Steal This Book!

Yes, you read that right. Steal this book. For free.

Nothing. Zero. Zilch. Nadda. Zip.

Undoubtedly you're asking yourself, "Why would he give away a book he probably spent six grueling months writing? Is he crazy?"

The answer…is yes. Sort of. I know that every day you're faced with hundreds of computer titles, and a lot of them don't deliver the value or the information you need. So here's the deal: I'll give you this book (or this chapter, if you've only downloaded part of the book) for free provided you do me a couple favors:

1. **Send this book to your friends:** No, not your manager. Your "with it" computer friends who are looking for the next Big Thing. JXTA is it. Trust me. They want to know about it.

2. **Send a link to the book's web site:** Maybe the book is too big to send. After all, not everyone can have a fibre optic Internet connection installed in their bedroom. The site, at **www.brendonwilson.com/projects/jxta**, provides chapter-sized PDFs for easy downloading by the bandwidth-challenged.

3. **Visit the book's web site:** Being a professional developer, you probably have Carpal Tunnel Syndrome and shudder at the idea of typing in example source code. Save yourself the trouble. Go to **www.brendonwilson.com/projects/jxta** and download the source code. And while you're there, why not download some of the chapters you're missing?

4. **Buy the book:** You knew there had to be a catch. Sure, the book's PDFs are free, but I'm hoping that enough of you like the book so much that you have to buy a copy. Either that, or none of you can stand to read the book from a screen (or, worse yet, print it all out <shudder>) and resort to paper to save what's left of your eyesight. The book is available at your local bookstore or from **Amazon.com** (at a **handsome discount**, I might add).

I now return to your regularly scheduled program: enjoy the book!

5

The Peer Resolver Protocol

Chapter 4, "The Peer Discovery Protocol," showed how to discover peers and advertisements using the Discovery service, but it did not address how the Discovery service handles sending and receiving messages. The Discovery service isn't responsible for sending its Discovery Query and Response Messages. Instead, the Discovery service is built on top of another service, the Resolver service, which handles sending and receiving messages for the Discovery service.

This chapter details the Peer Resolver Protocol (PRP) and the Resolver service that implements the protocol. The PRP defines a protocol for sending a generic query to a named handler located on another peer and processing a generic response to a query. Other services in JXTA, such as the Discovery service, build on the capabilities of the Resolver service and the PRP to provide higher-level services.

Introducing the Peer Resolver Protocol

The Discovery service detailed in Chapter 4 described two types of messages: one for sending a discovery query and another for sending a response to a discovery query. The deceptive simplicity of the discovery messages hides several layers of abstraction that insulate the developer from the inner complexities of the JXTA protocols. To develop new solutions built on top of the JXTA platform, it's essential to understand these layers.

When a peer sends a Discovery Query Message using the `getRemoteAdvertisements` method, the `DiscoveryService` implementation doesn't simply create a Discovery Query Message and pass it over the network itself. Instead, the Discovery service uses another service, the Resolver service, to handle the details of sending the message on its behalf. The Resolver service provides an implementation of the PRP, which defines how peers can exchange query and response messages.

The Resolver service is responsible for wrapping a query string in a more generic message format and sending it to a specific handler on a remote peer. In the case of the Discovery service, the query string is the Discovery Query Message and the handler is the remote peer's Discovery service. On the remote peer, a Resolver service instance is responsible for passing an incoming message to the appropriate handler and sending any response generated by the handler.

In the general case, the Resolver service needs only two types of messages:

- **Resolver Query Message**—A message format for sending queries
- **Resolver Response Message**—A message format for sending responses to queries

These two message formats define generic messages to send queries and responses between peers, as shown in Figure 5.1. At each end, a handler registered with a peer group's Resolver service instance processes query strings and generates response strings.

Like the Peer Discovery Protocol, a query is sent to known peers and propagated through known rendezvous peers. Any peer's Resolver service that receives a Resolver Query Message attempts to find a registered handler for the query. If a matching handler is found, the Resolver passes it the message and manages sending the response message generated by the handler.

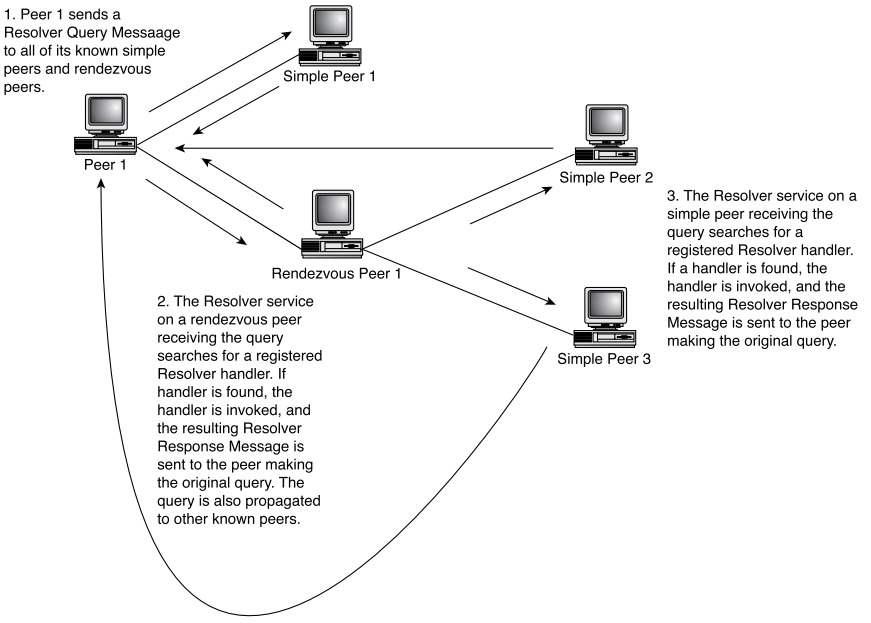

Figure 5.1 Exchange of Resolver messages.

Like the Discovery service, the Resolver service does not require a response to a Resolver Query Message. The registered handler might not generate a response to a given query and can indicate to the Resolver service the reason that it has not generated a response. A handler may not generate a response because it has decided that the query is invalid in some way. In this case, the query is not propagated to other peers. A handler might also indicate that it wants to learn the response generated by other peers in response to the query. To accomplish this, the handler can ask the Resolver service to resend the query in a manner that will allow it to receive the response generated by other peers.

Although the PRP is the default resolver protocol used by the JXTA reference implementation, developers can provide their own custom resolver mechanism. A developer might want to provide his own resolver mechanism to incorporate additional information that provides a better or more efficient service. This custom solution could be built independently of the PRP or could be built on top of the PRP.

The Resolver Query Message

Queries to other peers are wrapped inside a Resolver Query Message using the format shown in Listing 5.1.

Listing 5.1 **Resolver Query Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:ResolverQuery xmlns:jxta="http://jxta.org">
  <HandlerName> . . . </HandlerName>
  <Credential> . . . </Credential>
  <QueryID> . . . </QueryID>
  <SrcPeerID> . . . </SrcPeerID>
  <Query> . . . </Query>
</jxta:ResolverQuery>
```

The elements in the Resolver Query Message provide all the details that a peer's Resolver service needs to match the query to a registered handler:

- **`HandlerName`**—A required element containing a unique name specifying the name of the handler that the Resolver service on the destination peer should invoke to process the query. Because the Resolver service provides service within the context of a peer group, a handler name must be unique within a peer group. Only one handler of a given name should be registered on a given peer in a peer group, and this assumption is enforced in the Java reference implementation. If a second handler is registered with the Resolver for a peer group using a duplicate handler name, the first handler registered with the Resolver service is removed.
- **`Credential`**—An optional element containing an authentication token that identifies the source peer and its authorization to send the query to the peer group.
- **`QueryID`**—An optional element containing a long integer, encoded as a string, that defines an identifier for the query. This identifier should be unique to the query. This identifier should be sent back in a response to this query, allowing the sender to match a response to a specific query.
- **`SrcPeerID`**—A required element containing the ID of the peer sending the query. This Peer ID uses the standard JXTA URN format, as described in the JXTA Protocols Specification.
- **`Query`**—A required element containing the query string being sent to the remote peer's handler. This string could be anything; it is the responsibility of the handler to understand how to parse this query string, process the query, and possibly generate a response message.

The implementation of the Resolver Query Message, as shown in Figure 5.2, is divided in a similar manner to the Discovery Query and Response Messages. The `ResolverQueryMsg` abstract class in the `net.jxta.protocol` package defines the basic interface, variables for the query's attributes, and accessors to manipulate the attributes. Only the `getDocument` method is abstract, allowing implementers to provide their own mechanism for rendering the query to a `Document` instance.

The `ResolverQuery` class in the `net.jxta.impl.protocol` package provides an implementation of `getDocument` capable of creating a `StructuredTextDocument` representation of the query in the specified `MimeMediaType`. Several constructors can create a `ResolverQuery` instance from a variety of input parameters, including an `InputStream`, or the raw query attributes.

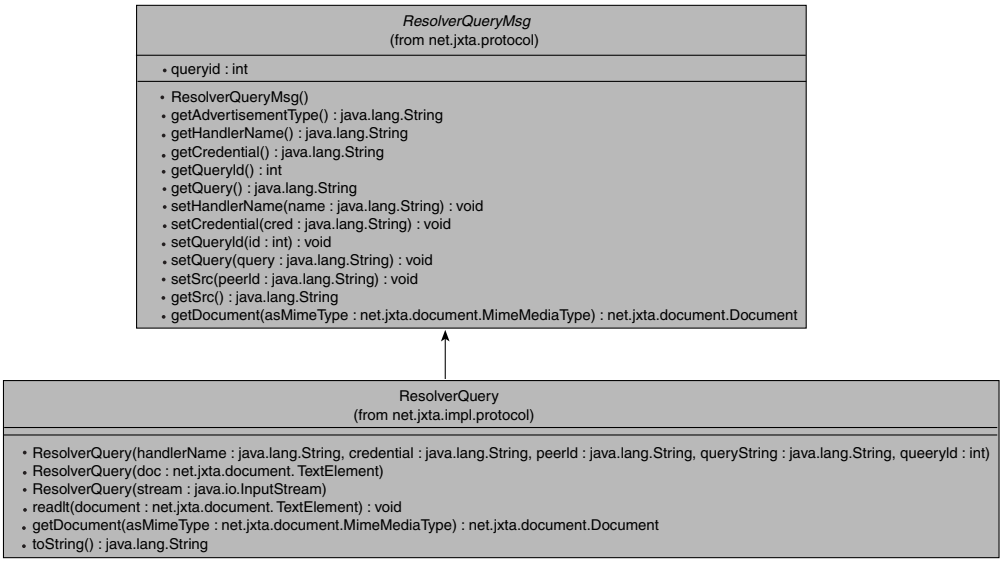

Figure 5.2 The Resolver Query Message classes.

A developer can create a Resolver Query Message at any time to send a query to a specific Resolver handler on a remote peer. For example, a call to `getRemoteAdvertisements` in the reference `DiscoveryService` implementation `DiscoveryServiceImpl` causes the `DiscoveryServiceImpl` to create a `DiscoveryQuery` instance, wrap it in a `ResolverQuery` instance, and send it using the Resolver service.

The Resolver Response Message

The Resolver Response Message responds to a Resolver Query Message using the format shown in Listing 5.2.

Listing 5.2 **Resolver Response Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:ResolverResponse xmlns:jxta="http://jxta.org">
  <HandlerName> . . . </HandlerName>
  <Credential> . . . </Credential>
  <QueryID> . . . </QueryID>
  <Response> . . . </Response>
</jxta:ResolverResponse>
```

The Resolver Response Message provides similar details to the Resolver Query Message:

- **`HandlerName`**—A required element containing the name of a handler that should be invoked by the remote peer's Resolver service to process the response. A different handler name from that used in the query might be used to allow a different Resolver handler to process the response.
- **`Credential`**—An optional element containing an authentication token that identifies the peer sending the response and its authorization to send the response to the destination peer group.
- **`QueryID`**—An optional element containing a long integer, encoded as a string, that defines an identifier for the query. This identifier should correspond to the `QueryID` sent in the query that caused the peer to generate this Resolver Response Message. If the `QueryID` provided in the original query is unique to the query and the handler, the sender can match this Resolver Response Message to the original query. Matching the response to a given query might be necessary to provide useful functionality in a P2P application.
- **`Response`**—A required element containing the response string being sent to the remote peer's handler. This string could be anything; it is the responsibility of the handler to understand how to parse this response string.

In the Java reference implementation of JXTA, the abstract definition of the Resolver Response Message is defined by `ResolverResponseMsg` in the `net.jxta.protocol` package, as shown in Figure 5.3. The reference implementation of the abstract class is implemented by the `ResolverResponse` class in the `net.jxta.impl.protocol` package.

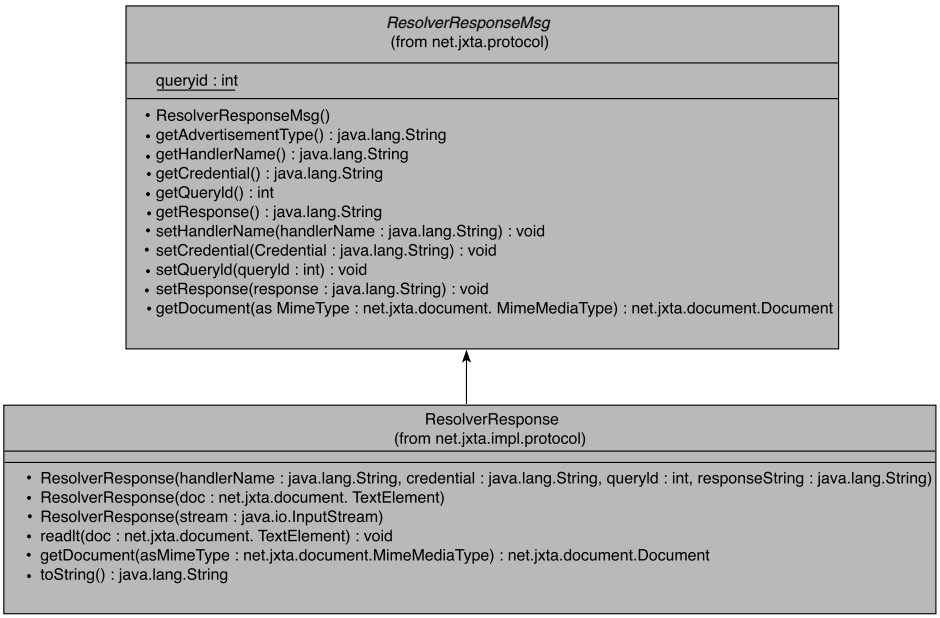

Figure 5.3 The Resolver Response Message classes.

A Resolver Response Message can be used to "push" information to peers by sending a Resolver Response Message without first receiving a Resolver Query Message. This capability is used by the `DiscoveryService` implementation `DiscoveryServiceImpl` to publish advertisements to remote peers whenever the `remotePublish` method is called.

The Resolver Service

The Resolver service, another JXTA core service, provides a simple interface that developers can use to send queries and responses between members of a peer group. The Resolver service is defined by the `ResolverService` interface in the `net.jxta.resolver` package, shown in Figure 5.4, which is derived from the `GenericResolver` interface.

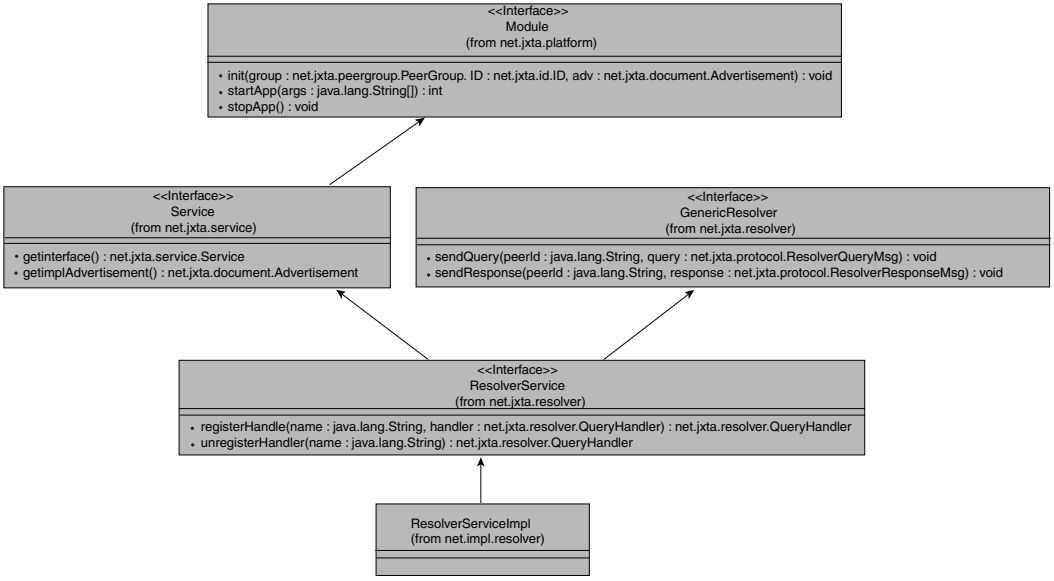

Figure 5.4 The Resolver service interfaces and classes.

The `GenericResolver` interface defines the methods for sending queries and responses using implementations of `ResolverQueryMsg` and `ResolverResponseMsg`. More important, the `ResolverService` interface defines the methods for registering and unregistering an implementation of the `net.jxta.resolver.QueryHandler` interface and associating it with a handler name. A registered `QueryHandler` instance is invoked when the Resolver service receives a Resolver Query or Response Message whose `HandlerName` matches the handler string used to register the handler with the `ResolverService` for a peer group.

The *QueryHandler* Interface

The `QueryHandler` interface, shown in Figure 5.5, is similar to the `DiscoveryListener` interface in Chapter 4. Like `DiscoveryListener`, the `QueryHandler` interface provides a developer with a way to provide his own mechanism for handling response messages. Unlike `DiscoveryListener`, the `QueryHandler` interface also provides a developer with a mechanism for handling query messages received from other peers.

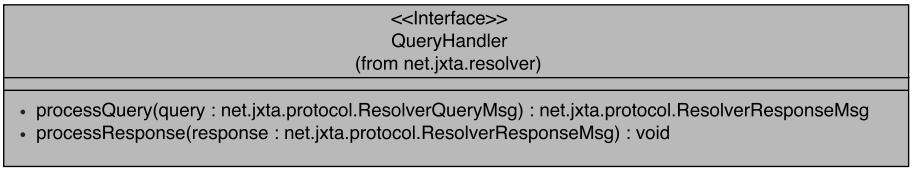

Figure 5.5 The `QueryHandler` interface.

To begin handling queries, a `QueryHandler` instance first must be registered with a peer group's `ResolverService` using a unique handler name. After it is registered, a peer group's `ResolverService` instance invokes the `QueryHandler`'s `processQuery` method to process `ResolverQueryMsg` instances addressed to the handler. The `processQuery` implementation is responsible for extracting and processing the query string from the `ResolverQueryMsg`. This processing can result in one of five outcomes:

- The `processQuery` method returns a populated `ResolverResponseMsg` object containing the response to the query. The `ResolverService` instance handles sending this response back to the peer that sent the original query.
- The `processQuery` method throws a `NoResponseException`, indicating that the handler has no response to the query. If the peer is a rendezvous peer, the ResolverService instance still propagates the query to other peers in the peer group.
- The `processQuery` method throws a `ResendQueryException`, indicating that the handler has no response to the query but would be interested in learning the response given by other peers. If the peer is a rendezvous peer, the `ResolverService` propagates the query message as usual to other peers in the peer group. In addition to propagating the message, the `ResolverService` instance resends the message (masquerading as the source peer) to obtain the responses provided by other peers to the query.
- The `processQuery` method throws a `DiscardException`, indicating that the `ResolverService` should discard the query entirely. The `ResolverService` instance discards the query and does not propagate the query to other peers in the peer group. A query can be discarded because the query, despite being well formed, might be invalid in some way.
- The `processQuery` method throws an `IOException` when the handler cannot process the query, possibly due to an error in the format of the query string. In the reference `ResolverService` implementation, this exception causes the Resolver service to act in the same manner as when a `DiscardException` occurs.

The `QueryHandler`'s `processResponse` method is invoked by the `ResolverService` to process a `ResolverResponseMsg` instance. Unlike `processQuery`, `processResponse` doesn't produce any results or throw any exceptions. Either the response is processed or it isn't. The `ResolverService` instance doesn't need to know anything about the results of the processing.

One thing that might seem curious is that the Resolver service and the `QueryHandler` interface don't provide information on how the query or response strings are formatted. No mechanism exists for a peer to discover how to format a query string for a given handler or what format to expect in response to a successful query. The details of the query and response string formatting are implicit in the implementation of the handler, and JXTA does not provide any way of discovering how to invoke the handler. This is one area that JXTA does not address but that could be addressed in the future by adopting one of the forthcoming XML-based standards for service discovery, such as the Web Services Description Language (WSDL).

Implementing a Resolver Handler

The example covered in this section creates a simple handler that allows a peer to query a remote peer for the value of a specified base raised to a specified power. The query string provides the base and power values in the format shown in Listing 5.3.

Listing 5.3 **The Example Query Message**

```
<?xml version="1.0"?>
<example:ExampleQuery xmlns:example="http://jxta.org">
  <base> . . . </base>
  <power> . . . </power>
</example:ExampleQuery>
```

Responses to the query provide the answer to the query using the format in Listing 5.4.

Listing 5.4 **The Example Response Message**

```
<?xml version="1.0"?>
<example:ExampleResponse xmlns:example="http://jxta.org">
  <base> . . . </base>
  <power> . . . </power>
  <answer> . . . </answer>
</example:ExampleResponse>
```

The example Resolver handler accepts a query, extracts the base and power values, calculates the value of the base raised to the power, and returns a response message populated with the base, power, and answer values.

Creating the Query and Response Strings

Implementing a Resolver handler requires a developer only to provide an implementation of the `QueryHandler` interface and register the handler with a peer group's Resolver service. However, a developer should still abstract the process of parsing query strings and formatting response strings, in the interest of readability and maintainability.

The Discovery service, covered in Chapter 4, relies on the `DiscoveryQueryMsg`, `DiscoveryQuery`, `DiscoveryResponseMsg`, and `DiscoveryResponse` classes. These classes provided a mechanism for the `DiscoveryService` implementation to produce or consume a query or response string in a fairly abstract fashion. For this example, there's no need to go as far as defining both an abstract class and an implementation for the query and response objects. The query and response objects used in this example use a similar approach to provide encapsulated parsing and formatting functionality. Listing 5.5 shows the source code for an object to handle parsing and formatting the query XML shown in Listing 5.4.

Listing 5.5 **Source Code for *ExampleQueryMsg.java***

```
package net.jxta.impl.shell.bin.example5_1;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
```

continues

Listing 5.5 **Continued**

```
 * An example query message, which will be wrapped by a Resolver Query
 * Message to send the query to other peers. The query essentially asks
 * a simple math question: "What is the value of (base) raised to (power)?"
*/
public class ExampleQueryMsg
{
    /**
     * The base for query.
     */
    private double base = 0.0;

    /**
     * The power for the query.
     */
    private double power = 0.0;


    /**
     * Creates a query object using the given base and power.
     *
     * @param   aBase the base for the query.
     * @param   aPower the power for the query.
     */
    public ExampleQueryMsg(double aBase, double aPower)
    {
        super();

        this.base = aBase;
        this.power = aPower;
    }

    /**
     * Creates a query object by parsing the given input stream.
     *
     * @param        stream the InputStream source of the query data.
     * @exception   IOException if an error occurs reading the stream.
     */
    public ExampleQueryMsg(InputStream stream) throws IOException
    {
            StructuredTextDocument document = (StructuredTextDocument)
                StructuredDocumentFactory.newStructuredDocument(
```

```
                new MimeMediaType("text/xml"), stream);

    Enumeration elements = document.getChildren();

    while (elements.hasMoreElements())
    {
        TextElement element = (TextElement) elements.nextElement();

        if(element.getName().equals("base"))
        {
            base = Double.valueOf(element.getTextValue()).doubleValue();
            continue;
        }

        if(element.getName().equals("power"))
        {
            power = Double.valueOf(
                element.getTextValue()).doubleValue();
            continue;
        }
    }
}

/**
 * Returns the base for the query.
 *
 * @return  the base value for the query.
 */
public double getBase()
{
    return base;
}

/**
 * Returns a Document representation of the query.
 *
 * @param   asMimeType the desired MIME type representation for the
 *          query.
 * @return  a Document form of the query in the specified MIME
 *          representation.
 */
```

continues

Listing 5.5 **Continued**

```
public Document getDocument(MimeMediaType asMimeType)
{
    StructuredDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            asMimeType, "example:ExampleQuery");
    Element element;

    element = document.createElement(
        "base", Double.toString(getBase()));
    document.appendChild(element);

    element = document.createElement("power",
        Double.toString(getPower()));
    document.appendChild(element);

    return document;
}

/**
 * Returns the power for the query.
 *
 * @return  the power value for the query.
 */
public double getPower()
{
    return power;
}

/**
 * Returns an XML String representation of the query.
 *
 * @return  the XML String representing this query.
 */
public String toString()
{
    try
    {
        StringWriter out = new StringWriter();
        StructuredTextDocument doc = (StructuredTextDocument)
            getDocument(new MimeMediaType("text/xml"));
        doc.sendToWriter(out);
```

```
            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

Like the `DiscoveryQueryMsg` and `DiscoveryQuery` classes, the `ExampleQueryMsg` class defines fields for the query, methods for rendering the message as a `Document` and a `String`, and constructors for populating a query. The `getDocument` method creates a `Document` for the given `MimeMediaType` using the `StructuredDocumentFactory` class and adds the `base` and `power` fields. The `toString` method provides a convenient way to render the query object to an XML string, resulting in a query string in the format defined at the beginning of this section.

The encapsulation of the response message format is almost identical, as shown in Listing 5.6.

Listing 5.6 **Source Code for *ExampleResponseMsg.java***

```
package net.jxta.impl.shell.bin.example5_1;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
 * An example query response, which will be wrapped by a Resolver Response
```

continues

Listing 5.6 **Continued**

```
 * Message to send the response to the query. The response contains the
 * answer to the simple math question posed by the query.
 */
public class ExampleResponseMsg
{
    /**
     * The base from the original query.
     */
    private double base = 0.0;

    /**
     * The power from the original query.
     */
    private double power = 0.0;

    /**
     * The answer value for the response.
     */
    private double answer = 0;


    /**
     * Creates a response object using the given answer value.
     *
     * @param   anAnswer the answer for the response.
     */
    public ExampleResponseMsg(double aBase, double aPower, double anAnswer)
    {
        this.base = aBase;
        this.power = aPower;
        this.answer = anAnswer;
    }

    /**
     * Creates a response object by parsing the given input stream.
     *
     * @param       stream the InputStream source of the response data.
     * @exception   IOException if an error occurs reading the stream.
     */
    public ExampleResponseMsg(InputStream stream) throws IOException
    {
```

```
    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType("text/xml"), stream);

    Enumeration elements = document.getChildren();

    while (elements.hasMoreElements())
    {
        TextElement element = (TextElement) elements.nextElement();

        if(element.getName().equals("answer"))
        {
            answer = Double.valueOf(
                element.getTextValue()).doubleValue();
            continue;
        }

        if(element.getName().equals("base"))
        {
            base = Double.valueOf(element.getTextValue()).doubleValue();
            continue;
        }

        if(element.getName().equals("power"))
        {
            power = Double.valueOf(
                element.getTextValue()).doubleValue();
            continue;
        }
    }
}

/**
 * Returns the answer for the response.
 *
 * @return  the answer value for the response.
 */
public double getAnswer()
{
    return answer;
}
```

continues

Listing 5.6 **Continued**

```
/**
 * Returns the base for the query.
 *
 * @return  the base value for the query.
 */
public double getBase()
{
    return base;
}

/**
 * Returns a Document representation of the response.
 *
 * @param   asMimeType the desired MIME type representation for
 *          the response.
 * @return  a Document form of the response in the specified MIME
 *      representation.
 */
public Document getDocument(MimeMediaType asMimeType)
{
    Element element;
    StructuredDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            asMimeType, "example:ExampleResponse");

    element = document.createElement(
        "base", Double.toString(getBase()));
    document.appendChild(element);

    element = document.createElement("power",
        Double.toString(getPower()));
    document.appendChild(element);

    element = document.createElement("answer",
        (new Double(getAnswer()).toString()));
    document.appendChild(element);

    return document;
}

/**
```

```
     * Returns the power for the query.
     *
     * @return  the power value for the query.
     */
    public double getPower()
    {
        return power;
    }

    /**
     * Returns an XML String representation of the response.
     *
     * @return  the XML String representing this response.
     */
    public String toString()
    {
        try
        {
            StringWriter buffer = new StringWriter();
            StructuredTextDocument document = (StructuredTextDocument)
                getDocument(new MimeMediaType("text/xml"));
            document.sendToWriter(buffer);

            return buffer.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

These objects simplify the task of creating a Resolver Query or Response Message. For example, a developer can create a query string and wrap it in a Resolver Query Message using only this code:

```
ExampleQueryMsg equery =
    new ExampleQueryMsg(base, power);
ResolverQuery query = new ResolverQuery("ExampleHandler",
    "JXTACRED", localPeerId, equery.toString(), queryId);
```

The query string can be extracted and parsed using this code:

```
equery = new ExampleQueryMsg(
    new ByteArrayInputStream((query.getQuery()).getBytes()));
```

Both of these examples demonstrate how much simpler it is to use the encapsulated query and response objects compared to the alternative of manually formatting or parsing the query or response string.

Implementing the *QueryHandler* Interface

The task of implementing the `QueryHandler` interface is greatly simplified by the query and response objects defined in the previous section. Listing 5.7 shows the source code for the sample `QueryHandler`.

Listing 5.7 **Source Code for *ExampleHandler.java***

```
package net.jxta.impl.shell.bin.example5_1;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import net.jxta.exception.NoResponseException;
import net.jxta.exception.DiscardQueryException;
import net.jxta.exception.ResendQueryException;

import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.protocol.ResolverQueryMsg;
import net.jxta.protocol.ResolverResponseMsg;

import net.jxta.resolver.QueryHandler;


/**
 * A simple handler to process Resolver queries.
 */
class ExampleHandler implements QueryHandler
{
    /**
     * Processes the Resolver query message and returns a response.
     *
     * @param        query the Resolver Query Message to be processed.
     * @return       a Resolver Response Message to send to the peer
     *               responsible for sending the query.
     * @exception    IOException throw if the query string can't be parsed.
     */
    public ResolverResponseMsg processQuery(ResolverQueryMsg query)
```

```
    throws IOException, NoResponseException, DiscardQueryException,
        ResendQueryException
{
    ResolverResponse response;
    ExampleQueryMsg eq;
    double answer = 0.0;

    System.out.println("Processing query...");

    // Parse the message from the query string.
    eq = new ExampleQueryMsg(
        new ByteArrayInputStream((query.getQuery()).getBytes()));

    // Perform the calculation.
    answer = Math.pow(eq.getBase(), eq.getPower());

    // Create the response message.
    ExampleResponseMsg er = new ExampleResponseMsg(eq.getBase(),
        eq.getPower(), answer);

    // Wrap the response message in a resolver response message.
    response = new ResolverResponse("ExampleHandler",
        "JXTACRED", query.getQueryId(), er.toString());

    return response;
}

/**
 * Process a Resolver response message.
 *
 * @param   response the Resolver Response Message to be processed.
 */
public void processResponse(ResolverResponseMsg response)
{
    ExampleResponseMsg er;

    System.out.println("Processing response...");

    try
    {
        // Extract the message from the Resolver response.
        er = new ExampleResponseMsg(
```

continues

Listing 5.7 **Continued**

```
                new ByteArrayInputStream(
                    (response.getResponse()).getBytes()));

            // Print out the answer given in the response.
            System.out.println(
                "The value of " + er.getBase() + " raised to "
                    + er.getPower() + " is: " + er.getAnswer());
        }
        catch (IOException e)
        {
            // This is not the right type of response message, or
            // the message is improperly formed. Ignore the message,
            // do nothing.
        }
    }
}
```

The `QueryHandler` interface's only two methods, `processQuery` and `processResponse`, use the objects defined in the previous section to parse and format the query and response strings. The only real work that is done by the `ExampleHandler` is the calculation of the response's `answer` value using the query's `base` and `power` values.

Registering the Handler with the *ResolverService* Instance

To see the `QueryHandler` implementation created in the previous section in action, an instance of `ExampleHandler` needs to be registered with a peer group's `ResolverService` instance. The following example `shell` command, shown in Listing 5.8, registers an `ExampleHandler` instance with the current peer group's `ResolverService` instance and sends an `ExampleQueryMsg` using input values provided by the user.

Listing 5.8 **Source Code for *example5_1.java***

```
package net.jxta.impl.shell.bin.example5_1;

import net.jxta.impl.protocol.ResolverQuery;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
```

```
import net.jxta.impl.shell.ShellObject;

import net.jxta.peergroup.PeerGroup;

import net.jxta.resolver.QueryHandler;
import net.jxta.resolver.ResolverService;


/**
 * A simple application to demonstrate the use of the Resolver service
 * to register a QueryHandler instance and process queries.
 */
public class example5_1 extends ShellApp
{
    /**
     * A flag indicating if the example's handler should be unregistered
     * from the peer group's Resolver service.
     */
    private boolean removeHandler = false;

    /**
     * A name to use to register the example handler with the
     * Resolver service.
     */
    private String name = "ExampleHandler";

    /**
     * The base value for the query.
     */
    private double base = 0.0;

    /**
     * The power value for the query.
     */
    private double power = 0.0;


    /**
     * Manages adding or removing the handler from the Resolver service.
     *
     * @param   resolver the Resolver service with which to register or
     *           unregister a handler.
```

continues

Listing 5.8 **Continued**

```
     */
    private void manageHandler(ResolverService resolver)
    {
        if (removeHandler)
        {
            // Unregister the handler from the Resolver service.
            ExampleHandler handler =
                (ExampleHandler) resolver.unregisterHandler(name);
        }
        else
        {
            // Create a new handler.
            ExampleHandler handler = new ExampleHandler();

            // Register the handler with the Resolver service.
            resolver.registerHandler(name, handler);
        }
    }

    /**
     * Parses the command-line arguments and initializes the command
     *
     * @param        args the arguments to be parsed.
     * @exception    IllegalArgumentException if an invalid parameter
     *                is passed.
     */
    private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;

        // Parse the arguments to the command.
        GetOpt parser = new GetOpt(args, "b:p:r");

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'b' :
                {
                    try
```

```
                {
                    // Obtain the "base" element for the query.
                    base = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    base = 0.0;
                }

                break;
            }

            case 'p' :
            {
                try
                {
                    // Obtain the "power" element for the query.
                    power = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    power = 0.0;
                }

                break;
            }

            case 'r' :
            {
                // Remove the handler.
                removeHandler = true;
                break;
            }
        }
    }
}

/**
```

continues

Listing 5.8 **Continued**

```
     * The implementation of the Shell command, invoked when the command
     * is started by the user from the Shell.
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *           the command.
     */
    public int startApp(String[] args)
    {
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group.
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

        // Get the Resolver service for the current peer group.
        ResolverService resolver = currentGroup.getResolverService();

        try
        {
            // Parse the command-line arguments.
            parseArguments(args);
        }
        catch (IllegalArgumentException e)
        {
            println("Incorrect parameters passed to the command.");
            result = ShellApp.appParamError;
        }

        // Manage the handler for the Resolver service. This
        // adds or removes the handler as specified by the
        // command-line parameters.
        manageHandler(resolver);

        // If we're not removing the handler, send a query using
        // the Resolver service.
if (!removeHandler)
        {
```

```
                ExampleQueryMsg equery = new ExampleQueryMsg(base, power);
                String localPeerId = currentGroup.getPeerID().toString();

                // Wrap the query in a resolver query.
                ResolverQuery query = new ResolverQuery("ExampleHandler",
                    "JXTACRED", localPeerId, equery.toString(), 0);

                // Send the query using the resolver.
                println("Sending base="+base+", power="+power);
                resolver.sendQuery(null, query);
            }

            return result;
        }
    }
```

Of particular importance is the registration of the `ExampleHandler`:

```
// Register the handler with the Resolver service.
resolver.registerHandler(name, handler);
```

The `name` variable defines the name of the handler that identifies this handler to the `ResolverService` instance. In the `example5_1` command, `name` is set to `ExampleHandler`. A Resolver Query Message or a Resolver Response Message must use the same handler name to identify the target handler for its query or response string.

Because a peer group's `ResolverService` instance can define only one handler with a given name, the `registerHandler` method replaces an existing handler. Any handler previously registered with the `ResolverService` instance using the same handler name is returned by the `registerHandler` method.

Sending a Resolver Query Message

To send a query, the `example5_1` command creates an `ExampleQueryMsg` object using the `base` and `power` values provided by the user and wraps it in a `ResolverQuery` object:

```
ExampleQueryMsg equery = new ExampleQueryMsg(base, power);
String localPeerId = currentGroup.getPeerID().toString();
ResolverQuery query = new ResolverQuery("ExampleHandler",
    "JXTACRED", localPeerId, equery.toString(), 0);
```

The identifier for the local peer, `localPeerId`, is retrieved from the `PeerGroup` object holding the current peer group in the Shell when the command is invoked. The `JXTACRED string` provides a value for the `Credential` in the

`ResolverQuery`. Currently, the JXTA reference implementation doesn't provide any abstract mechanism for validating credentials, although this feature is expected in the future. Currently, developers can implement their own credential validation schemes within their `QueryHandler` implementations until this shortcoming is addressed.

Finally, the Resolver Query Message is sent to all peers in the `ResolverService` instance's peer group using this line:

```
resolver.sendQuery(null, query);
```

The first parameter identifies the Peer ID for the destination of the query. If this parameter is `null`, the `ResolverService` instance propagates the query to all peers in the peer group.

When a Resolver service receives a Resolver Query Message, it extracts the `HandlerName`, checks for a matching registered `QueryHandler` instance, and, if one exists, passes the Resolver Query Message object to the handler's `processQuery` method.

Using the *ExampleHandler* Class

To see the example in action, two peers on the P2P network must register an `ExampleHandler` instance with a specific peer group's `ResolverService` instance using the same handler name. Because it's unlikely that another peer will be running the example code at the same time, you must start two instances of the Shell. To start two instances of the Shell, follow these steps:

1. Delete the `PlatformConfig` file and the `pse` and `cm` directories from your `Shell` directory. Run the `Shell`, force reconfiguration using the `peerconfig` command, and `exit` the `Shell`.
2. Copy the `Shell` subdirectory from the JXTA Demo install directory into a directory called `Shell2`. This directory should be at the same directory level as the original `Shell` subdirectory.
3. Compile the example's code, and place a copy in both the `Shell` and `Shell2` subdirectories. This is required because the example code must be to be available to both `Shell` instances.
4. Run the `Shell` in the `Shell` directory from the command line, as in previous examples. Configure it as usual.
5. Run the `Shell` in the `Shell2` directory from the command line, as in previous examples. In the TCP Settings section of the Advanced tab, specify a different TCP port number (for example, 9702). In the HTTP Settings section of the Advanced tab, specify a different HTTP port number (for example, 9703). In the Basic tab, enter a different name for the peer.

After each Shell has loaded, issue a peer discovery in each Shell using `peers -r`, and ensure that each peer can see the other using the `peers` command. Each peer must be capable of seeing the other peer's name in the list returned by `peers` for the example to work. When both peers can see each other, run the example in the first Shell instance:

```
JXTA>example5_1
```

The command registers an `ExampleHandler` with the current peer group's Resolver service and sends a default query. The default query for the example uses a value of `0.0` for both the `base` and the `power` attributes. No response to this query is received because probably no other peer on the system at this time has a matching handler registered with its Resolver service for the current peer group.

Run the example in the second Shell instance to register a handler. This time, the default query is handled by the `ExampleHandler` registered in the first Shell instance. The first Shell's `ExampleHandler` instance prints to the command console (not the Shell console):

```
Processing query...
```

This indicates that the Resolver service has received a query and passed it to the `processQuery` method of the `ExampleHandler`. The `ExampleHandler`'s `processQuery` method has been invoked correctly, and the handler is processing the query. When the handler returns a response, the Resolver service sends it back to the second Shell instance's peer. When this response is received by the second Shell instance, `ExampleHandler` prints the results to the command console (again, not to the Shell console):

```
Processing response...
The value of 0.0 raised to the power 0.0 is: 1.0
```

This indicates that the `processResponse` method of the `ExampleHandler` registered in the second Shell has been invoked by the Resolver service correctly. Now that both peers have registered a handler, try sending a more meaningful query using this line:

```
JXTA>example5_1 -b4 -p2
```

The query asks other peer for the value of `4` raised to the power 2. The other peer should respond with the value `16`.

Unregistering the Handler

When an application no longer wants a handler to receive messages, it can unregister the handler from the Resolver service. To unregister the handler,

the `unregister` method is called using the name originally used to register to handler:

```
ExampleHandler handler = (ExampleHandler)
    resolver.unregisterHandler(name);
```

Unregistering the handler returns the `QueryHandler` instance that the `ResolverService` has unregistered. If the call to `unregister` returns `null`, the `ResolverService` instance cannot find any registered handler instance with the given name.

Sending Responses

The `example4_6` command developed in the Chapter 4 showed how the Discovery service can be used to publish advertisements to other peers using the `remotePublish` method. To do this, the Discovery service sends a Discovery Response Message using the `ResolverService`'s `sendResponse` method:

```
public void sendResponse(String destPeer, ResolverResponseMsg response);
```

The `sendResponse` method allows a peer to send a Resolver Response Message without first receiving a Resolver Query Message. Using this method, the example given in Listing 5.9 allows a peer to publish answers to other peers.

Listing 5.9 **Source Code for *example5_2.java***

```
package net.jxta.impl.shell.bin.example5_2;

import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;

import net.jxta.impl.shell.bin.example5_1.ExampleResponseMsg;

import net.jxta.peergroup.PeerGroup;

import net.jxta.resolver.ResolverService;


/**
 * A simple application to demonstrate the use of the Resolver service to
 * send a Resolver Response Message without first receiving a Resolver
 * Query Message.
```

```
 */
public class example5_2 extends ShellApp
{
    /**
     * The base value for the response.
     */
    private double base = 0.0;

    /**
     * The power value for the response.
     */
    private double power = 0.0;

    /**
     * The answer value for the response.
     */
    private double answer = 0;


    /**
     * Parses the command-line arguments and initializes the command
     *
     * @param       args the arguments to be parsed.
     * @exception   IllegalArgumentException if an invalid parameter
     *              is passed.
     */
    private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;

        // Parse the arguments to the command.
        GetOpt parser = new GetOpt(args, "b:p:a:");

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'b' :
                {
                    try
                    {
```

continues

Listing 5.9 **Continued**

```
                    // Obtain the "base" element for the response.
                    base = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    base = 0.0;
                }

                break;
            }

            case 'p' :
            {
                try
                {
                    // Obtain the "power" element for the response.
                    power = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    power = 0.0;
                }

                break;
            }

            case 'a' :
            {
                try
                {
                    // Obtain the "answer" element for the response.
                    answer =  (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
```

```
                    answer = 0.0;
                }

                break;
            }
        }
    }
}

/**
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param   args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *           the command.
 */
public int startApp(String[] args)
{
     int result = appNoError;

    // Get the shell's environment.
    ShellEnv theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
    ShellObject theShellObject = theEnvironment.get("stdgroup");
    PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

    // Get the Resolver service for the current peer group.
    ResolverService resolver = currentGroup.getResolverService();

    try
    {
        // Parse the command-line arguments.
        parseArguments(args);
    }
    catch (IllegalArgumentException e)
    {
        println("Incorrect parameters passed to the command.");
        return ShellApp.appParamError;
    }
```

continues

Listing 5.9 **Continued**

```
            String localPeerId = currentGroup.getPeerID().toString();
            ExampleResponseMsg eresponse =
                new ExampleResponseMsg(base, power, answer);
            ResolverResponse pushRes = new ResolverResponse("ExampleHandler",
                "JXTACRED", 0, eresponse.toString());

            // Print out the information we're about to send.
            System.out.println(
                "Sending: base=" + base + ", power=" + power
                    + ", answer=" + answer);

            // Send the response using the resolver.
            resolver.sendResponse(null, pushRes);

            return result;
        }
    }
```

A Resolver Response Message is created by the command in a similar fashion to the `ExampleHandler`'s `processQuery` method in the previous example:

```
ExampleResponseMsg eresponse =
    new ExampleResponseMsg(base, power, answer);
ResolverResponse pushRes = new ResolverResponse("ExampleHandler",
    "JXTACRED", 0, eresponse.toString());
```

Using the arguments passed to the command, the `example5_2` command wraps an `ExampleResponseMsg` in a `ResolverResponse` message. Unlike the previous example, the response is sent using the `ResolverService` directly:

```
resolver.sendResponse(null, pushRes);
```

The first parameter to the `sendResponse` method specifies a destination peer, in the form of a Peer ID `String`. If this string is `null`, the `ResolverService` instance sends the response message to every known peer and propagates the message via known rendezvous peers.

To test the example, start two Shell instances using the procedure given in the previous example. Register an `ExampleHandler` in each instance using the `example5_1` command and then invoke the `example5_2` command in the first Shell instance using this line:

```
JXTA>example5_2 -b4 -p2 -a16
```

This command sends an `ExampleResponseMsg` to all known peers, using a base value of `4`, a power value of `2`, and an answer value of `16`. The second Shell instance receives the message, and the Resolver service invokes the `ExampleHandler` to print a message to the system:

```
The value of 4.0 raised to the power 2.0 is: 16.0
```

The `example5_2` command enables a user to send a response without requiring a query first, allowing a peer to publish an answer before the question has been asked. One of the interesting things to note here is that a peer can provide incorrect answers! This is actually a core problem in P2P computing that is currently the subject of much discussion.

Summary

In this chapter, you learned that the Resolver service is used as a building block by the Discovery service to provide a more generic message-handling framework. Using the Resolver service, you should now be able to create and register handlers to provide your own functionality to a peer group.

In the next chapter, you explore the Rendezvous Protocol and the Rendezvous service. Despite its name, the Rendezvous service is not solely used to provide rendezvous peer services to other peers. The Rendezvous service is a building block that can also be used by services on a peer to propagate messages to other peers within the same peer group. For example, the Resolver service explored in this chapter used the Rendezvous service to propagate queries to remote peers. The next chapter details the protocol behind the Rendezvous service and how it can be used by developers to handle propagating messages to other peers.