CHAPTER 6

Client Search

THIS CHAPTER LOOKS AT WHAT THE CLIENT has to do once it has found a service locator and wishes to find a service. From the service locator, the client will get a `ServiceRegistrar` object. To find a service from the locator, the client needs to prepare a description of the service, which it does using a `ServiceTemplate` object. The client will then call one of two methods on the `ServiceRegistrar` to return either a single matching service or a set of matching services.

Searching for Services with the ServiceRegistrar

A client gets a `ServiceRegistrar` object from the lookup service, and it uses the `lookup()` method to search for a service stored on that lookup service. Here is the `lookup()` method:

```
public Class ServiceRegistrar {
    public java.lang.Object lookup(ServiceTemplate tmpl)
                            throws java.rmi.RemoteException;
    public ServiceMatches lookup(ServiceTemplate tmpl,
                                 int maxMatches)
                            throws java.rmi.RemoteException;
}
```

The first of these methods just finds a service that matches the request. The second finds a set (as many as `maxMatches`).

The lookup methods use a class of type `ServiceTemplate` to specify the service looked for:

```
package net.jini.core.lookup;

public Class ServiceTemplate {
    public ServiceID serviceID;
    public java.lang.Class[] serviceTypes;
    public Entry[] attributeSetTemplates;
```

```
    ServiceTemplate(ServiceID serviceID,
                    java.lang.Class[] serviceTypes,
                    Entry[] attrSetTemplates);
}
```

Although each service should have been assigned a `serviceID` by a lookup service, a client might not know the `serviceID` (it could be the first time the client has looked for this service, for example). In this case, the `serviceID` is set to `null`. If the client does know the `serviceID`, then it can set the value to find the service. The `attributeSetTemplates` is a set of `Entry` elements used to match attributes, and it will be discussed in the "Matching Services" section, later in this chapter.

The major parameter of the `lookup()` methods is a `ServiceTemplate`, which contains a list of `serviceTypes`. We know that services export instances of a class, but how does the client ask so that it gets a suitable instance delivered from the lookup locator?

Although the lookup services keep instances of objects for the service, the client will only know about a service from its specification (unless it already has a `serviceID` for the service), and the specification will almost certainly be a Java interface. Therefore, the client needs to ask using this interface. An interface can have a class object just like ordinary classes, so the list of `serviceTypes` will typically be a list of class objects for service interfaces. Thus, the client will usually request an interface object.

To be more concrete, suppose a toaster is defined by this interface:

```
public interface Toaster extends java.io.Serializable {
    public void setDarkness(int dark);
    public void startToasting();
}
```

A Breville "Extra Lift" toaster would implement this interface in one particular way, as would other toasters:

```
public class BrevilleExtraLiftToaster implements Toaster {
    public void setDarkness(int dark) {
        ...
    }
    public void startToasting() {
        ...
    }
}
```

When the `Toaster` service starts, it exports an object of class `BrevilleExtra-LiftToaster` to the lookup service. However, the client does not know what type of toaster is out there, so it will make a request like this:

```
System.setSecurityManager(new RMISecurityManager());

    // specify the interface object
    Class[] toasterClasses = new Class[1];
    toasterClasses[0] = Toaster.class;

    // prepare a search template of serviceID, classes and entries
    ServiceTemplate template = new ServiceTemplate(null,
                                                   toasterClasses,
                                                   null);

    // now find a toaster
    Toaster toaster = null;
    try {
        toaster = (Toaster) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        System.exit(2);
    }
```

Notice that `lookup()` can throw an exception. This can occur if, for example, the service requested cannot be de-serialized.

As a result of calling the `lookup()` method, an object (an instance of a class implementing the `Toaster` interface) has been transported across to the client, and the object has been coerced to be of this `Toaster` type. This object has two methods: `setDarkness()` and `startToasting()`. No other information is available about the toaster's capabilities because the interface does not specify any more, and in this case the set of attribute values was `null`. So the client can call either of the two methods:

```
toaster.setDarkness(1);
toaster.startToasting();
```

Before leaving this discussion, you might wonder what the role of `System.set-SecurityManager(new RMISecurityManager())` is. A serialized object has been transported across the network and is reconstituted and coerced to an object implementing `Toaster`. We know that here it will, in fact, be an object of class `BrevilleExtraLiftToaster`, but the client doesn't need to know that. Or does it? Certainly the client will not have a class definition for this class on its side. But when

the toaster object begins to run, then it must run using its `BrevilleExtraLift-Toaster` code! Where does it get it from?

From the server—most likely by an HTTP request on the server. This means that the `Toaster` object is *loading a class definition* across the network, and this requires security access. So a security manager capable of granting this access must be installed before the load request is made.

Note the difference between loading a serialized instance and loading a class definition: the first does not require access rights; only the second does. So if the client had the class definitions of all possible toasters, then it would never need to load a class and would not need a security manager that allows classes to be loaded across the network. This is not likely, but may perhaps be needed in a high-security environment.

Receiving the ServiceMatches Object

If a client wishes to search for more than one match to a service request from a particular lookup service, then it specifies the maximum number of matches it would like returned by using the `maxMatches` parameter of the second `lookup()` method. The client gets back a `ServiceMatches` object that looks like this:

```
package net.jini.core.lookup;

public Class ServiceMatches {
    public ServiceItem[] items;
    public int totalMatches ;
}
```

The number of elements in `items` need not be the same as `totalMatches`. Suppose there are five matching services stored on the locator. In that case, `totalMatches` will be set to 5 after a lookup. However, if you used `maxMatches` to limit the search to at most two matches, then `items` will be set to be an array with only two elements.

In addition, not all elements of this array need be non-null! Note that in `lookup(tmpl)` when asking for only one match, an exception can be returned, such as when the service is not serializable. No exception is thrown here, because although one match might be bad, the others might still be okay. So a value of `null` as the array element value is used to signify this. The following code shows how to properly handle the `ServiceMatches` object:

```
ServiceMatches matches = registrar.lookup(template, 10);
    // NB: matches.totalMatches may be greater than matches.items.length
    for (int n = 0; n < matches.items.length; n++) {
```

```
        Toaster toaster = (Toaster) matches.items[n].service;
    if (toaster != null) {
        toaster.setDarkness(1);
        toaster.startToasting();
        }
    }
```

This code will start up to ten toasters cooking at once!

Matching Services

As mentioned previously, a client attempts to find one or more services that satisfy its requirements by creating a `ServiceTemplate` object and using this in a registrar's `lookup()` call. A `ServiceTemplate` object has three fields:

```
ServiceID         serviceID;
java.lang.Class[] serviceTypes;
Entry[]           attributeSetTemplates;
```

If the client is repeating a request, then it may have recorded the `serviceID` from an earlier request. The `serviceID` is a globally unique identifier, so it can be used to identify a service unambiguously. This `serviceID` can be used by the service locator as a filter to quickly discard other services.

Alternatively, a client may want to find a service satisfying several interface requirements at once. For example, a client may look for a service that implements both `Toaster` and `FireAlarm` (so that it can properly handle burnt toast). The client will fill the `serviceTypes` array with all of the interface classes that the service has to implement.

And finally, the client will specify a set of attributes in the `attrSetTemplates` field that must be satisfied by each service. Each attribute required by the client is taken in turn and matched against the set offered by the service. For example, in addition to requesting a `Toaster` with a `FireAlarm`, a client entry may specify a location in GP South Building. This will be tried against all the variations of location offered by the service. A single match is good enough. An additional client requirement of, say, manufacturer would also have to be matched by the service.

The following more formal description comes from the `ServiceTemplate` API documentation:

1. A service item (`item`) matches a service template (`tmpl`) if: `item.serviceID` equals `tmpl.serviceID` (or if `tmpl.serviceID` is `null`); and `item.service` is an instance of every type in `tmpl.serviceTypes`; and `item.attributeSets`

contains at least one matching entry for each entry template in `tmpl.attributeSetTemplates`.

2. An entry matches an entry template if the class of the template is the same as, or a superclass of, the class of the entry, and every non-null field in the template equals the corresponding field of the entry. Every entry can be used to match more than one template. Note that in a service template, for `serviceTypes` and `attributeSetTemplates`, a `null` field is equivalent to an empty array; both represent a wildcard.

Summary

A client prepares a `ServiceTemplate`, which is a list of class objects and a list of entries. For each service locator that is found, the client can query the `ServiceRegistrar` object by preparing a `ServiceTemplate` object and calling the `ServiceRegistrar` object's `lookup()` method to see if the locator has a service matching the template. If the match is successful, an object is returned that can be cast into the class required. Service methods can then be invoked on this object.