CHAPTER 8

Client/Server Mode

Up to this point, you have been using db4o as a simple standalone database in local mode, and have been accessing the database by directly opening the database file. In this mode, db4o acts as an embedded database for applications where access is restricted to one user, process, or thread at a time. However, db4o can do more than this, providing support for a wider range of application scenarios through its client/server modes.

Introduction to db4o Client/Server Modes

In general, a client/server system consists of clients that interact with a central server. The server provides a service using a daemon to listen for and accept connections. Clients then connect to the server via this daemon to perform tasks such as data retrieval, updates, deletions, and general administration.

The db4o API includes functionality to let db4o run as a server, and allows you to write clients that interact with the server, as illustrated in Figure 8-1. In a distributed environment you might choose this methodology to accept connections from a computer, or from PDAs, handheld devices, or cell phones.

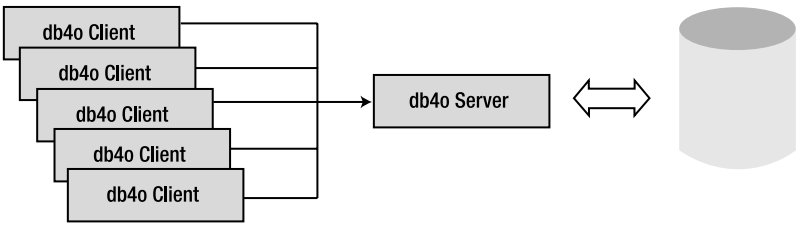

Figure 8-1. *Several db4o clients connecting to a db4o database server*

db4o supports three varieties of client/server interaction, which are introduced here and discussed in later sections:

- The first mode is the **networking mode**, which you have probably used when working with other database solutions. Here, remote clients open a TCP/IP connection to transfer, query, insert, modify, and delete instructions to, and data from, the db4o server. This mode works in db4o exactly as you would expect it to.

- db4o also supports an **embedded mode**, which doesn't involve a distributed system, although the client and the server are quite distinct objects. Instead, both the client and the server are run on the same virtual machine. The communication between the server and the client is the same as in networking mode, but in this mode you work entirely within one virtual machine, which is extremely useful in some applications. One example application for this mode could be the design of a desktop application that uses db4o as storage. If the application is later redesigned to work in a distributed mode, then it is very simple to convert to work in network mode—all you have to do is specify an IP address or hostname and TCP port number for the server.

- The last mode is used for "out-of-band" communications with the server. In this mode the information sent does not belong to the db4o protocol, and does not consist of data objects, but instead is completely user-defined. You can send objects to the server and the server can do with them whatever is needed. This is extremely useful for sending messages to the server like: "do a defragment", "stop yourself", "perform a savecopy", and so forth.

Figure 8-2 depicts all three modes operating in a single environment.

Working in a client/server mode introduces the need for access control and authentication. Up to this point you have not had to think about these issues with db4o, but you are probably familiar with the need for users to log in to gain access to other databases. As you will see, db4o's access control is pretty simple compared to most others.

On the other hand, virtually everything you have learned from the preceding chapters transfers directly over to client/server mode. Storing, updating, and deleting objects, and the query mechanisms, all work exactly the same way whichever mode you are working in.

Let's explore how each of the client/server modes works by looking at a few examples.

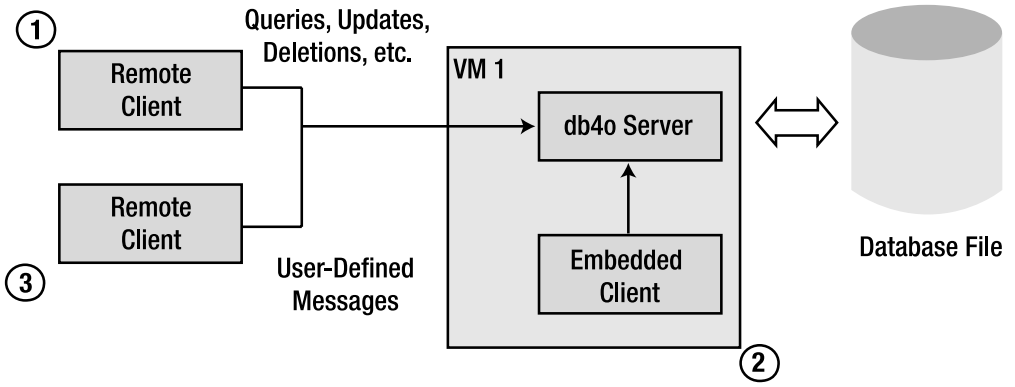

Figure 8-2. *db4o in embedded, networking, and out-of-band-modes*

Networking Mode

Consider a scenario where your customer wants to manage a database consisting of the names, addresses, and contact information of certified C# and Java developers. What you need is a db4o server that hosts this database.

Server

The following code starts a db4o server running, ready to accept connections from clients:

```
// C#
using System;
using com.db4o;

namespace com.db4o.dg2db4o.chapter8
{
    class RunServer
    {
        private bool stop = false;

        private void Run()
        {
            lock (this)
            {
                Console.WriteLine("Starting server...");
                ObjectServer server = Db4o.OpenServer("C:/netserver.yap", 8732);
                server.GrantAccess("user1", "password");
                server.GrantAccess("user2", "password");
```

```
                try
                {
                    // stop condition will be defined in a later example
                    while (!stop)
                    {
                        Console.WriteLine("SERVER: Server is listening ...");
                        Monitor.Wait(this, 60000);

                    }
                }
                catch (ThreadInterruptedException tie)
                {
                    Console.WriteLine("Thread Error!" + tie);
                }
                finally
                {
                    server.Close();
                }
            }
            Console.WriteLine("Server ends...");
        }

    }
}

// JAVA
package com.db4o.dg2db4o.chapter8;

import java.io.*;
import com.db4o.*;

public class RunServer implements Runnable
{
    private boolean stop = false;

    public void run()
    {
        synchronized (this)
        {
            ObjectServer server = Db4o.openServer("c:/netserver.yap", 8732);
            server.grantAccess("user1", "password");
            server.grantAccess("user2", "password");
```

```
            try
            {
                // stop condition will be defined in a later example
                while (!stop)
                {
                    System.out.println("SERVER:[" +
                        System.currentTimeMillis()
                        + "] Server's running... ");
                    this.wait(60000);
                }
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
            finally
            {
                server.close();
            }
        }
    }
}
```

■ **Note** As you've seen in earlier chapters, note that no schema or table creation is required! Any objects your client sends will be stored by the server without problems. This is one of the key strengths of a "zero administration" server, but of course, your clients need to be reliable and send only "correct" data.

In our case this would mean the clients should send `Person` objects containing `_name` and `_age` fields. However, even if the clients decide one year later to "upgrade" their `Person` objects to include addresses, db4o will quietly execute the necessary schema changes. We'll further discuss this matter in Chapter 12.

Now let's have a look what the previous code does. You can see that you simply open a server with `Db4o.OpenServer`: this simply means that an extra thread is started automatically that listens for incoming requests from clients. Requests for database operations to be executed are fulfilled using the database file specified in the call to `OpenServer`. The filename parameter refers to exactly the same kind of db4o database file you might use if db4o were running in local mode. The second parameter is the TCP port on which you wish your server to listen. You should make sure you use an unassigned port number—see `www.iana.org/assignments/port-numbers` for a list of port number assignments. You should also make sure your port number is not used by any other application in your environment or another db4o server.

The next step is to grant access to the server. You can grant access to as many users as you wish, using `GrantAccess` to set the access username and password for each one. Each client will then have to supply valid credentials in order to connect. If you forget to grant access with a password and a user, then no client will be able to connect.

Once the server has started, and before entering the `try/catch` block, you might want to set the priority of the thread in which this code runs to a low value, because the db4o server establishes its own high-priority thread. In the `while` loop we continue until the boolean flag `stop` is set. The example code does not define a stop condition: it simply loops until the thread or process is interrupted. You can either insert your own condition here, which will set `stop` to `true`, or you can use out-of-band signaling, which is explained later on this chapter.

This code does nothing more than communicate to you that the server is alive every minute. Of course, the real purpose of the server is to listen for and process client requests. And this is indeed what the server quietly does in the background. To try this out, we need some clients.

Clients

Returning to the project requirements, we need some way to enter data into the database. Let's create a client that connects to the server and allows data to be stored:

```
// C#
using System;
using com.db4o;

namespace com.db4o.dg2db4o.chapter8
{
    class AddClient
    {
        private ObjectContainer aClient;
        private bool stop = false;

        public void Run()
        {
            Console.WriteLine("Starting add client...");
            aClient = Db4o.OpenClient("localhost", 8732, "user1", "password");
            while (!stop)
            {
                Console.WriteLine("ADDCLIENT: Please enter a name: ");
                String name = Console.ReadLine();
                Console.WriteLine("ADDCLIENT: Please enter an age: ");
                int age = Convert.ToInt32(Console.ReadLine());
                aClient.Set(new Person(name, age));
                aClient.Commit();
            }
            aClient.Close();
        }

    }
}
```

```
// JAVA
package com.db4o.dg2db4o.chapter8;

import java.util.Scanner;
import com.db4o.*;

public class AddClient implements Runnable
{
    private boolean stop = false;
    ObjectContainer aClient = null;
    Scanner sc = new Scanner(System.in);

    public void run()
    {
        try
        {
            aClient = Db4o.openClient("localhost", 8732, "user1", "password");
            while (!stop)
            {
                System.out.print("\nADDCLIENT:Enter the developer's
                    name and age (e.g. 'Tom 44'): ");
                aClient.set(new Person(sc.next(), sc.nextInt()));
                aClient.commit();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            aClient.close();
        }
    }
}
```

This code looks suspiciously incomplete, but it really has all you need. You open a client with `Db4o.openClient`, passing the hostname or IP address of the computer on which the server is running. In the listing, the host is "localhost", which means that the server and client are on the same computer (although not necessarily the same virtual machine)—in a live network environment this would be a resolvable hostname or a network IP address. You also specify the server's port number, and a set of valid username and password credentials.

The `OpenClient` call returns a client `ObjectContainer` reference, which can then be used exactly as you have used `ObjectContainer`s in local mode.

Next, we loop continuously, entering a name and an age each time to create a new `Person` object, and saving this person in the client `ObjectContainer`, meaning these objects will also be saved to the db4o database. We call `Commit` on the `ObjectContainer` each time to make sure the object is written to the database. (Chapter 9 will explain the idea of committing transactions in more detail.) Again the listing does not include a way of setting the `stop` flag—you can add your own condition to test in the `while` loop to determine when to set the flag.

So what is missing? This is supposed to be a multiuser database server, so let's give it another user to deal with. We can create another client capable of simply viewing the database content:

```
// C #
using System;
using com.db4o;

namespace com.db4o.dg2db4o.chapter8
{
    class ListClient
    {
        ObjectContainer lClient = null;
        private bool stop = false;

        public void Run()
        {
            lock (this)
            {
                Console.WriteLine("Starting list client...");
                try
                {
                    lClient = Db4o.OpenClient("127.0.0.1", 8732, "user2",
                        "password");
                    while (!stop)
                    {
                        ListResult(lClient.Get(new Person()));
                        Monitor.Wait(this, 10000);
                    }
                }
                catch (ThreadInterruptedException tie)
                {
                    Console.WriteLine("Thread Error!" + tie);
                }
                finally
                {
                    lClient.Close();
                }
            }
        }
```

```
        private static void ListResult(ObjectSet result)
        {
            Console.WriteLine ("LISTCLIENT: Listing...");
            while (result.HasNext())
            {
                Console.WriteLine("LISTCLIENT:" + result.Next() + "\n");
            }
        }

    }
}

// JAVA
package com.db4o.dg2db4o.chapter8;

import com.db4o.*;

public class ListClient implements Runnable
{
    private boolean stop = false;
    ObjectContainer rClient = null;

    public void run()
    {
        synchronized(this){
            try
            {
                rClient = Db4o.openClient("127.0.0.1", 8732, "user2", "password");
                while (!stop)
                {
                    listResult(rClient.get(new Person()));
                    Thread.sleep(15000);
                }
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
            finally
            {
                aClient.close();
            }
        }
    }
```

```
    public static void listResult(ObjectSet result)
    {
        System.out.println("LISTCLIENT: Listing...");
        while (result.hasNext())
        {
            System.out.println("LISTCLIENT:" + result.next());
        }
    }
}
```

This client opens the network connection to the server with `OpenClient` as before. To show that you can use the IP address, this example has been written to connect to the local host using the loopback IP address, 127.0.0.1. Then it enters the `while` loop, where it periodically queries the database and lists the results.

Running the Example

You can try out this example in a number of different ways. You can run the three classes listed as separate threads within one virtual machine; you can run them as three separate processes on the same computer; or, to give the best representation of a real network server environment, you can run them on three separate networked computers so that you have real remote clients. In the latter case, you need to specify the correct hostname or IP address for your clients to connect to.

■ **Caution** You need to make sure that the server and client processes all have the data classes in their classpath. If you are running this example with remote clients, you need to remember to deploy the `Person` class file to the remote computers. In a real client/server scenario you can package your data classes in a DLL or JAR library and include this library in your client *and* server applications.

To run as processes, you will need to create separate main methods to instantiate each class, and to call `Run` for each instance. You can run these on the same computer, or separate networked computers—it's more fun if you do it over a real network!

To run as threads, you will need to create a class with a main method that starts the threads, as shown here:

```
// C#
public static void Main(string[] args)
{
    RunServer s = new RunServer();
    Thread serverThread = new Thread(new ThreadStart(s.Run));
    serverThread.Priority = ThreadPriority.Highest;
    AddClient a = new AddClient();
    ListClient l = new ListClient();
    Thread addThread = new Thread(new ThreadStart(a.Run));
    Thread listThread = new Thread(new ThreadStart(l.Run));
```

```
    serverThread.Start();
    addThread.Start();
    listThread.Start();
}

// JAVA
public static void main(String[] args)
{
    Thread s = new Thread(new RunServer(),"server");
    Thread a = new Thread(new AddClient(), "add client");
    Thread l = new Thread(new ListClient(), "list client");
    s.setPriority(Thread.MAX_PRIORITY);
    s.start();
    a.start();
    l.start();
}
```

If you start the threads in the normal way, as in the previous listing, it is not 100 percent guaranteed that the server thread will start first. The following variations, using a `System.Threading.Timer` in C# and a `java.util.concurrent.ScheduledExecutorService` in Java (you need Java 5.0), delay the start of the client threads by a specified time to let the server get started:

```
// C#
public static void Main(string[] args)
{
    RunServer s = new RunServer();
    Thread serverThread = new Thread(new ThreadStart(s.Run));
    serverThread.Start();

    AutoResetEvent autoEvent = new AutoResetEvent(false);
    ThreadStarter threadStart = new ThreadStarter();
    // the TimerCallback delegate specifies the methods
    // associated with a Timer object
    TimerCallback timerDelegate = new TimerCallback(threadStart.StartClients);
    Timer delay = new Timer(timerDelegate, autoEvent, 5000, 0);
    autoEvent.WaitOne(10000, false);
    delay.Dispose();
}

namespace com.db4o.dg2db4o.chapter8
{
    class ThreadStarter
    {
```

```
        // This method is used by the timer delegate.
        public void StartClients(Object stateInfo)
        {
            AutoResetEvent autoEvent = (AutoResetEvent)stateInfo;
            RunAddClient a = new RunAddClient();
            RunListClient l = new RunListClient();
            Thread addThread = new Thread(new ThreadStart(a.Run));
            Thread listThread = new Thread(new ThreadStart(l.Run));
            addThread.Start();
            listThread.Start();
            autoEvent.Set();
        }
    }
}

// JAVA
public static void main(String[] args)
{
    Thread s = new Thread(new RunServer(),"server");
    Thread a = new Thread(new RunAddClient(), "add client");
    Thread l = new Thread(new RunListClient(), "list client");
    ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(3);
    System.out.println("MAIN: Created Scheduler ...");
    ScheduledFuture<?> fs = scheduler.schedule(s,0,TimeUnit.SECONDS);
    ScheduledFuture<?> fa = scheduler.schedule(a,5,TimeUnit.SECONDS);
    ScheduledFuture<?> fl = scheduler.schedule(l,5,TimeUnit.SECONDS);
    while(!fs.isDone()){
        try
        {
            Thread.currentThread().sleep(1000);
        }
        catch(InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }
    fl.cancel(true);
    scheduler.shutdown();
}
```

Output

If you run the example in any of these ways, you can watch the client/server interaction live. The server runs, the `AddClient` takes all your input and sends `Person` objects to the server database, and the `ListClient` periodically shows the full content of the database.

Let's see what the output of this example would look like (the Java version is shown). We will look separately at the output from the server and each client. First, the server:

```
[db4o 5.0.010   2006-02-14 23:38:15]
 Server listening on port: '8732'
SERVER:[1139960296015] Server's running...
[db4o 5.0.010   2006-02-14 23:38:21]
 Client 'user1' connected.
[db4o 5.0.010   2006-02-14 23:38:25]
 Client 'user2' connected.
SERVER:[1139960356015] Server's running...
SERVER:[1139960416015] Server's running...
```

As we started the server first, it shows us each minute that it is alive. Every client connection is also listed. The first connection is the `AddClient` and the second connection is the `ListClient`. Let's look now at their output. First, the `AddClient`:

```
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
Michael 31
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
Sue 24
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
Tim 21
```

And now the `ListClient`:

```
LISTCLIENT: Listing...
LISTCLIENT:[Michael;31]
LISTCLIENT: Listing...
LISTCLIENT:[Michael;31]
LISTCLIENT:[Sue;24]
LISTCLIENT: Listing...
LISTCLIENT:[Michael;31]
LISTCLIENT:[Sue;24]
LISTCLIENT:[Tim;21]
```

The first time around, `ListClient` shows nothing in the database as the user was still typing in the details of the first developer in `AddClient`. You can then see the developers being added one by one to the database.

If you run in threads rather than separate processes, the output will be interleaved.

Embedded Mode

Running db4o in embedded mode is conceptually no different from running it in networking mode, except that you are running both the client and the server in the same virtual machine. If you do plan to use a server within a multithreaded application on a single machine, you will

get better performance in embedded mode than in networking mode, as you avoid the overheads introduced by networking protocols. Communicating in networking mode, with the local host or loopback, uses your machine's TCP/IP protocol stack.

There are some differences you need to be aware of. First, the use of 0 for the port number is taken by db4o as a indication that no networking is to take place (the Internet Assigned Numbers Authority has reserved TCP/UDP Port 0 so that it has no specific purpose). This is simply a matter of specifying the port value 0 in the call to `OpenServer`. Second, user authentication is not required, since everything is taking place within the same application—clients will not be accessing the server via remote connections.

Finally, and most significantly in terms of coding, you need to have an `ObjectServer` *reference* directly available in order to create a database client. Compare the code to create a client in each mode:

```
// C#
ObjectContainer networkClient = Db4o.OpenClient("localhost", 8732, "user1", "pwd");
ObjectContainer embeddedClient = server.OpenClient(); // 'server' is an ObjectServer

// JAVA
ObjectContainer networkClient = Db4o.openClient("localhost", 8732, "user1", "pwd");
ObjectContainer embeddedClient = server.openClient(); // 'server' is an ObjectServer
```

In general, the embedded case is probably more difficult. In networking mode you just specify the host and port, and let the network protocols take care of finding the server. As long as there is a server listening at the specified destination, everything will work. In embedded mode, each thread that wants to open a client will have to have an `ObjectServer` reference to the server object that is instantiated, possibly in a different thread, by a call to `OpenServer`.

This is very easy to do if your server and all your clients are created within the same thread. This is not a very useful scenario, though, and offers no real advantage over a standalone database in local mode (although in Chapter 9 we will do this, for a very specific reason). Usually, you want concurrent access to the server from different threads.

To illustrate the differences, let's modify the example of the "developers" database to work in embedded mode rather than networking mode.

Starting the Server

Previously, we simply opened the server in the `RunServer` process/thread. Now, we want to *register* the server in such a way that all clients can look up a *server registry* to obtain the right server reference. The server is to be started at the time of registration.

A `ServerRegistry` class that allows this is listed here. `ObjectServer` references are held in a `Hashtable` or `HashMap`.

```
//C#
using System;
using System.Collections;
using System.IO;
using com.db4o;
```

```
namespace com.db4o.dg2db4o.chapter8
{
    class ServerRegistry
    {
        private Hashtable servers;

        public ServerRegistry()
        {
            servers = new Hashtable();
        }

        public ObjectServer RegisterServer(String filename, String id)
        {
            lock (this)
            {
                ObjectServer server = Db4o.OpenServer(filename, 0);
                servers.Add(id, server);
                return server;
            }
        }

        public ObjectServer GetServer(String id)
        {
            lock (this)
            {
                return (ObjectServer)servers[id];
            }
        }
    }
}

// JAVA
package com.db4o.dg2db4o.chapter8;

import com.db4o.*;
import java.io.File;
import java.util.Map;
import java.util.HashMap;

public class ServerRegistry
{
    private Map<String, ObjectServer> servers;

    public ServerRegistry()
    {
        servers = new HashMap<String, ObjectServer>();
    }
```

```
    public synchronized ObjectServer registerServer(String filename, String id)
    {
        ObjectServer server = Db4o.openServer(filename, 0);
        servers.put(id, server);
        return server;
    }

    public synchronized ObjectServer getServer(String id)
    {
        return servers.get(id);
    }
}
```

The main method that starts the application running now creates a registry, registers a server with the id "myserver", and passes the registry to all the threads that it starts:

```
// C#
public static void Main(string[] args)
{
    ServerRegistry sr = new ServerRegistry();
    sr.RegisterServer("C:/embeddedserver.yap", "myserver");
    RunEmbeddedServer s = new RunEmbeddedServer(sr);
    RunEmbeddedAddClient a = new RunEmbeddedAddClient(sr);
    RunEmbeddedListClient l = new RunEmbeddedListClient(sr);
    Thread serverThread = new Thread(new ThreadStart(s.Run));
    Thread addThread = new Thread(new ThreadStart(a.Run));
    Thread listThread = new Thread(new ThreadStart(l.Run));
    serverThread.Start();
    addThread.Start();
    listThread.Start();
}

// JAVA
public static void main(String[] args)
{
    ServerRegistry sr = new ServerRegistry();
    sr.registerServer("C:/embeddedserver.yap", "myserver");
    Thread s = new Thread(new RunEmbeddedServer(sr),"monitor server");
    Thread a = new Thread(new RunEmbeddedAddClient(sr), "add client");
    Thread l = new Thread(new RunEmbeddedListClient(sr), "list client");
    s.start();
    l.start();
    a.start();
}
```

Server and Client Threads

We also need to create new versions of the server and client threads that make use of the registry. Each client now needs an `ObjectServer` instance variable and a constructor that uses a `ServerRegistry` to initialize this variable. The relevant code for the new `RunEmbeddedServer` class is shown next. The `Run` method is identical to `RunServer`, shown previously, except that it does not open a new server. This thread now simply monitors the server, rather than having responsibility for starting it. The `using`/`import` statements are the same as for the network mode classes, and are not shown here explicitly.

```
// C#
namespace com.db4o.dg2db4o.chapter8
{
    class RunEmbeddedServer
    {
        private ObjectServer server;
        private bool stop = false;

        public RunEmbeddedServer(ServerRegistry sr)
        {
            server = sr.GetServer("myserver");
        }

        private void Run()
        {
        ...
```

```
// JAVA
package com.db4o.dg2db4o.chapter8;

public class RunEmbeddedServer implements Runnable
{
    private ObjectServer server;
    private boolean stop = false;

    public RunEmbeddedServer(ServerRegistry sr)
    {
        server = sr.getServer("myserver");
    }

    public void run()
    {
    ...
```

Similarly, the new `RunAddEmbeddedClient` needs to get its reference to the server from the registry, and uses this to create a client `ObjectContainer`. The `Run` method is otherwise the same as that in `AddClient` earlier.

```
// C#
namespace com.db4o.dg2db4o.chapter8
{
    class RunEmbeddedAddClient
    {
        private ObjectServer server;
        private ObjectContainer aClient;
        private bool stop = false;

        public RunEmbeddedAddClient(ServerRegistry sr)
        {
            server = sr.GetServer("myserver");
        }

        public void Run()
        {
            lock (this)
            {
                try
                {
                    aClient = server.OpenClient();
                    ...
```

```
// JAVA
package com.db4o.dg2db4o.chapter8;

public class RunEmbeddedAddClient implements Runnable
{
    ObjectServer server;
    ObjectContainer aClient = null;
    private boolean stop = false;

    public RunEmbeddedAddClient(ServerRegistry sr)
    {
        server = sr.getServer("myserver");
    }

    public void run()
    {
        synchronized(this){
            try {
                aClient = server.openClient();
                ...
```

The modifications to the new `RunEmbeddedListClient` are similar. The output is identical to the networking mode example. You would expect a performance increase, although you will not detect this in a simple example like this.

Using a Singleton Registry

The previous example demonstrated one way of making server references available to clients—you can probably think of other ways of doing this. A variation on the previous example is to use the well-known singleton design pattern to ensure that there is only one instance of `ServerRegistry` in memory at any time. This means that client threads can obtain the singleton, and do not need to have a `ServerRegistry` instance explicitly passed to them. The `ServerRegistry` is modified to contain the following static variable and method:

```
// C#
private static ServerRegistry theInstance;

public static ServerRegistry GetInstance()
{
    if (theInstance == null)
        theInstance = new ServerRegistry();

    return theInstance;
}

// JAVA
private static ServerRegistry theInstance;

public static ServerRegistry getInstance()
{
    if (theInstance == null)
        theInstance = new ServerRegistry();

    return theInstance;
}
```

In this case, the main method creates the registry by calling `GetInstance`, rather than with `new`:

```
ServerRegistry sr = ServerRegistry.GetInstance();    // C#

ServerRegistry sr = ServerRegistry.GetInstance();    // JAVA
```

Client threads are intialized with default constructors. They can get the singleton registry simply by calling the `GetInstance` method. `RunEmbeddedAddClient` is shown as an example:

```
// C#
public void Run()
{
    ServerRegistry sr = ServerRegistry.GetInstance();
    server = sr.GetServer("myserver");
    try
    {
        aClient = server.OpenClient();
        ...
```

```
// JAVA
public void run() {
    ServerRegistry sr = ServerRegistry.getInstance();
    server = sr.getServer("myserver");
    try {
        aClient = server.openClient();
        ...
```

Out-of-Band Signaling

There is one problem with the examples you have seen so far in this chapter: there is no way to stop them (other than by interrupting the processes with brute force). We did include the flag `stop` in each class, but so far we have not provided a way to set any flags so that anything does actually stop. This is where out-of-band signaling comes in useful.

Running db4o in networking or embedded mode means that the client `ObjectContainer` is communicating with the server. This communication is defined by the set of 11 methods that the API defines for the `ObjectContainer`, although there are more if you use the `ExtObjectContainer` (see Chapter 10 for details). These commands are as follows:

- Database modification commands: `Delete` and `Set`
- Querying commands: `Get` and two types of `Query` (see Chapters 5 and 6)
- Transactional commands: `Commit` and `Rollback` (see Chapter 9)
- Commands for object activation: `Activate` and `Deactivate` (see Chapter 7)
- Two management commands: `Close` and `Ext` (see Chapters 5 and 10)

This is a bit limiting for working in client/server mode. In many cases more communication is needed between the client and the server. For example, with these commands there is no way for a client to tell the server:

- To shut down.
- To perform a defragment to increase performance and decrease the database file size.

Note that in client/server mode the `Close` method of `ObjectContainer` just closes the client `ObjectContainer`, not the server.

In order to address this need, db4o provides two useful interfaces that can help to send any message from the client to the server, in a separate stream from the data, using a mode known as *out-of-band signaling*.

The first one is the `MessageSender`:

```
// C#
public interface MessageSender
{
   public void Send(Object obj);
}
```

```
// JAVA
public interface MessageSender
{
   public void send(Object obj);
}
```

■ **Caution** These interfaces in the .NET version of db4o do not, at the time of this writing, follow typical naming conventions—if you are a .NET programmer you would probably expect this to be called `IMessageSender`.

This is a very simple interface, and it implies that a `MessageSender` object can send an object—this can be any object at all. So where does this object get sent to? Not surprisingly, it is sent to a `MessageRecipient`, which implements the following interface:

```
// C#
public interface MessageRecipient
{
   public void ProcessMessage(ObjectContainer con, Object message);
}

// JAVA
public interface MessageRecipient
{
   public void processMessage(ObjectContainer con, Object message);
}
```

A `MessageRecipient` simply processes the message object. Processing can be defined to involve anything you like, including calling the `Close` method of a *server*. Both interfaces can be found in the namespace/package `com.db4o.messaging`.

Let's show how this all works by modifying the scenario you have seen throughout this chapter so that the server will shut down in response to a user request issued in the "add" client. We could add a completely separate client just for this purpose, but the "add" client is already dealing with user input, so it will do just fine for this demonstration.

The modifications will work the same way in either networking mode or embedded mode—we'll specifically show networking mode here as this is a more useful scenario. The new features will be as follows:

1. There will be a stop condition in the "add" client, so that when the user enters the string *"stop"* when prompted for a name, the stop flag is set to `true` and the `while` loop is exited.
2. At this point, the "add" client will send message to tell the server to stop.
3. The process that is monitoring the server will receive the message, and respond by stopping the server.

Sounds simple enough—so how do we do it? The process is illustrated in Figure 8-3, and explained in the following sections.

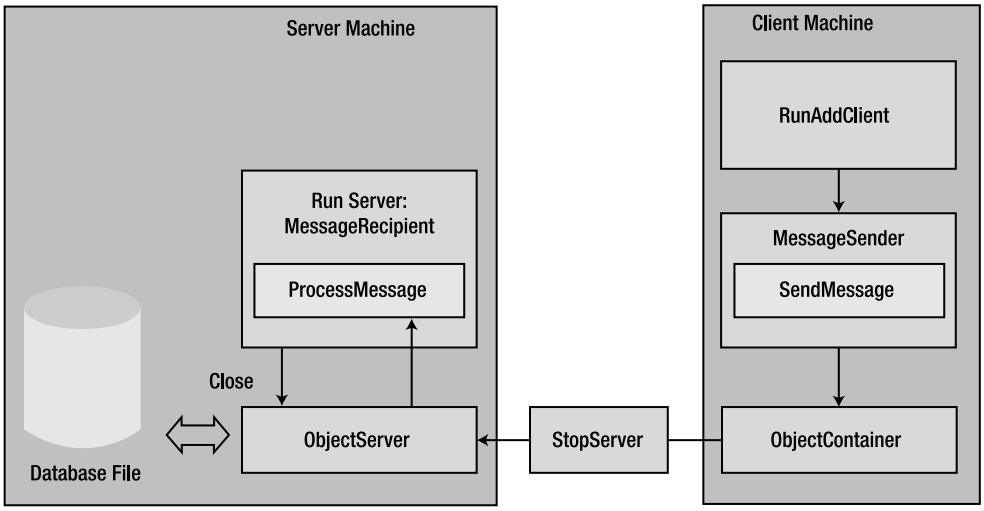

Figure 8-3. *Out-of-band signaling example*

Sending the Message

The basic idea is that the client thread or process gets a `MessageSender` instance from its `ObjectContainer`—messages sent using this will go to the server to which it is connected (by a network or in-process connection). The modified `Run` method, in `RunAddClient` in this example, looks like this:

```
// C#
public void Run()
{
    try
    {
        aClient = Db4o.OpenClient("localhost", 8732, "user1", "password");
        while (!stop)
        {
            Console.WriteLine("\nADDCLIENT: Please enter a name: ");
            String name = Console.ReadLine();
            if (name.Equals("stop"))
            {
                stop = true;
            }
```

```
            else
            {
                Console.WriteLine("\nADDCLIENT: Please enter an age: ");
                int age = Convert.ToInt32(Console.ReadLine());
                aClient.Set(new Person(name, age));
                aClient.Commit();
            }
        }
    }
    catch (ThreadInterruptedException tie)
    {
        Console.WriteLine("Thread Error!" + tie);
    }
    finally
    {
        MessageSender messageSender = aClient.Ext().Configure()
            .GetMessageSender();
        messageSender.Send(new StopServer("ADDCLIENT says stop!"));
        aClient.Close();
    }
 }

// JAVA
public void run()
{
    try
    {
        aClient = Db4o.openClient("localhost", 8732, "user1", "password");
        while (!stop)
        {
            System.out.print("\nADDCLIENT:Enter the developer's name and age (e.g.
                'Tom 44'): ");
            String name = sc.next();
            if(name.equals("stop"))
            {
                stop = true;
            }
            else
            {
                aClient.set(new Person(name, sc.nextInt()));
                aClient.commit();
            }
        }
    }
```

```
    catch (Exception e)
    {
        e.printStackTrace();
    }
    finally
    {
        MessageSender messageSender = aClient.ext().configure()
            .getMessageSender();
        messageSender.send(new StopServer("\nADDCLIENT says stop!"));
        aClient.close();
    }
}
```

The `MessageSender` is obtained using the `GetMessageSender` method that belongs to the `ObjectContainer`, named `aClient` in this code (you will understand the `Ext().Configure()` part of this method call when you have read Chapters 9 and 10, but it doesn't matter exactly how this works here).

Once obtained, the `MessageSender` is used to send an object that is an instance of a class called `StopServer`. This class is simply a holder for a string that allows the client to send a comment with the message. The fact that this particular message is intended to stop the server is indicated by the class of object sent. `StopServer` is pretty simple, but for completeness, here is a listing:

```
// C#
namespace com.db4o.dg2db4o.chapter8
{
    class StopServer
    {
        private string _info;

        public StopServer(string info)
        {
            _info = info;
        }

        public override string ToString()
        {
            return _info;
        }
    }
}

// JAVA
package com.db4o.dg2db4o.chapter8;

public class StopServer {
    private String _info;
```

```
    public StopServer(String info) {
        _info = info;
    }

    public String toString(){
        return _info;
    }
}
```

■ **Note** You can, if you wish, specify much more information or functionality in a message object than this. In some cases it might be useful to implement the Command design pattern, in which actions and their parameters are encapsulated into command objects. This is one of the "Gang of Four" design patterns, described in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).

Processing the Message

Now that the message object has been sent, you will want to process it—that is, check that it is an instance of `StopServer` and if so, stop the server and print out the comment it contains. You do so by making the thread or process that is monitoring the server implement the `MessageRecipient` interface by providing a `ProcessMessage` method.

Crucially, the thread or process must also register itself with the actual `ObjectServer` as the `MessageRecipient` for messages sent to that server by its clients. It does so by calling the `SetMessageRecipient` method that belongs to the `ObjectServer`, passing a `this` reference to itself as the parameter.

The class declaration, of `RunServer` in this example, is now:

```
class RunServer : MessageRecipient    // C#

class RunServer implements MessageRecipient, Runnable    // JAVA
```

The modified `Run` method and the new `ProcessMessage` method look like this:

```
// C#
private void Run()
{
    Console.WriteLine("Starting server...");
    ObjectServer server = Db4o.OpenServer("c:/netserver.yap", 8732);
    server.GrantAccess("user1", "password");
    server.GrantAccess("user2", "password");
    server.Ext().Configure().SetMessageRecipient(this);
    lock (this)
    {
```

```
        try
        {
            while (!stop)
            {
                Monitor.Wait(this, 60000);
                Console.WriteLine("SERVER: Server is listening ...");
            }
        }
        catch (ThreadInterruptedException tie)
        {
            Console.WriteLine("Thread Error!" + tie);
        }
        finally
        {
            server.Close();
        }
    }
    Console.WriteLine("Server ends...");
}

public void ProcessMessage(ObjectContainer con, Object message)
{
    lock (this)
    {
        if (message is StopServer)
        {
            Console.WriteLine("\nSERVER:" + message);
            stop = true;
            Monitor.PulseAll(this);
        }
    }
}

// JAVA
public void run() {
    System.out.println("Starting server...");
    ObjectServer server = Db4o.openServer("c:/netserver.yap", 8732);
    server.grantAccess("user1", "password");
    server.grantAccess("user2", "password");
    server.ext().configure().setMessageRecipient(this);
    synchronized (this) {
        try {
            while (!stop) {  // Out of band messages here later...
                System.out.println("\nSERVER:[" + System.currentTimeMillis()
                + "] Server's running... ");
                this.wait(60000);
            }
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("\nSERVER:[" + System.currentTimeMillis()
            + "] Server's stopped! ");
            server.close();
        }
    }
}

public void processMessage(ObjectContainer con, Object message) {
    synchronized (this) {
        if(message instanceof StopServer){
            System.out.println("\nSERVER:" + message);
            stop = true;
            this.notify();
        }
    }
}
```

When the message is received by the server, the `ProcessMessage` method is executed. If the object that was sent was indeed a `StopServer` object, the `stop` flag is set. The thread is notified so that the stop condition is evaluated immediately and it breaks out of the `while` loop, finally calling `Close` on the server.

Here's an example of the output from a test run (Java version), showing the output from the server and a client. First, the server:

```
[db4o 5.0.010   2006-02-16 09:33:05]
 Server listening on port: '8732'
SERVER:[1140082385708] Server's running...
[db4o 5.0.010   2006-02-16 09:33:10]
 Client 'user1' connected.
[db4o 5.0.010   2006-02-16 09:33:14]
 Client 'user2' connected.
SERVER:[1140082435708] Server's running...
SERVER:ADDCLIENT says stop!
[db4o 5.0.010   2006-02-16 09:33:30]
 Connection closed by client 'user1'.
[db4o 5.0.010   2006-02-16 09:33:30]
 'c:/netserver.yap' close request
[db4o 5.0.010   2006-02-16 09:33:30]
 'c:/netserver.yap' closed
```

And now the `AddClient`:

```
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
Michael 31
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
Sue 24
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
Tim 20
ADDCLIENT:Enter the developer's name and age (e.g. 'Tom 44'):
stop
```

Note that the next time the "list" client tries to access the server it will fail to do so, and will close after handling the resultant exception. It would be nice to be able to have the server send messages to all connected clients—for example, to warn them that it is about to shut down—but sending messages from server to clients is not supported at the time of this writing.

Summary

You have seen that the client/server mode in db4o is quite simple. Everything is done via configuration in the code. The networking mode allows connections, with authentication, from remote clients, whereas the embedded mode works for concurrent access by multiple clients within the same virtual machine. Finally, you have learned how a client can communicate with the server "out of band."

In the real world a lot of companies use these kinds of client/server communication features. For example, MR Controls, based in Calgary, uses this db4o feature extensively. They build SCADA (Supervisory Control and Data Acquisition) systems for power utilities and the gas and oil industry.

In the next chapter you will learn about database transactions in db4o, and in particular about how they work in the case of concurrent clients.