of the `ServiceInfoVersionFilter` are constructed with a string representing a version and a parameter indicating whether the filter should look for exact version matches, or versions that are equal to or greater than the specified version. The input version number is “tokenized” into an array of strings.

The `check()` method finds any `ServiceInfo` attribute that is on the service and checks its version against the original input version. If the two match exactly, then the predicate method returns true. The complicated case is when the filter is looking for versions that are greater than or equal to the input version. The code here tokenizes the service’s version and walks down the two token arrays, pairwise matching version numbers. If any token in the input version is greater than the service’s version, the method returns false. While this code isn’t perfect, it is capable of comparing many styles of version numbers correctly.

The previous two filters give you a way to do very flexible matching over fields of the `ServiceInfo` attribute. But this begs a question: How can you use *two* filters at the same time? The `ServiceDiscoveryManager` APIs only allow you to pass in a single filter to any call to `lookup()` or `createLookupCache()`.

The answer is to use a filter that can aggregate other filters, and this is exactly what the third filter example does. This filter lets you connect up other filters by logical operators (either AND or OR), and evaluates all of them in turn. The

`AggregationFilter` can be a useful way to chain together sets of filtering operations, as shown in Listing 10-4.

Listing 10–4 AggregationFilter.java

```
// A filter that joins other filters together.

package corejini.chapter10;

import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceItemFilter;
import java.util.ArrayList;
import java.util.Iterator;
```

Listing 10–4 AggregationFilter.java(continued)

```
public class AggregationFilter
        implements ServiceItemFilter {
    private ArrayList list = new ArrayList();
    private int operator;

    public static final int AND = 1;// all must match
    public static final int OR = 2;// any must match

    public AggregationFilter(int operator) {
        if (operator == AND || operator == OR) {
            this.operator = operator;
        } else {
            throw new IllegalArgumentException(
                                          “Bogus operator”);
        }
    }

    public void add(ServiceItemFilter filter) {
        list.add(filter);
    }

    public boolean check(ServiceItem item) {
        if (list.size() == 0) {
            return true;
        }

        Iterator iter = list.iterator();

        while (iter.hasNext()) {
            ServiceItemFilter filter =
                (ServiceItemFilter) iter.next();

            boolean result = filter.check(item);


            if (result && operator == OR) {
                return true;
            }

            if (!result && operator == AND) {
                return false;
            }
        }
```

Listing 10–4 AggregationFilter.java(continued)

```
        // If we’ve made it to the end without shortcutting,
        // then we’re in OR and have seen nothing but falses
        // (and should return false), or we’re in AND and
        // we’ve seen nothing but trues (and should return
        // true).
        return (operator == AND);
    }
}
```

The `AggregationFilter` is created by specifying an operand that dicates whether the supplied filters are to be combined by a logical AND operation or a logical OR operation. Filters are added to the aggregation by the `add()` method. The implementation of `check()` simply iterates through the supplied filters, evaluating each in turn according to the specified combination rule.

Finally, the last bit of code is an example to show off how these filters work. Listing 10-5 is an extension of the original `ClientLookupExample`. The only major difference is that the `getFilter()` method has been overridden to use the `AggregationFilter` to connect together a `ServiceInfoVersionFilter` with a `ServiceInfoVendorFilter`.

Listing 10–5 ClientLookupWithFiltering.java

```
// A version of ClientLookupExample to use filtering

package corejini.chapter10;

import net.jini.lookup.ServiceItemFilter;
import java.io.IOException;

public class ClientLookupWithFiltering
        extends ClientLookupExample {
    public ClientLookupWithFiltering() throws IOException {
    }

    public ServiceItemFilter getFilter() {
        AggregationFilter filt =
            new AggregationFilter(AggregationFilter.AND);

        filt.add(new ServiceInfoVendorFilter(“Sun”));
        filt.add(new ServiceInfoVersionFilter(“1.0”,
                ServiceInfoVersionFilter.EQUAL_OR_GREATER));

        return filt;
    }

    public static void main(String[] args) {
        try {
            ClientLookupWithFiltering cle =
                new ClientLookupWithFiltering();
            cle.runTests();
        } catch (Exception ex) {
            System.err.println(ex.toString());
        }

        System.exit(0);
    }
}
```

This short snippet of code is simply an extension of the original `ServiceDiscoveryManager`. The only significant change is that this version overrides the `getFilter()` method, which is used when the example creates a lookup cache.

Here, `getFilter()` creates an `AggregationFilter` holding a `ServiceInfoVendorFilter` and a `ServiceInfoVersionFilter`.

Compiling and Running the Example

Before running, make sure that the class files needed for the `ServiceDiscoveryManager`'s remote event listener are still present in the `client-dl` directory—you'll need them for this example too. If you've deleted them, follow the instructions from the earlier example to extract them again.

On Windows:

```
javac -classpath C:\files;
                 C:\jini1_1\lib\jini-core.jar;
                 C:\jini1_1\lib\jini-ext.jar;
                 C:\jini1_1\lib\sun-util.jar;
                 C:\client
    -d C:\client
  C:\files\corejini\chapter10\ClientLookupWithFiltering.java

java -cp C:\jini1_1\lib\jini-core.jar;
        C:\jini1_1\lib\jini-ext.jar;
        C:\jini1_1\lib\sun-util.jar;
        C:\client
    -Djava.security.policy=C:\files\policy
    -Djava.rmi.server.codebase=http://myhost:8086/
corejini.chapter10.ClientLookupWithFiltering
```

On UNIX:

```
javac -classpath /files:
                 /files/jini1_1/lib/jini-core.jar:
                 /files/jini1_1/lib/jini-ext.jar:
                 /files/jini1_1/lib/sun-util.jar:
                 /files/client
    -d /files/client
    /files/corejini/chapter10/ClientLookupWithFiltering.java

java -cp /files/jini1_1/lib/jini-core.jar:
        /files/jini1_1/lib/jini-ext.jar:
        /files/jini1_1/lib/sun-util.jar:
        /files/client
    -Djava.security.policy=/files/policy
    -Djava.rmi.server.codebase=http://myhost:8086/
corejini.chapter10.ClientLookupWithFiltering
```

This program runs the same sequence of tests that the previous example ran. But here you see that the searches involving the cache return only a select few services—those that contain the string “`Sun`” in the vendor identification, and those with a version greater than or equal to “`1.0`.” (The first calls to `lookup()` do not use a filter, and thus here you see a “`Xerox PARC`” service being returned.)

```
Non-blocking lookup found no services.
Blocking lookup found:   Name = Address Book Service
  Manufacturer = Xerox PARC
  Vendor = Xerox PARC
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.xerox.dispatch.addressbook.AddressBookServiceProxy@0
Cache lookup found no services.
Pausing...
Trying again...
Cache lookup found:   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@f16dfd46
Getting all cached services...
[0]   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.1 alpha
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@af330156
[1]   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@f16dfd46
```

Using Events

The final example of this chapter shows you how to use the event mechanisms provided by the `ServiceDiscoveryManager`. In both of the earlier examples, you saw how the `LookupCache` initially contained no information. The first call to `lookup()` on this cache returned no results; only after waiting for the cache to fill did `lookup()` return services to the caller.

For many clients, this will be perfectly acceptable behavior. Certain clients may be content to examine the contents of the cache only after it has had time to fill, perhaps after some predetermined blocking period has expired.

Other clients, though, may want fine-grained information about available services, *as they become available*. A Jini service browser, for instance, may want to display services to the user as they "trickle in." You probably wouldn't want to write a browser so that it had to wake up every few seconds and do a `lookup()` on the cache to try to figure out what's changed.

Such clients will want to use the event mechanisms provided by `LookupCache` to get information about changes in the state of the cache, as described in the discussion of events earlier.

This example presents an extension of the earlier code, modified to install an event listener to detect changes in the cache. The code you'll see here creates an instance of a class called `DiffListener`, which is a `ServiceDiscoveryListener` that prints out information about changes in the state of the cache. Once this listener is installed, you can see the services trickle in to the `LookupCache` between the time the example program first calls `lookup()` and after the wait.

Listing 10-6 shows the code for `DiffListener`.

Listing 10–6 DiffListener.java

```
// A ServiceDiscoveryListener that shows what's changed

package corejini.chapter10;

import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.lookup.ServiceDiscoveryEvent;
import net.jini.lookup.LookupCache;
import net.jini.core.entry.Entry;

public class DiffListener implements ServiceDiscoveryLis-
tener {
    public DiffListener() {
    }

    // The preEventServiceItem will be null, while the
    // postEventServiceItem will hold the newly-added
    // service item
    public void serviceAdded(ServiceDiscoveryEvent ev) {
        System.out.println("+++ SERVICE ADDED");
        ServiceItem item = ev.getPostEventServiceItem();
        printServiceItem(item);
    }

    // The preEventServiceItem holds the newly-removed
    // service, while the postEventServiceItem is null
    public void serviceRemoved(ServiceDiscoveryEvent ev) {
        System.out.println("+++ SERVICE REMOVED");
        ServiceItem item = ev.getPreEventServiceItem();
        System.out.println("Service's ID was " +
                           item.serviceID);
    }
```

Listing 10–6 DiffListener.java

```
public void serviceChanged(ServiceDiscoveryEvent ev) {
       System.out.println("+++ SERVICE CHANGED");

       // Get both the pre and post service items.
       ServiceItem pre = ev.getPreEventServiceItem();
       ServiceItem post = ev.getPostEventServiceItem();

        // This shouldn't happen!
       if (pre == null && post == null) {
           System.out.println("Null service items!?");
           return;
       }

        // This block of code looks at the proxies to determine
        // if they've changed.  It also looks for some common
        // error cases...notably, if the proxy is null, chances
        // are you've got codebase problems.
        if (pre.service == null && post.service == null) {
            System.out.println(
                 "The service's proxy is still null");
            System.out.println("Codebase problem?");
        } else if (pre.service == null && post.service != null) {
            System.out.println("The service's proxy is no " +
                               "longer null");
            System.out.println("Proxy now: " + post.service +
                                " (" +
                               post.service.getClass().getName()
                                + ")");
       } else if (pre.service != null && post.service == null) {
            System.out.println(
                 "The service's proxy has become null");
            System.out.println("Check codebase");
       } else if (!pre.service.equals(post.service)) {
          System.out.println("The service's proxy has changed");
           System.out.println("Proxy was: " +  pre.service +
                              " (" +
                              pre.service.getClass().getName() +
                              ")");
          System.out.println("Proxy now: " + post.service +
                              " (" +
                            post.service.getClass().getName()
                             + ")");
        }
```

Listing 10–6 DiffListener.java

```
        // The ID of a service should *never* change. If it
        // does change, chances are you've changed the ID
         // in a ServiceItem in the cache!  Remember that you
       // should never write to ServiceItems in the cache.
        if (!pre.serviceID.equals(post.serviceID)) {
            System.err.println("Uh oh, the ID has changed");
            System.err.println(
                "Don't muck with the service items!");
            System.out.println("ID was: " + pre.serviceID);
            System.out.println("ID now: " + pre.serviceID);
        } else {
            System.out.println("ID: " + pre.serviceID);
        }

     // Other changes can happen because of attribute
     // modifications.  This code doesn't detect those...
    }

    // A helper method to print out a service item
    public static void printServiceItem(ServiceItem item) {
        if (item == null) {
            System.out.println("Bogus: null service item");
            return;
        }

        if (item.service == null) {
            System.out.println("Bogus: service proxy is null");
            System.out.println("This service's codebase is " +
                               "probably misconfigured");
        } else {
            System.out.println("Proxy: " + item.service);
            System.out.println("Class: " +
                              item.service.getClass().getName());
        }

        System.out.println("ID:    " + item.serviceID);
```

Listing 10–6 DiffListener.java

```
         Entry[] attrs = item.attributeSets;

         if (attrs == null || attrs.length == 0) {
            System.out.println("Service has no attributes");
        } else {
            for (int i=0 ; i<attrs.length ; i++) {
                System.out.print("[" + i + "] ");
                if (attrs[i] == null) {
                     System.out.println("null entry (possible " +
                                       "codebase problem)");
                } else {
                     System.out.println(attrs[i]);
                }
             }
         }
    }
}
```

To meet the `ServiceDiscoveryListener` interface, `DiffListener` must implement three methods. The first, `serviceAdded()`, is called whenever a new service has appeared in the cache. The implementation here simply gets the `ServiceItem` for the new service and calls the static method `printService()` on it to display some details about the service.

The second method, `serviceRemoved()`, is called whenever a service has been dropped from the cache. Typically this will be because the service has disappeared from the lookup services for the community. Note that the code here fetches the `ServiceItem` for the service as it existed *before* the change, by using `getPreEventServiceItem()`. When a service is dropped from the cache, the `getPostEventServiceItem()` method will return null.

Finally, the last method this class must implement is `serviceChanged()`. The implementation of this method shown here does a bit of work to try to figure out what aspects of the service changed. The code also looks a bit for common error cases. The method first gets both the pre- and post-event `ServiceItems`. It then examines the proxies for each service. A null proxy typically indicates that you've got either a problem with codebase (meaning that the code for the proxy can't be found), or you've got a problem with security (either no security manager, or an overly-restrictive security policy, which means that *no* non-local code will be loaded). The method will detect null proxies, and will note if a proxy becomes null, or ceases to be null. The code here will also compare the two proxies (using `equals()`) to see if the proxies report that they are different.

Next, the code compares the IDs for the two service items. This is a bit of sanity checking: recall that services should *always* use the same ID, everywhere. If you ever see different service IDs for the "same" service, then chances are you've modified one of the `ServiceItems` stored in the cache. Remember that any `ServiceItems` returned to you from the cache should be considered immutable; if you change them, you are likely to corrupt the cache.

Change events for services will often be sent because the set of attributes on a service changes. This code, unfortunately, doesn't detect such changes. You could easily modify it to walk over the two sets of attributes, seeing if attributes were added, removed, or modified.

Next, take a look at the final extension of the main example code in Listing 10-7:

```
Listing 10–7 ClientLookupWithEvents.java
```

```
// A version of ClientLookupExample that uses events

package corejini.chapter10;

import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.lookup.LookupCache;
import java.io.IOException;

public class ClientLookupWithEvents extends ClientLookupEx-
ample {
    protected ServiceDiscoveryListener myListener;

    public ClientLookupWithEvents() throws IOException {
        System.out.println("Creating with events");
        myListener = new DiffListener();
    }

    public ServiceDiscoveryListener getListener() {
        return myListener;
    }

    public static void main(String[] args) {
        try {
            ClientLookupWithEvents cle =
                new ClientLookupWithEvents();

            cle.runTests();
        } catch (Exception ex) {
            System.err.println(ex.toString());
        }

        System.exit(0);
    }
}
```

This class is a very short extension of the first `ClientLookupExample`. The only substantial change here is that the `getListener()` method is overridden to return an instance of the `DiffListener` class.

If you look back to `ClientLookupExample`, you'll see that `getListener()` is called when the program first creates its `LookupCache`. The listener that is returned from this call is installed so that it will receive `ServiceDiscoveryEvents` as the cache is updated.

Compiling and Running the Example

Once again, before running, make sure that the class files needed for the `ServiceDiscoveryManager`'s remote event listener are still present in the `client-dl` directory—you'll need them for this example too. If you've deleted them, follow the instructions from the earlier example to extract them again.

On Windows:

```
javac -classpath C:\files;
                 C:\jini1_1\lib\jini-core.jar;
                 C:\jini1_1\lib\jini-ext.jar;
                 C:\jini1_1\lib\sun-util.jar;
                 C:\client
    -d C:\client
  C:\files\corejini\chapter10\ClientLookupWithEvents.java

java -cp C:\jini1_1\lib\jini-core.jar;
        C:\jini1_1\lib\jini-ext.jar;
        C:\jini1_1\lib\sun-util.jar;
        C:\client
    -Djava.security.policy=C:\files\policy
    -Djava.rmi.server.codebase=http://myhost:8086/
corejini.chapter10.ClientLookupWithEvents
```

On UNIX:

```
javac -classpath /files:
                 /files/jini1_1/lib/jini-core.jar:
                 /files/jini1_1/lib/jini-ext.jar:
                 /files/jini1_1/lib/sun-util.jar:
                 /files/client
    -d /files/client
    /files/corejini/chapter10/ClientLookupWithEvents.java

java -cp /files/jini1_1/lib/jini-core.jar:
        /files/jini1_1/lib/jini-ext.jar:
        /files/jini1_1/lib/sun-util.jar:
        /files/client
    -Djava.security.policy=/files/policy
    -Djava.rmi.server.codebase=http://myhost:8086/
        corejini.chapter10.ClientLookupWithEvents
```

This program runs exactly the same sequence of calls as the first example. Again, note that the non-blocking lookup finds no services and, more interestingly for the purposes of this example, the call to `lookup()` on the `LookupCache` also reports that no services were found.

```
Non-blocking lookup found no services.
Blocking lookup found:   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.1 alpha
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@af330156
Cache lookup found no services.
Pausing...
```

After this sequence, however (which is the same as the previous run), you see a number of `ServiceDiscoveryEvents` being received by the `DiffListener`. These events show the changes to the cache as they happen.

```
+++ SERVICE ADDED
Proxy: com.xerox.dispatch.addressbook.AddressBookServiceProxy@0
Class: com.xerox.dispatch.addressbook.AddressBookServiceProxy
ID: dbc234f7-defb-4dc4-9a22-97361fd89172
[0] net.jini.lookup.entry.ServiceInfo(name=Address Book Service,manufac-
turer=Xerox PARC,vendor=Xerox PARC,version=1.0,model=,serialNumber=)
+++ SERVICE ADDED
Proxy: com.sun.jini.reggie.RegistrarProxy@dd2c2c30
Class: com.sun.jini.reggie.RegistrarProxy
ID: aaf8f7d8-e76c-424a-8862-a37c18da37de
[0]      net.jini.lookup.entry.ServiceInfo(name=Lookup,manufacturer=Sun
Microsystems, Inc.,vendor=Sun Microsystems, Inc.,version=1.0,model=,serial-
Number=)
+++ SERVICE ADDED
Proxy: com.xerox.dispatch.authenticator.AuthenticatorProxy@0
Class: com.xerox.dispatch.authenticator.AuthenticatorProxy
ID: 3131ea32-e31d-4047-9e21-f1bf4eb95d60
[0] net.jini.lookup.entry.ServiceInfo(name=Address Book Service,manufac-
turer=Xerox PARC,vendor=Xerox PARC,version=1.0,model=,serialNumber=)
+++ SERVICE ADDED
```

```
Proxy: com.sun.jini.reggie.RegistrarProxy@af330156
Class: com.sun.jini.reggie.RegistrarProxy
ID: 5b41c161-dab6-47a9-b22a-bbd29cee3c4c
[0]     net.jini.lookup.entry.ServiceInfo(name=Lookup,manufacturer=Sun Microsystems, Inc.,vendor=Sun Microsystems, Inc.,version=1.1 alpha,model=,serialNumber=)
```

After a number of events have been received, the final call to `lookup()` shows the new state of the cache, which reflects the changes represented by the various `ServiceDiscoveryEvents.`

```
Trying again...
Cache lookup found:   Name = Address Book Service
  Manufacturer = Xerox PARC
  Vendor = Xerox PARC
  Version = 1.0
  Model = null
  Serial Number = null
  Proxy is com.xerox.dispatch.addressbook.AddressBookServiceProxy@0
Getting all cached services...
[0]   Name = Address Book Service
  Manufacturer = Xerox PARC
  Vendor = Xerox PARC
  Version = 1.0
  Model =
  Serial Number =
Proxy is com.xerox.dispatch.addressbook.AddressBookServiceProxy@0
[1]   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@f16dfd46
[2]   Name = Authentication Service
  Manufacturer = Xerox PARC
  Vendor = Xerox PARC
  Version = 1.0
  Model =
  Serial Number =
  Proxy is com.xerox.dispatch.authenticator.AuthenticatorProxy@0
[3]   Name = Lookup
  Manufacturer = Sun Microsystems, Inc.
  Vendor = Sun Microsystems, Inc.
  Version = 1.1 alpha
  Model =
  Serial Number =
  Proxy is com.sun.jini.reggie.RegistrarProxy@af330156
```

Summary

This chapter has shown the details of how clients interact with lookup services using the high-level—and very powerful—`ServiceDiscoveryManager`. This class provides a layer of “insulation” between your application code and the low-level `ServiceRegistrar` interface.

The interfaces you’ve seen here will, in all likelihood, be the most common APIs you use when building Jini applications.

What’s Next?

In the next chapter, you’ll put your newly-found knowledge of lookup services to good use by building a complete lookup service browser from the ground up.