CHAPTER 17

ServiceDiscoveryManager

Clients and services need to find lookup services. Both can do this using low-level core classes or discovery utilities such as `LookupDiscoveryManager`. Once a lookup service is found, a service just needs to register with it and try to keep the lease alive for as long as it wants to. A service can make use of the `JoinManager` class for this.

The `ServiceDiscoveryManager` class performs client-side functions similar to that of `JoinManager` for services, and it simplifies the tasks of finding services. It is only available in Jini 1.1 and later.

ServiceDiscoveryManager Interface

The `ServiceDiscoveryManager` class is a utility class designed to help in the various client-side lookup cases that can occur:

- A client may wish to use a service immediately or later.
- A client may want to use multiple services.
- A client will want to find services by their interfaces, but may also want to apply additional criteria, such as being a "fast enough" printer.
- A client may just wish to use a service if it is available at the time of the request, but alternatively may want to be informed of new services becoming available and to respond to this new availability (e.g., a service browser).

Due to the variety of possible cases, the `ServiceDiscoveryManager` class is more complex than `JoinManager`. Its interface includes the following:

```
package net.jini.lookup;
public class ServiceDiscoveryManager {
    public ServiceDiscoveryManager(DiscoveryManagement discoveryMgr,
                                   LeaseRenewalManager leaseMgr)
           throws IOException;
    public ServiceDiscoveryManager(DiscoveryManagement discoveryMgr,
                                   LeaseRenewalManager leaseMgr,
                                   Configuration config)
           throws IOException,
                  ConfigurationException;
    LookupCache createLookupCache(ServiceTemplate tmpl,
```

```
                                ServiceItemFilter filter,
                                ServiceDiscoveryListener listener);
    ServiceItem[] lookup(ServiceTemplate tmpl,
                         int maxMatches, ServiceItemFilter filter);
    ServiceItem lookup(ServiceTemplate tmpl,
                       ServiceItemFilter filter);
    ServiceItem lookup(ServiceTemplate tmpl,
                       ServiceItemFilter filter, long wait);
    ServiceItem[] lookup(ServiceTemplate tmpl,
                         int minMaxMatch, int maxMatches,
                         ServiceItemFilter filter, long wait);
    void terminate();
}
```

ServiceItemFilter Interface

Most methods of the client lookup manager require a `ServiceItemFilter`. This is a simple interface designed to be an additional filter on the client side to help in finding services. The primary way for a client to find a service is to ask for an instance of an interface, possibly with additional entry attributes. This matching is performed on the lookup service, and it only involves a form of exact pattern matching. It allows the client to ask for a toaster, for example, that will handle two slices of toast exactly, but not for one that will toast two or more.

Performing arbitrary Boolean matching on the lookup service raises a security issue, as it would involve running some code from the client or service in the lookup service, and it also raises a possible performance issue for the lookup service. This means that enhancing the matching process in the lookup services is unlikely to occur, so any more sophisticated matching must be done by the client. The `ServiceItemFilter` allows additional Boolean filtering to be performed on the client side, by client code, so these issues are local to the client only.

The `ServiceItemFilter` interface is as follows:

```
package net.jini.lookup;
public interface ServiceItemFilter {
    boolean check(ServiceItem item);
}
```

A client filter will implement this interface to perform additional checking.

Client-side filtering will not solve all of the problems of locating the "best" service. Some situations will still require other services that know "local" information, such as distances in a building.

Finding a Service Immediately

The simplest scenario for a client is that it wants to find a service immediately, use it, and then (perhaps) terminate. The client will be prepared to wait a certain amount of time before giving up. All issues of discovery can be given to the `ServiceDiscoveryManager`, and the task of finding a service can be given to a method such as `lookup()` with a `wait` parameter. The `lookup()`

method will block until a suitable service is found or the time limit is reached. If the time limit is reached, a `null` object will be returned; otherwise, a non-null service object will be returned.

```
package client;
import common.FileClassifier;
import common.MIMEType;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
/**
 * ImmediateClientLookup.java
 */
public class ImmediateClientLookup {
    private static final long WAITFOR = 100000L;
    public static void main(String argv[]) {
        new ImmediateClientLookup();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
    public ImmediateClientLookup() {
        ServiceDiscoveryManager clientMgr = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null, // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
```

```
        // Try to find the service, blocking until timeout if necessary
        try {
            item = clientMgr.lookup(template,
                                    null, // no filter
                                    WAITFOR); // timeout
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("no service");
            System.exit(1);
        }
        // Get the service
        FileClassifier classifier = (FileClassifier) item.service;
        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }
        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            // Try several file types: .txt, .rtf, .abc
            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }
    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
```

```
            System.out.println(type.toString());
        }
    }
} // ImmediateClientLookup
```

Using a Filter

An example in Chapter 15 discussed how to select a printer with a speed greater than a certain value. This type of problem is well suited to the `ServiceDiscoveryManager` using a `ServiceItemFilter`. The `ServiceItemFilter` interface has a `check()` method, which is called on the client side to perform additional filtering of services. This method can accept or reject a service based on criteria supplied by the client. The following program illustrates how this `check()` method can be used to select only printer services with a speed of greater than 24 pages per minute.

```
package client;
import common.Printer;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceItemFilter;
/**
 * TestPrinterSpeedFilter.java
 */
public class TestPrinterSpeedFilter implements ServiceItemFilter {
    private static final long WAITFOR = 100000L;

    public TestPrinterSpeedFilter() {
        ServiceDiscoveryManager clientMgr = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        Class[] classes = new Class[] {Printer.class};
```

```
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
         ServiceItem item = null;
        try {
            item = clientMgr.lookup(template,
                                    this,     // filter
                                    WAITFOR); // timeout
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.exit(1);
        }
        Printer printer = (Printer) item.service;
        // Now use the printer
        // ...
    }
    public boolean check(ServiceItem item) {
        // This is the filter
        Printer printer = (Printer) item.service;
        if (printer.getSpeed() > 24) {
            return true;
        } else {
            return false;
        }
    }
    public static void main(String[] args) {

        TestPrinterSpeed f = new TestPrinterSpeed();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
} // TestPrinterSpeed
```

Building a Cache of Services

A client may wish to make use of a service multiple times. If the client simply found a suitable reference to a service, then before each use it would have to check that the reference was still valid, and if not, it would need to find another one. The client may also want to use minor variants of a service, such as a fast printer one time and a slow one the next. While this management can be done easily enough in each case, the `ServiceDiscoveryManager` can supply a cache of

services that will do this work for you. This cache will monitor lookup services to keep the cache as up to date as possible.

The cache is defined as an interface:

```
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);
    void terminate();
}
```

A suitable implementation object can be created by the `ServiceDiscoveryManager` method:

```
LookupCache createLookupCache(ServiceTemplate tmpl,
                              ServiceItemFilter filter,
                              ServiceDiscoveryListener listener);
```

We will ignore the `ServiceDiscoveryListener` until the next section of this chapter. It can be set to `null` in `createLookupCache()`.

The `LookupCache` created by `createLookupCache()` takes a template for matching against interface and entry attributes. It also takes a filter to perform additional client-side Boolean filtering of services. The cache will then maintain a set of references to services matching the template and passing the filter. These references are all local to the client and consist of the service proxies and their attributes downloaded to the client. Searching for a service can then be done by local methods: `LookupCache.lookup()`. These can take an additional filter that can be used to further refine the set of services returned to the client.

The search in the cache is done directly on the proxy services and attributes already found by the client, and it does not involve querying lookup services. Essentially, this involves a tradeoff of lookup service activity while the client is idle to produce fast local response when the client is active.

There are versions of `ServiceDiscoveryManager.lookup()` with a time parameter, which block until a service is found or the method times out. These methods do not use polling, but instead use event notification because they are trying to find services based on remote calls to lookup services. The `lookup()` methods of `LookupCache` do not implement such a blocking call because the methods run purely locally, and it is reasonable to poll the cache for a short time if need be.

Here is a version of the file classifier client that creates and examines the cache for suitable service:

```
package client;
import common.FileClassifier;
import common.MIMEType;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
```

```
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.lookup.LookupCache;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
/**
 * CachedClientLookup.java
 */
public class CachedClientLookup {
    private static final long WAITFOR = 100000L;
    public static void main(String argv[]) {
        new CachedClientLookup();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
    public CachedClientLookup() {
        ServiceDiscoveryManager clientMgr = null;
        LookupCache cache = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                    new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {FileClassifier.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        try {
            cache = clientMgr.createLookupCache(template,
                                                null,  // no filter
                                                null); // no listener
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
```

```
        // loop until we find a service
        ServiceItem item = null;
        while (item == null) {
            System.out.println("no service yet");
            try {
                Thread.currentThread().sleep(1000);
            } catch(java.lang.InterruptedException e) {
                // do nothing
            }
            // see if a service is there now
            item = cache.lookup(null);
        }
        FileClassifier classifier = (FileClassifier) item.service;
        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }
        // Now we have a suitable service, use it
        MIMEType type;
        try {
            String fileName;

            fileName = "file1.txt";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file2.rtf";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);

            fileName = "file3.abc";
            type = classifier.getMIMEType(fileName);
            printType(fileName, type);
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }
    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }
} // CachedClientLookup
```

Running the CachedClientLookup

While it is OK to poll the local cache, the cache itself must get its contents from lookup services, and in general it is not OK to poll these because doing so involves possibly heavy network traffic. The cache gets its information by registering itself as a listener for service events from the lookup services. The lookup services will then call `notify()` on the cache listener. This is a remote call from the remote lookup service to the local cache, done (probably) using an RMI stub. In fact, the Sun implementation of `ServiceDiscoveryManager` uses a nested class, `ServiceDiscoveryManager.LookupCacheImpl.LookupListener`, which has an RMI stub.

In order for the cache to actually work, it is necessary to set the RMI codebase property `java.rmi.server.codebase` to a suitable location for the class files (such as an HTTP server), and to make sure that the class `net/jini/lookup/ServiceDiscoveryManager$LookupCacheImpl$LookupListener_Stub.class` is accessible from this codebase. The stub file may be found in the library `lib/jini-ext.jar` in the Jini 1.1 distribution. It has to be extracted from there and placed in the codebase using a command such as this:

```
unzip jini-ext.jar 'net/jini/lookup/ServiceDiscoveryManager$LookupCacheIm-
pl$LookupListener_Stub.class' -d /home/WWW/htdocs/classes
```

Note that the specification just says this type of thing has to be done but does not descend to details about the class name—that is left to the documentation of the `ServiceDiscoveryManager` as implemented by Sun. If another implementation is made of the Jini classes, then it would probably use a different remote class.

Monitoring Changes to the Cache

The cache uses remote events to monitor the state of lookup services. It includes a local mechanism to pass some of these changes to a client by means of the `ServiceDiscoveryListener` interface:

```
package net.jini.lookup;
interface ServiceDiscoveryListener {
    void serviceAdded(ServiceDiscoveryEvent event);
    void serviceChanged(ServiceDiscoveryEvent event);
    void serviceRemoved(ServiceDiscoveryEvent event);
}
```

with event

```
package net.jini.lookup;
class ServiceDiscoveryEvent extends EventObject {
    ServiceItem getPostEventServiceItem();
    ServiceItem getPreEventServiceItem();
}
```

Clients are not likely to be interested in all events generated by lookup services, even for services in which they are interested. For example, if a new service registers itself with ten lookup services, they will all generate transition events from `NO_MATCH` to `MATCH`, but the client will usually only be interested in seeing the first of these—the other nine are just repeated

information. Similarly, if a service's lease expires from one lookup service, then that doesn't matter much, but if it expires from all lookup services that the client knows of, then it does matter, because the service is no longer available to it. The cache consequently prunes events so that the client gets information about the real services rather than information about the lookup services.

From Chapter 16, recall the example involving monitoring changes to services from a lookup service viewpoint, reporting each change to lookup services. A client-oriented view just monitors changes in services themselves, which can be done easily using `ServiceDiscoveryEvent` objects:

```
package client;
import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.lookup.ServiceDiscoveryListener;
import net.jini.lookup.ServiceDiscoveryEvent;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.LookupCache;
/**
 * ServiceMonitor.java
 */
public class ServiceMonitor implements ServiceDiscoveryListener {
    public static void main(String argv[]) {
        new ServiceMonitor();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }
    public ServiceMonitor() {
        ServiceDiscoveryManager clientMgr = null;
        LookupCache cache = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null,  // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
```

```
        ServiceTemplate template = new ServiceTemplate(null, null,
                                                       null);
        try {
            cache = clientMgr.createLookupCache(template,
                                                null,  // no filter
                                                this); // listener
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    // methods for ServiceDiscoveryListener
    public void serviceAdded(ServiceDiscoveryEvent evt) {
        // evt.getPreEventServiceItem() == null
        ServiceItem postItem = evt.getPostEventServiceItem();
        System.out.println("Service appeared: " +
                           postItem.service.getClass().toString());
    }
    public void serviceChanged(ServiceDiscoveryEvent evt) {
        ServiceItem preItem = evt.getPostEventServiceItem();
        ServiceItem postItem = evt.getPreEventServiceItem() ;
        System.out.println("Service changed: " +
                           postItem.service.getClass().toString());
    }
    public void serviceRemoved(ServiceDiscoveryEvent evt) {
        // evt.getPostEventServiceItem() == null
        ServiceItem preItem = evt.getPreEventServiceItem();
        System.out.println("Service disappeared: " +
                           preItem.service.getClass().toString());
    }

} // ServiceMonitor
```

Summary

Clients searching for services have varying requirements. Clients may wish to block until a service is found, or continue doing something while a cache of services is built. Or they may wish to filter the services using more complex logic than just service type and `Entry` values. The service discovery manager is a complex class that can be used for many of these different situations, and it makes it easier to write clients with nontrivial service requirements.