O'REILLY® Short Cuts

Darkstar: The Java Game Server

by Brendan Burns

ISBN: 978-0-596-51484-6

Released: August 24, 2007

So, you have a great idea for the next big multiplayer game. Maybe it's a virtual world based on your favorite sci-fi television show. Or maybe it's an online bowling league for you and your friends. Regardless, the challenge of building a networked multiplayer computer game goes far beyond having a great idea. It can be so significant that it prevents great games from becoming reality.

Darkstar breaks down this barrier of complexity. It provides an easy-to-use library of functions that handles the challenging aspects of networked game development for you. Further, it provides a robust, industrial-strength server that can scale with your game as it grows in popularity. With Darkstar, you can quickly turn your idea for a multiplayer game into a (virtual) reality.

Contents

Introduction

Multiplayer Games

So, you have a great idea for the next big multiplayer game. Maybe its a virtual world based on your favorite sci-fi television show. Or maybe its an online bowling league for you and your friends. Regardless, the challenge of building a networked multiplayer computer game has traditionally been greater than simply having a good idea. In many cases, the challenge is so significant that it has prevented great games from being played. Darkstar breaks down this barrier of complexity. It provides an easy-to-use library of functions that handles the challenging aspects of networked game development for you. Further, it provides a robust, industrial-strength server that can scale with your game as it grows in popularity. With Darkstar, you can quickly turn your idea for a multiplayer game into a (virtual) reality.

One of the most significant developments in computer gaming in the past decade has been the explosion of networked multiplayer computer games. From massively multiplayer role-playing games that create virtual worlds (or even solar systems) for players to explore, to people playing online chess, poker or other traditional games, online gaming has broadly impacted the games that we play and who we play against. Multiplayer games offer some of the most rewarding gaming experiences. Despite continual advances in artificial intelligence, it is still quite far from replicating the challenge and pleasure of playing against another human. However, for most of their history, server-based multiplayer computer games have been the domain of large game companies. This is largely due to the resources and expertise required to create a successful and robust multiplayer networked game. You might have an idea for a great game, but not the expertise to develop it, or the resources to support the thousands of players who would flock to your game. Wouldn't it be great if someone else did the hard work to make a implementing a multiplayer networked game easy? And wouldn't it be even better if the game you built could easily and rapidly scale as more and more people started playing your game? Darkstar, an application server for gaming does this and more. Darkstar frees you from the mundane task of writing complex code to support many, many players simultaneously playing your game from all over the Internet. Instead you can focus on the important (and fun!) part: creating the next big multiplayer gaming sensation!

So how does Darkstar do this? Darkstar is an *application server* for building computer games. For a game programmer, it provides a library of functions that implement the services required to build a successful multiplayer game. It also provides a robust server architecture that is reliable and scales well as the number

of user grows. This Short Cut explores the details of the Darkstar game server and provides examples of how it can be used to build a fully-featured multiplayer game.

The Architecture of Multiplayer Games

At first glance, a network multiplayer game can seem like an intimidatingly complex software system. However, in reality, a multiplayer game is really just a carefully organized orchestration of a number of relatively simple components. This section discusses a general high-level architecture for networked multiplayer games. Although individual games may differ slightly in the details of their implementation, this general framework provides a unified context for understanding multiplayer network games. This section describes a classical client-server architecture for computer games. Although peer-to-peer (P2P) gaming has been the subject of much academic research, such technologies have yet to see widespread deployment, and are not supported by the Darkstar application server. As such, they are beyond this Short Cut's scope.

There are many different levels of abstraction for viewing a multiplayer network game. The highest level of abstraction is that of the client and server. Each *client* manages a single player's interactions with the game. The *server* mediates the interaction between all of these clients and acts as the game's referee. In any multiplayer game, there are many different players (the "multi" in "multiplayer"). Each of these players interacts with the client application running on his computer through keyboard, joystick or mouse. The client application, in turn, communicates these interactions ("moves") to a single server over the network (either local or Internet). The game server organizes the game by maintaining the *game state*, which represents with absolute certainty, the current state of the entire game world.

In a single *turn* of the game, the game server receives moves from each of the clients, applies the rules of the game to update the game state given the client's moves, and sends this new game state back to the client. Upon receiving the new game state, the client displays it for the user and prepares to accept new user moves. Thus, the algorithm for a basic game loop is as follows:

1. Client connects to server
2. Client receives game state from server
3. Client displays game state to user
4. Client sends user input to server
5. Server receives client input and updates game state using game rules

6. Server sends new game state to client
7. Repeat 2–6 until client disconnects

Within the client, there are four basic pieces of functionality:

- User input
- Sending moves to the server
- Receiving game state from the server
- Graphical display

Within the server there are three basic pieces of functionality:

- User input
- Applying game logic
- Broadcasting the new game state

The interactions of these components is shown in **Figure 1**.

Maintaining the game state on the server centralizes game information in a single location. This dramatically simplifies the process of updating the state of the game given a set of moves from the clients. It also eliminates any possible disagreement between the clients about the state of the game. The central game server is always right. This centralization also dramatically reduces the possible ways in which a client/player can cheat. Because the game server applies the rules of the game, it can ensure that the movements of every player obeys the rules.

An Application Server for Games

There are many challenges to programming successful multiplayer games. These challenges can be succinctly separated into four categories:

- Communication
- Concurrency
- Persistence
- Scaling

The first two of these involve the practical details of getting a multiplayer game to work. Communication involves the network dialogue between client and server. Concurrency deals with the issues surrounding multiple clients simultaneously accessing and modifying the shared game state. The second two categories involve implementing a robust and reliable game. Persistence ensures that the game state is stored and saved even if the server crashes. Scaling ensures that your game serv-

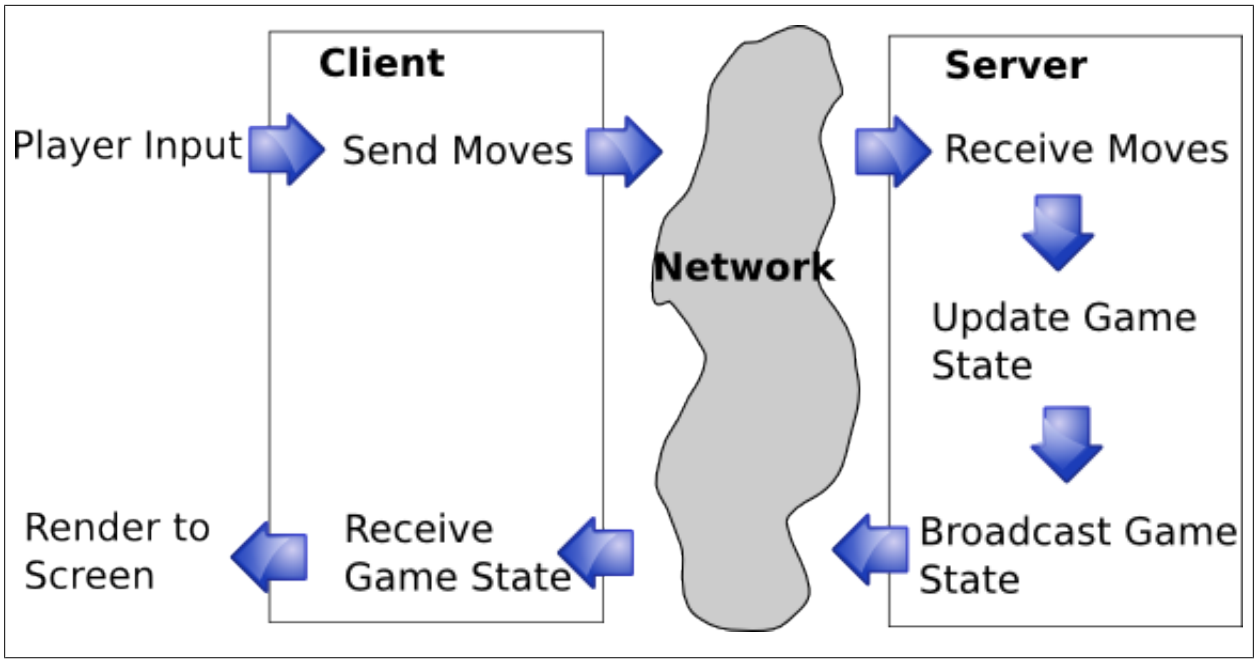

Figure 1. High-level diagram of client-server interaction

er's performance is maintained even with thousands of simultaneous users. Each of these challenges are significant and difficult to implement correctly. Together, they often stand as an insurmountable barrier to individual programmers implementing their own multiplayer games. Moreover, implementing network communication and concurrency correctly isn't really the fun part. What we're really interested in is writing games!

Fortunately, Darkstar is an *application server* designed for gaming. The purpose of an application server is to provide robust, efficient, and easy-to-use implementations of commonly required functionality (in this case a library of functions for communication, concurrency, persistence, and scaling) so that users can focus on implementing the functionality specific to their application (in this case your game!). The functionality provided by the Darkstar application server allows individual game programmers to focus solely on the game they're developing. The Darkstar application server also allows you to rapidly develop a robust multiplayer networked game in a short period of time. Additionally, it provides confidence that your program will scale to many players.

The Darkstar Game Server

This section explains how to install the Darkstar game server. The Darkstar game server supports the following hardware/OS combinations:

- Windows XP on 32-bit processors
- Mac OS X 10.4.x on PowerPC or Intel processors

- Red Hat/Fedora Core Linux on 32-bit x86 processors

On some platforms (Windows and Linux) you may need to install or upgrade the Java Virtual Machine (JVM). Darkstar requires J2SE version 5.0 or higher. On OS X or Solaris, the correct Java virtual machine is already installed. Once your JVM is up to date, you need to download the game server itself. Darkstar can be downloaded from ***http://www.projectdarkstar.com***. At the time of writing, the latest version was 0.9.1. Darkstar is distributed as a *.zip* file. Once you have downloaded it, simply unpack it in the directory where you want it to reside. The final step to installing the game server is to set an environment variable to point to the location of the game server. On Windows, create a system variable named "sgshome" through the "System" control panel. Set the value of the variable to the location of the game server's directory. On Mac OS X and Linux you need to modify the *.bashrc* file in your home directory to include the line export **`SGSHOME=`*`location`***, where *location* is the actual location of the game server's directory. Once you have completed this step, Darkstar is successfully installed. It's time to write your first game!

Your First Game

The Darkstar application server can simultaneously run many different games or many different instances of the same game. To achieve this, Darkstar relies on these different games to implement a common interface: `com.sun.sgs.app.AppListener`. The `AppListener` interface requires the following methods to be implemented:

`-void initialize(java.util.Properties properties)`

This method is called when the class is first initialized. The `Properties` object contains initialization properties loaded from a game properties file. This file is discussed later on in this section.

> **Note**
>
> A game server's implementation of the `AppListener` class is stored persistently to a file when Darkstar exits (either by being shutdown or crashing). As a result, this method is only run the *very first time* Darkstar starts. To force Darkstar to re-initialize your class you need to erase the object cache from the filesystem. For more on this see the section on `ManagedObjects` and persistence (**Managing Game State with Persistent Objects**).

`ClientSessionListener loggedIn(ClientSession session)`

> This method is called when a client logs into the game server. This method returns a implementation of the `ClientSessionListener` interface, which is used for subsequent interactions with the client. If this method returns null, the game server rejects the client's attempt to login. The next section discusses the `ClientSessionListener` and `ClientSession` objects in detail.

Unlike traditional applications, your game server doesn't have a `main(...)` method. Instead, Darkstar uses an *event-driven* model. Your game implements a series of listener interfaces, such as `AppListener`. These listener interfaces are used in similar manner to the event listeners (e.g.,`java.awt.event.ActionListener`) in Java GUI programming. When a particular event occurs, such as a client logging into your game server, the Darkstar server calls the `loggedIn(...)` method in your implemention of the `AppListener` event listener interface. There are several different listener interfaces used in the Darkstar application server, they are discussed in turn in later sections.

In addition to implementing the `AppListener` interface, your main game class should also implement the `java.io.Serializable` interface so that it can be persistently stored to file. `Serializable` is a marker interface, it does not require any methods be implemented. However, all of the instance variables in your main game class must be `Serializable` as well. Serializing game server objects will also be discussed in detail in the **Managing Game State with Persistent Objects** section.

The following is a listing for a simple (and silly) game. It does nothing but print a message when initialized. It also prints a message when a client attempts to connect and rejects all clients.

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.*;

public class SillyGameMain implements AppListener, Serializable
{
    // Initialize the game
    // prints a message on initialization
    public void initialize(Properties properties) {
        System.out.println("Game server initialized!");
    }

    // Handle a client login.
    // Prints a message and rejects login by returning null.
    public ClientSessionListener loggedIn(ClientSession session) {
        System.out.println("Client login attempt!");
        return null;
```

```
    }
}
```

To compile the code use `sgs.jar`, which is in the `lib` directory of the main Darkstar directory.

In order to run your game, Darkstar needs to know the name of the class that implements the `AppListener` interface. This is accomplished through the *game properties* file. The game properties just a standard Java properties file. Java properties files are text files that contain property definitions. Darkstar requires that four properties be declared:

`com.sun.sgs.app.name`

The text name of the game. This is can be any string.

`com.sun.sgs.app.listener`

The name of the main game class which implements the `AppListener` interface.

`com.sun.sgs.app.port`

The number of network port on which the server listens for connections.

`com.sun.sgs.app.root`

The root directory where the game's persistent classes should be saved. See the **Managing Game State with Persistent Objects** section.

Here is *sillygame.properties*, a properties file that runs `SillyGameMain`:

```
# Silly Game Properties (use # for comments in properties files)
com.sun.sgs.app.name=A Silly Game
com.sun.sgs.app.listener=SillyGameMain
com.sun.sgs.app.port=1221
com.sun.sgs.app.root=silly_game_data
```

> **Note**
>
> The directory specified in the `com.sun.sgs.app.root` property must exist and it must contain a directory named "`dsdb`". Darkstar will not currently create these directories for you. If either of these directories do not exist, Darkstar will not be able to start your game.

Once you have created `sillygame.properties`, you can run your game using the `sgs.sh` script (`sgs.bat` on Windows). The script is located in the main game server directory. The script takes two arguments. The first argument is the a classpath that must contain the main game class specified in the `com.sun.sgs.app.listener` property. The second argument is the properties file, in this case

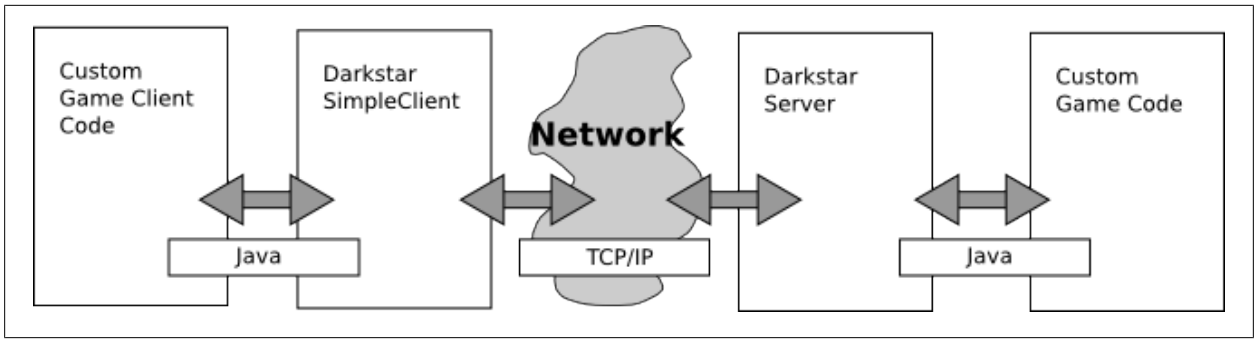

Figure 2. The basic pattern of communication in Darkstar

`sillygame.properties`. For example (assuming that you compiled SillyGameMain into a directory named "classes"):

```
sgs.sh ./classes sillygame.properties
```

As the server starts up, it prints a number of status messages, including the message from `SillyGameMain`'s `initialize` method. The final message indicates that the SillyGame server is successfully up and running. Of course, this silly game rejects all client logins. That's not very fun. In the next section we'll explore how to create clients, handle client login, logout, and communicate, and begin to create a full-fledged game.

Clients and Client Communication

Although the server is the heart of any multiplayer game. It doesn't accomplish much without clients to play it. In order to interact with the game and each other, clients communicate with the server over the network. Developing a network communication protocol can be complicated and time-consuming. Fortunately, Darkstar provides a great deal of library code that dramatically simplifies network communication between server and clients. Darkstar provides functions to help implement both the server side and the client side communication. In both cases, the user provides custom implementations of event-handling interfaces that interact with the Darkstar communication code to implement the networked interactions between your game client and your game server. This interaction is illustrated in **Figure 2**.

The first section describes how the Darkstar server accepts connections from a client, receives commands from the client, sends game state back to the client, and handles client disconnection (either graceful or forced). The second section describes Darkstar's client-side communication: establishing a connection, sending messages, receiving game state, and disconnecting from the server. The **Client-Server Authentication** section discusses client authentication, and **Managing Game State with Persistent Objects** describes how to send Java objects across

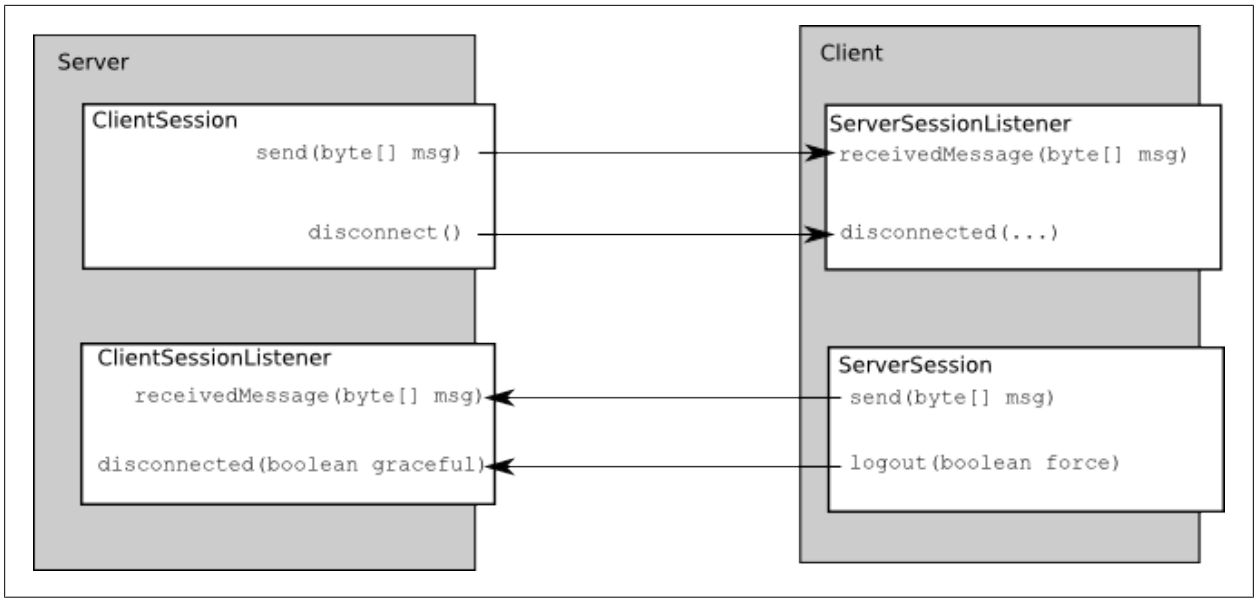

Figure 3. The communication interfaces and their relationships

the network. Over the course of the section, a complete example of client-server communication (I wouldn't call it a "game" just yet...) is developed.

Basic communication between client and server in Darkstar is organized into *Session* and *SessionListener* interfaces. Session objects are used to initiate communication, either to send data, or to log out of the server. SessionListener objects are listeners that receive notification of communication events when data is received or a client logs out. In Darkstar, a pair of Session and SessionListener objects exist on the server and on the client. They are named by what they communicate with, thus the Session and SessionListener objects on the client are called `ServerSession` and `ServerSessionListener` respectively because they listen to the server. Likewise, the objects on the server are `ClientSession` and `ClientSessionListener`. The pattern of communication between these interfaces is illustrated in **Figure 3**. The following sections describe this communication in detail.

Server-Side Client Handling

The implementation of the `AppListener` interface returned null from its `public void loggedIn(...)` method. This meant that although clients could attempt to connect, those connection attempts would fail. Thus, the first step in implementing client handling on the server side is returning a non-null `ClientSessionListener`. This `ClientSessionListener` is an implementation of the interface specific to your particular game. The `ClientSessionListener` interface handles events on the server related to a particular client. It is notified whenever the client sends a message to the server and when the client disconnects. Clients can disconnect gracefully, meaning they tell the server they are going to disconnect, or they can disconnect

ungracefully and simply disappear from the server, either due to some sort of network failure or the client program being terminated abruptly. For each client that connects to the server, a different ClientListener should be created. The `ClientSessionListener` interface has two methods that require implementation:

`public void receivedMessage(bytes[] message)`

This method is called when the server receives a message from a particular client. The raw bytes sent by the client are send as the message parameter to this method.

`public void disconnected(boolean graceful)`

This method is called when the client disconnects. The graceful parameter is true if the client requested disconnection and was subsequently disconnected. If graceful is false, the client disconnection was forced because the network connection was severed, for example by forcefully quitting the client or network failure.

The following is a simple implementation of a `ClientSessionListener`, which just prints the message sent by the client to the console:

```
import com.sun.sgs.app.*;
import java.io.Serializable;

// A simple client listener implementation
public class SimpleClientListener
    implements ClientSessionListener, Serializable
{
    protected ClientSession client;

    // Constructor
    public SimpleClientListener(ClientSession client) {
           this.client = client;
    }

    // Prints the message which was received (assumes it is a String)
    public void receivedMessage(byte[] message) {
           System.out.println(new String(message));
    }

    // React to client disconnection.
    public void disconnected(boolean graceful) {
           if (graceful) {
               System.out.println("Graceful disconnect from client.");
           }
           else {
               System.out.println("Forced disconnect from client.");
           }
```

```
    }
}
```

To use this `SimpleClientListener` to handle client connections, the implementation of `AppListener` is modified to return an instance of the `SimpleClientListener`, is shown in the following code. Note that the only difference between `SimpleGameMain` and `SillyGameMain` is the `loggedIn` method.

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.*;

public class SimpleGameMain implements AppListener, Serializable
{
    // Initialize the game
    // prints a message on initialization
    public void initialize(Properties properties) {
        System.out.println("Game server initialized!");
    }

    // Handle a client login.
    // Prints a message and accepts the login.
    public ClientSessionListener loggedIn(ClientSession session) {
        System.out.println("Client login attempt!");
        return new SimpleClientListener(session);
    }
}
```

So far, the server can accept client connections and receive data from the client. To send data back to the client, the server needs to use methods in the `ClientSession` class. For every client that is connected to the server, the server stores a unique `ClientSession` object associated with that client. `ClientSession` objects are used for server initiated actions for a particular client. You can use the `ClientSession` to send data to the client, to determine if it is still connected to the server and to forcibly disconnect the client. You can also use the `ClientSession` object to obtain information about the particular client, you can get the client's login name as well as the `ClientSessionId` object which represents the client's session identifier. Each time a client connects to the server, the server creates a new `ClientSession` object and passes it to the `AppListener`'s `loggedIn` method. Here is a detailed description of the `ClientSession`'s methods:

`public void send(byte[] message)`

Sends an array of raw bytes to the client

`public String getName()`

Returns the name of the client which is represented by this session.

`public ClientSessionId getSessionId()`

This function returns the `ClientSessionId` object which represents the identifier for this client's session

`public boolean isConnected()`

Tests if the client still connected to the server?

`public void disconnect()`

Forcibly disconnect the client from the server.

Its a good idea to have your implementation of `ClientSessionListener` take the `ClientSession` as parameter to its constructor. That way you can store the `ClientSession` for a client in your `ClientSessionListener` implementation. Once stored, you can use the `ClientSession` object to interact with the client whenever you need to. Most often, this means using the `ClientSession` to send data back to the client. For example, if we change the `receivedMessage(...)` method as follows, it will echo whatever it received back to the client:

```
import com.sun.sgs.app.*;
import java.io.Serializable;

// A simple client listener implementation
public class EchoClientListener
    implements ClientSessionListener, Serializable
{
    protected ClientSession client;

    // Constructor
    public EchoClientListener(ClientSession client) {
           this.client = client;
    }

    // Prints the message which was received (assumes it is a String)
    public void receivedMessage(byte[] message) {
           System.out.println(new String(message));
           client.send(message);
    }

    // React to client disconnection.
    public void disconnected(boolean graceful) { ... }
}
```

Client-Side implementation

In addition to providing a library of functions for client communication on the server-side, the Darkstar application server also provides a library of functions to simplify the implementation of the client application. This section describes how

to use these functions. It is worth noting, that this client is only provided as a convenience. A game developer is free to implement their own client if it suits them.

ServerSessionListener

Like the `ClientServerListener`, implementing a client requires implementing the `ServerSessionListener` interface. The `ServerSessionListener` listens for communication events from the server, including receiving messages and disconnection. The `ServerSessionListener` interface requires the implementation of five methods:

`void disconnected(boolean graceful, String reason)`

This method is called when the client is disconnected for some reason. If graceful is true it is because the client requested disconnection. If it is false, it is for some other reason, such as network failure or the server evicting the client. The reason string provides a description of why the client was disconnected.

`void reconnecting()`

This method is called when the server the client is currently communicating with has failed and the client is attempting to fail-over to a back up server.

`void reconnected()`

This method is called when the client successfully reconnects to the server after fail-over has occured.

`void receivedMessage(byte[] msg)`

This method is called when the client receives data from the server.

`ClientChannelListener joinedChannel(ClientChannel channel)`

This method is called when a client joints a communication channel.

Here is an example implementation of a `ServerSessionListener` which simply prints the data received from the server as well as status messages for every event:

```
import com.sun.sgs.client.*;

public class MyServerSessionListener implements ServerSessionListener {
   public void disconnected(boolean graceful, String reason) {
       System.out.println("The client was "+(graceful?"gracefully":"forceably")+
                          " disconnected from the server.  Reason: "+reason);
   }

   public void receivedMessage(byte[] msg) {
       System.out.println("Received message: ");
       System.out.println(new String(msg));
   }
```

```
    public ClientChannelListener joinedChannel(ClientChannel channel) {
        System.out.println("Joined channel: "+channel.getName());
        return null;
    }

    public void reconnected() { System.out.println("Reconnected to server."); }
    public void reconnecting() { System.out.println("Reconnecting to server."); }
}
```

ServerSession

Now that we can receive data from the server, we're almost done, but to make a complete game, we need to send data to the server as well. The `ServerSession` interface provides this functionality. `ServerSession` has two methods for communicating with the server:

`void send(byte[] message)`

: Sends a message to the server, may throw `java.io.IOException` if a communication error occurs.

`void logout(boolean force)`

: Logs the client out from the server. If force is true, the client attempts to log out gracefully, but will forcibly log out, dropping the connection, even if the server doesn't respond to the log out request. If the client successfully logs out, either forcibly or gracefully, the `disconnected(...)` method in the client's `ServerSessionListener` is notified.

`ServerSession` also has two methods for inquiring about the state of the client:

`boolean isConnected()`

: Queries if the client is currently connected to the server, returns true if the client is connected.

`SessionId getSessionId()`

: Obtains the identifier object describing this particular session.

SimpleClient

To make your life easier, the Darkstar application server provides the `com.sun.sgs.client.simple.SimpleClient` class. This class implements the `ServerSession` interface. To construct an instance of `SimpleClient`, you need to pass a game-specific implementation of the `SimpleClientListener` interface to the `SimpleClient` constructor. The `SimpleClientListener` interface extends `ServerSessionListener` and adds three methods related to client login:

`PasswordAuthentication getPasswordAuthentication()`

Used by `SimpleClient` to obtain the username and password for logging in to the server. This method should query the user and return a `java.net.PasswordAuthentication` object containing the appropriate username and password.

`void loggedIn()`

This method is called when the client successfully logs into the server.

`void loginFailed(String reason)`

This method is called when login to the server fails. The reason string provides the reason for the failure.

The following is a complete implementation of the `SimpleClientListener` interface, which extends the previous implementation of the `MyServerSessionListener` and adds support for the `waitForLogin` method described below. This implementation queries the users for a user name and password using the `javax.swing.JOptionPane` which displays a password in clear text. Obviously, a more sophisticated login dialog would use a text field that hides the password.

```
import com.sun.sgs.client.*;
import com.sun.sgs.client.simple.*;
import java.io.*;
import java.net.PasswordAuthentication;
import javax.swing.JOptionPane;

public class MySimpleClientListener
  extends MyServerSessionListener
  implements SimpleClientListener
{
    protected Object loginLock;

    public MySimpleClientListener() {
       loginLock = new Object();
    }

   public PasswordAuthentication getPasswordAuthentication() {
       String user, password;

       user = JOptionPane.showInputDialog(null, "Please enter a username", "");
       password = JOptionPane.showInputDialog(null, "Please enter a password", "");
       char[] pw = new char[password.length()];
       password.getChars(0,password.length(), pw, 0);
       return new PasswordAuthentication(user, pw);
   }

   public void loggedIn() {
       System.out.println("Client logged in successfully.");
```

```
        synchronized (loginLock) {
           loginLock.notifyAll();
        }
    }

    public void loginFailed(String reason) {
        System.out.println("Login failed.  Reason: "+reason);
        synchronized (loginLock) {
           loginLock.notifyAll();
        }
    }

     public void waitForLogin() throws InterruptedException {
         synchronized(loginLock) {
            loginLock.wait();
         }
     }
}
```

Once you have finished implementing the `SimpleClientListener` interface, you can use the interface to construct an instance of the `SimpleClient` class. This `SimpleClient` can then be used to initiate and maintain communication with the server. `SimpleClient` provides the following functions for client-side communications:

`void login(java.util.Properties properties)`

Initiates a client login to the server using the information in `connectionProperties`. This method expects two properties to be set: The property "host" should map to the hostname or IP address of the server the client should connect to. The property "port" should map to the port number that the client should use. Both properties should map to String objects.

`void logout(boolean force)`

Requests a logout from the server. If force is true, the client is forcibly disconnected from the server whether or not an acknowledgement from the server is received.

`boolean isConnected()`

Tests if the client is currently connected to the server.

`void send(byte[] msg)`

Sends bytes to the server, where it is received by the `ClientListener's received(...)` method.

`SessionId getSessionId()`

Returns the session identifier associated with this client's connection to the server.

Once an instance of the `SimpleClient` is constructed, the login method is used to initiate a connection to the server. Once the connection is established, the client goes into the game loop: repeatedly querying user input, sending data to the server, receiving data from the server, and displaying it to the user.

> **Note**
>
> The `login` method is asynchronous. This means that it returns immediately after it has been called and *before* the client has successfully completed its login attempt. However, in most common clients, you want the client to wait until after the user is successfully logged in before you start doing anything else. As a result, I have found it useful to add the following method to my implementation of `SimpleClientListener`:
>
> ```
> void waitForLogin() throws InterruptedException
> ```
>
> This method waits until either the client successfully logs in, or the login fails. You can use `waitForLogin` to wait until the client has successfully logged in, or the login has failed.

The following is an implementation of a main routine for a game client that uses `MySimpleClientListener` and the `SimpleClient` class to communicate with the server:

```
import com.sun.sgs.client.simple.SimpleClient;
import java.util.Properties;
import java.io.*;
import javax.swing.JOptionPane;

public class MySimpleClient {
   //  Usage: java MySimpleClient <host> <port>
   public static void main(String[] args) {
     // Create a client listener and client
     MySimpleClientListener listen = new MySimpleClientListener();
     SimpleClient sc = new SimpleClient(listen);

     // Create the login properties and login.
     Properties connectionProperties = new Properties();
     connectionProperties.setProperty("host", args[0]);
     connectionProperties.setProperty("port", args[1]);
     try {
         sc.login(connectionProperties);
     } catch (IOException ex) {
         ex.printStackTrace();
         System.exit(1);
     }
     // Wait for login to complete.
```

```
    try {
      listen.waitForLogin();
    } catch (InterruptedException ignore) {}

    // Read commands and send them to server.
    // Logout when the user types 'logout'
    String cmd = null;
    do {
      cmd = JOptionPane.showInputDialog(null, "Please enter a command", "");
      try {
        sc.send(cmd.getBytes());
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    } while (!"logout".equals(cmd));
    sc.logout(false);
  }
}
```

Taking all of these classes together, we have the makings of a complete game. Compile all the classes, using *<sgs-home>/lib/sgs.jar* for the server classes and *<sgs-home>/lib/sgs-client.jar* for the client classes. To run the server, you need to create a properties file. Once the properties file is created, you can run the server with the command:

```
sgs <classpath of classes> <server properties file>
```

In a different console window, you can run the client:

```
java -classpath <classpath of classes and sgs-client.jar> MySimpleClient
       <server-hostname> <server-port>
```

When the client starts up, you will be prompted for a user name and password, then for messages to send the server. If everything is working correctly, you should see the commands being sent to the server, and, if you run the echo server, the messages being sent back to the client.

Client-Server Authentication

In most multiplayer games, a player logging on to a server logs in with a user name and password. This enables the game to save and restore information particular to the player. For board games, this might simply be a cumulative win-loss record of games played so far and perhaps a position in a ranking system. For more complex games, like online role-playing games, there is much more information stored: the inventory, skills and health of the player as well as the player's location in the world and the player's class and guild associations. In either of these cases, identifying a user at login time requires authenticating that they are who they claim to

be. This is accomplished by having the user present a password as well as a user name at login time.

On the client-side, authentication is handled by the `public java.net.PasswordAuthentication getPasswordAuthentication()` method in the `SimpleClientListener` interface. This method queries the user, creates and returns a `PasswordAuthentication` object containing a user name and password for logging into the server.

By default, games running in Darkstar do not perform any authentication. Any password presented by the user is accepted as valid. Although this is useful in a development environment, it isn't desirable in the real world full of truly villianous villians! Fortunately, Darkstar also provides support for basic name/password authentication, as well as any custom authentication scheme your game might need.

Basic authentication is handled by the `com.sun.sgs.impl.auth.NamePasswordAuthenticator` class. To use this authenticator, you need to create a flat file containing name/password pairs. Darkstar also provides a utility class (`com.sun.sgs.impl.auth.PasswordFileEditor`) for creating these files. Usage of the tool is as follows (assuming that *sgs.jar* is in your classpath):

```
java com.sun.sgs.impl.auth.PasswordFileEditor <password-file> <user-name> <password>
```

Once your password file is created, you can edit your *server.properties* file to include:

```
# Use the name/password authenticator
com.sun.sgs.app.authenticators=com.sun.sgs.impl.auth.NamePasswordAuthenticator
# Use your password file
com.sun.sgs.impl.auth.NamePasswordAuthenticator.PasswordFile=/path/to/password/file
```

To create a custom authenticator, you need to create a class that implements the `com.sun.sgs.auth.IdentityAuthenticator` interface. This interface requires the following three methods:

`public String[] getSupportedCredentialTypes()`

Returns the types of credentials supported by this authenticator. Currently should only return `{"NameAndPasswordCredentials"}`.

`public void assignContext(com.sun.sgs.kernel.KernelAppContext context)`

Gets the server-specific application context object, currently unused.

`public com.sun.sgs.auth.Identity authenticateIdentity (com.sun.sgs.auth.IdentityCredentials)`

Returns an Identity object if authentication is successful. Throws a `CredentialException` if authentication fails.

When creating a custom authenticator, the `authenticatedIdentity` method takes an `IdentityCredentials` object. This is a super class for a set of different credential implementations. Currently, there is only a single implementation, the `NamePasswordCredentials`. `NamePasswordCredentials` provides two methods:

`public String getName()`

Gets the user name used in these credentials

`public char[] getPassword()`

Gets the password used in these credentials

Once authentication is complete, the custom authenticator must return a custom implementation of the `com.sun.sgs.auth.Identity` interface. This interface requires the implementation of three methods:

`public String getName()`

Gets the user name of this identity.

`public void notifyLoggedIn()`

Event handler, called when the user with the same user name logs in.

`public void notifyLoggedOut()`

Event handler, called when the user with the same user name logs out.

Putting this all together, the following is an implementation of a trivial custom authenticator that requires that the username and password be the same. (I never said it was a high-security authenticator!)

```
import com.sun.sgs.auth.*;
import com.sun.sgs.impl.auth.*;
import com.sun.sgs.kernel.KernelAppContext;

public class MyCustomAuthenticator implements com.sgs.auth.IdentityAuthenticator
{
   protected KernelAppContext context;

   public static class MyIdentity implements Identity {
       protected String name;

       public MyIdentity(String name) {
          this.name = name;
       }

       public String getName() { return name; }

       public void notifyLoggedIn() { System.out.println(name+" logged in."); }
```

```
        public void notifyLoggedOut() { System.out.println(name+" logged out."); }
    }

    public String[] getSupportedCredentialTypes() { return new String[] {"NameAndPasswordCredential

    public void assignContext(KernelAppContext c) { this.context = c; }

    public Identity authenticateIdentity(IdentityCredentials credentials)
         throws LoginException
    {
        if (!c instanceof NamePasswordCredentials)
           throw new CredentialException("Unsupported credentials: "+credentials.getType());
        NamePasswordCredentials npc = (NamePasswordCredentials)credentials;
        if (npc.getName().equals(new String(npc.getPassword())))
           return new MyIdentity(npc.getName());
        else
           throw new CredentialException("Name and password must match!");
    }
}
```

To use your custom authenticator, you need to set the `com.sun.sgs.app.authenticators` property to equal the name of your class in your *server.properties* file, e.g., add the following line to your *server.properties* file.

```
# Custom Authenticator
com.sun.sgs.app.authenticators=MyCustomAuthenticator
```

Currently, Darkstar only has name and password authentication. Other more sophisticated methods of authentication, like biometrics, are currently not possible.

Sending Objects

You may have noticed that all data between client and server is sent and received using arrays of bytes. Darkstar is designed this way to facilitate interactions between Darkstar servers written in Java and clients written in another language such as C++. However, if both your client and your server are written in Java, its much nicer to be able to send and receive actual objects. For this to work, all of the objects you send between client and server must implement the `java.io.Serializable` interface. This interface is simply a marker that says your class can be converted (or serialized) to a raw byte-based representation. To be serializable, an object must only contain primitive types (e.g., Boolean, int, double, etc.) or objects which are themselves serializable.

As an example, consider this serializable message for a space game that indicates a rocket was fired:

```
public class FireRocketMessage implements java.io.Serializable {
   protected double force;
```

```
   public FireRocketMessage(double force) {
      this.force = force;
   }
   ...
}
```

When you de-serialize an object, converting it from a byte array to an object, it is returned as a generic `Object` class. You must cast it into the appropriate class. To accomplish this, you can use the *instanceof* test to determine its concrete type and cast it accordingly. This function does this:

```
public void handleMessage(Object msg) {
    if (msg instanceof FireRocketMessage) {
        handleRocketMessage((FireRocketMessage)msg);
    }
    else if (msg instanceof NextTypeOfMessage) {
        ...
    }
    ...
    else {
       System.err.println("Unknown message: "+o);
    }
}
```

To actually convert an object into bytes, you need to use an `ObjectOutputStream` and a `ByteArrayOutputStream`. The `ObjectOutputStream` knows how to write objects and the `ByteArrayOutputStream` is an `OutputStream` object that writes to a byte array. The following is a send method that you could add to either `ServerSession` or `ClientSession` to send objects instead of bytes:

```
import java.io.*;
...
public void send(Object msg)
   throws IOException
{
   ByteArrayOutputStream bytes = new ByteArrayOutputStream();
   ObjectOutputStream oos = new ObjectOutputStream(bytes);
   oos.write(msg);
   oos.flush();
   send(bytes.toByteArray());
}
```

Similarly, you can write a `receivedMessage(byte[] msg)` method that converts the byte array into an object and calls a `receivedMessage(Object msg)` method. This method can be added to both your `ClientSessionListener` and your `ServerSessionListener` implementations.

```
import java.io.*;
...
public void receivedMessage(byte[] msg) {
```

```
    try {
      ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(msg));
      Object m = ois.readObject();
      receivedMessage(m);
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    catch (ClassNotFoundException ex) {
      ex.printStackTrace();
    }
}
```

By adding these methods to your client and server side classes, you can easily transmit objects back and forth without having to worry about the specific details of converting them to and from raw arrays of bytes.

Managing Game State with Persistent Objects

Communication is necessary and important, but without game state there is no game. You can talk until your blue in the face, but its very difficult to play a game of chess without a board and pieces to refer to. *Game state* contains all of the data necessary to represent current state of the entire game world. In chess it is the location of positions on the board and who's move it is. A massively multiplayer online role playing game (MMORPG) has a more complex game state. It contains your character, the possessions you were carrying, the nearby dragon that you were battling, the cost of beer at the local inn, and much more. In an online chess game it would contain the state of the chess board and whose move it is.

To simplify the lives of developers, in Darkstar all of this state is represented by traditional Java objects. You can use object-oriented design to construct objects that represent your game state. If you need to represent a sword in a MMORPG, you just declare a `Sword` class. Better yet, that `Sword` class can extend from a `Weapon` parent class, which could, in turn, extend from an `Item` class. Designing the state of your game is just like designing the objects for any other program you might write.

In the traditional server-client model, the entire state of the game world is stored on the server. This ensures a single authoritative copy of the data, but it potentially requires simultaneous (or *concurrent*) access and modification of the data by multiple clients. Managing concurrent data access is one of the most difficult challenges in software development. Failure to use locks to protect data from simultaneous modification can lead to race conditions and other bugs. However, incorrect use of locks leads to significant performance penalties or even worse: deadlock. Fortunately, Darkstar provides a services that make implementing cor-

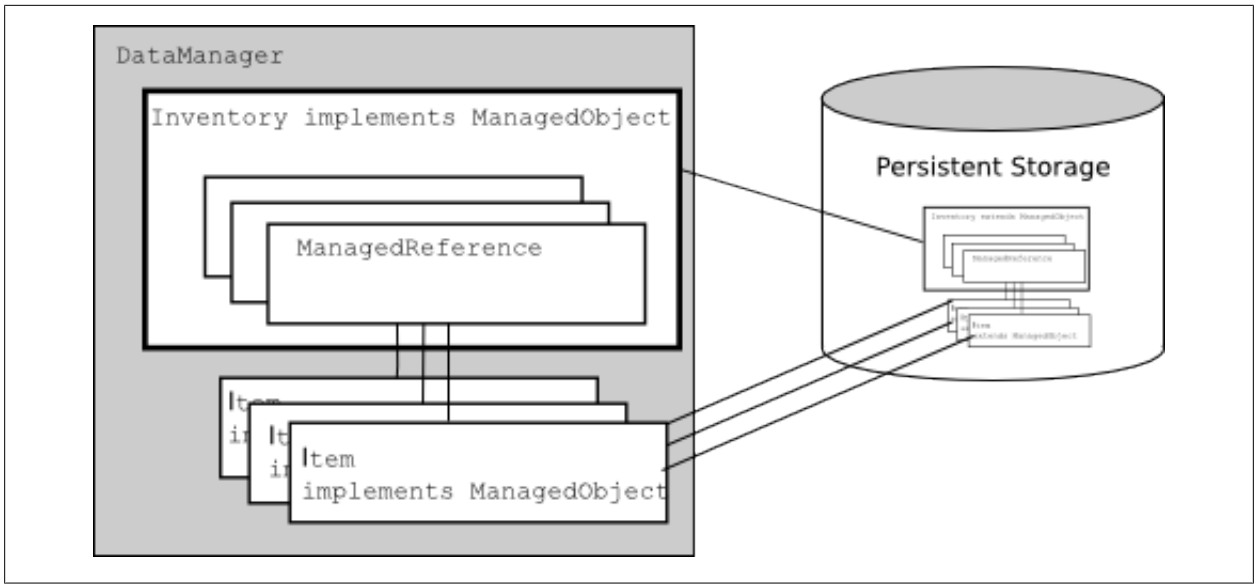

Figure 4. Objects for creation and maintenance of game state in the Darkstar application server

rect concurrent access to game state significantly easier and less error prone. In addition to the challenge of concurrency, maintaining the entire state of the game on a single server creates a single point of failure. If the server crashes, all your game data might be lost. To eliminate this possibility, Darkstar *persistently* stores the game state to the filesystem so that when the server is restarted after a crash, the game starts up exactly where it left off.

The objects and relationships for representing game state in Darkstar are shown in **Figure 4** and discussed in detail in the following sections.

The DataManager and ManagedObjects

In Darkstar, services for handling concurrency are provided by the `DataManager` interface. All of the game state objects in your game are registered with the `DataManager` when they are created. All subsequent access to the objects is managed by the `DataManager`. This ensures that concurrent access is handled safely and efficiently. All objects that are managed by the `DataManager` must implement the `ManagedObject` interface. `ManagedObject` is a *marker* interface that is used to indicate that the object is managed by the `DataManager` it doesn't require the implementation of any methods. All `ManagedObjects` must also implement the `java.io.Serializable` so that they can be persistently stored. Darkstar automatically persists all game state by writing it to disk. This ensures that even if your server crashes, the game state will survive. When you restart the game server, or fail over to a different server, the state is seamlessly maintained.

`java.io.Serializable` is also a marker interface. For more on the details of writing a serializable object see **Appendix A, *Serialization***. The following is an example of an `Item` class from a role-playing game. An Item is a `ManagedObject` that serves as a super class for all items in the game.

```
import com.sun.sgs.app.ManagedObject;
import java.io.Serializable;

public class Item implements ManagedObject, Serializable
{
   static final long serialVersionUID = 0L;
   protected double price;

   public Item(double price) {
      this.price = price;
   }

   public double getPrice() { return price; }
}
```

`ManagedObject`s are registered with the `DataManager` by *binding* them to a specific, and presumably meaningful, name. For example, you might bind an object representing a player to her username. Or an AI player to a name like Elf-10. In this way, the `DataManager` is basically a sophisticated dictionary, mapping `String` names to specific `ManagedObject`s. This enables all the different parts of your game to obtain references to the same set of objects. The `DataManager` provides the following methods for adding, querying and removing `ManagedObject`s:

`public void setBinding(String name, ManagedObject object)`

This binds a specific instance of a `ManagedObject` to the specified name. If another object was already bound to the name, it is replaced.

`public void removeBinding(String name)`

Removes a binding of a name to a `ManagedObject`. It will throw `NameNotFoundException` if the name is not bound to an object.

> **Note**
>
> Does not remove the object itself, this must be done through `DataManager.removeObject(...)`.

`public void removeObject(ManagedObject o)`

Removes an object from storage in the `DataManager`. It will throw an `ObjectNotFoundException` if the object is not currently stored in the `DataManager`.

> **Note**
>
> Does not remove the binding between the name and the object, this must be done through `DataManager.removeBinding(...)`.

`public <T> T getBinding(String name, Class<T> type)`

Obtains the `ManagedObject` of the specified type, bound to the specified name. It throws a `NameNotBoundException` if no object is bound to that name. It throws an `ObjectNotFoundException` if the name is bound to an object, but that object can't be found. This can occur if you have called `DataManager.removeObject(...)`, but have not called `DataManager.removeBinding(...)`.

> **Note**
>
> Objects stored in the `DataManager` are never garbage collected. They persist forever until they are explicitly removed from the `DataManager`.

The lifecycle of a `ManagedObject` is as follows:

- A new object implementing the `ManagedObject` interface is constructed.
- This `ManagedObject` is bound to a name using DataManager.
- The `ManagedObject` is used for some period of time.
- The `ManagedObject` is deleted by calling `removeBinding(...)` and `removeObject(...)`.

The `DataManager` also provides a method for iterating through all bound names:

`public String nextBoundName(String name)`

This method returns the next name bound to a `ManagedObject` in order after the specified name. If there are no more bound names, the method returns null. If the specified name is null, the `DataManager` returns the first bound name.

The signature for `getBinding(...)` is somewhat unusual. It makes use of generic typing which was introduced in Java 5.0. The method takes as an argument an instance of a `Class` object, which contains the type of the `ManagedObject` that is referenced by the `ManagedReference`. For example, it might be `Class<String>` if `String` were a `ManagedObject`. This enables the `get` and `getForUpdate` methods to return an object of the appropriate type, rather than simply returning an object of type `Object` and forcing the user to cast it into its actual type. While this is convenient, it forces the programmer to obtain the correct `Class` object. There are three methods for obtaining a `Class` object.

The easiest is to refer to the static "class" member variable:

```
Class<Item> item_class = Item.class;
```

If you have an instance of that object, you can call its `getClass()` method. For example:

```
Item foo;
...
Class<Item> item_class = foo.getClass();
```

Alternatively, you can use the `static Class.forName(String name)` method:

```
Class<Item> item_class;
try {
   item_class = (Class<Item>)Class.forName("Item");
} catch (ClassNotFoundException ex) {
   ex.printStackTrace();
}
```

Because either of these last methods is computationally expensive, and the reference to the `Class` can be used throughout the lifespan of your objects, if you need to use them, it makes sense to have the `Class` be an instance variable in your `ManagedObject`, which is obtained once in the constructor, and used in all of the other methods.

As an example of this using the `DataManager`, imagine we have a `Forge` class that is used to create swords, meltdown swords, but can also meltdown all the swords in the world:

```
import com.sun.sgs.app.*;

public class Forge {
   private static long swords_id = 0;

   public static Sword createSword() {
       Sword s = new Sword(swords_id++);
       DataManager dm = AppContext.getDataManager();
       dm.setBinding("Sword_"+swords_id, s);
       return s;
   }

   public static void meltdownSword(Sword s) {
       DataManager dm = AppContext.getDataManager();
       dm.removeBinding("Sword_"+s.getId());
       dm.removeObject(s);
   }

   public static void meltdownAllSwords() {
       DataManaged dm = AppContext.getDataManager();
       String name = dm.nextBoundName(null);
```

```
        while (name != null) {
            if (name.startsWith("Sword_")) {
                Sword s = dm.getBinding(name, Sword.class);
                meltdownSword(s);
            }
        }
    }
}
```

ManagedReferences

Often times in writing your game, a `ManagedObject` must refer to another `ManagedObject`. For example, in the MMORPG example described earlier, the `ManagedObject` representing the inventory of the store, contains a reference to the `ManagedObject` representing the sword being purchased. When these sort of references occur, they must be stored in the `ManagedObject` as a reference to a `ManagedReference` rather than a direct reference to the `ManagedObject` in question. This enables Darkstar to understand that the object should not be persistently stored as a part of the inventory class, but rather persistently stored on its own. The `DataManager` class provides two methods related to managed references:

`public ManagedReference createReference(ManagedObject object)`

> Creates a `ManagedReference` that refers to the given instance of a `ManagedObject`.

`public void markForUpdate(ManagedObject object)`

> Notifies Darkstar that you will be modifying the given `ManagedObject`. This is equivalent to calling `DataManager.createReference().getForUpdate(...)`, except that it does not return the `ManagedObject` since the callee already has a reference to it. This method allows users to upgrade a reader lock on a `ManagedObject` to a writer lock.

> **Note**
>
> If you want to have a `ManagedObject` as an instance variable of another `ManagedObject`, you *must* instead store a `ManagedReference` to that `ManagedObject`. If you do not do this, Darkstar will make a clone of the `ManagedObject`, to serve as the instance variable. In most cases this will cause your code to malfunction because it expects that the two objects are the same.

The `ManagedReference` interface provides three methods:

`public <T> T get(Class<T> type)`

Returns the `ManagedObject` that this `ManagedReference` refers to. `Class<T>` is an instance of a `Class` object that refers to the type of referenced `ManagedObject`.

> **Note**
>
> Obtaining a `ManagedObject` using this method indicates to the `DataManager` that this reference is being used for reading only, the returned instance should not be modified. See **The Challenge of Concurrency** for further details on concurrency control.

`public <T> T getForUpdate(Class<T> type)`

Returns the `ManagedObject` which this `ManagedReference` refers to. `Class<T>` is an instance of a `Class` object, which refers to the type of referenced `ManagedObject`.

> **Note**
>
> Obtaining a `ManagedObject` using this method indicates to the `DataManager` that this reference is being used for reading and writing. And prevents others from simultaneously accessing this object. See the following section for further details.

`public BigInteger getId()`

Gets the identifier for the referenced `ManagedObject`. Two `ManagedReference` objects, `a` and `b`, refer to the same `ManagedObject` if and only if `a.getId() == b.getId()`.

The following is an example of implementing the inventory of a store in a role-playing game using `ManagedReferences`:

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.Vector;

public class Inventory implements ManagedObject, Serializable {
  static final long serialVersionUID = 0L;
  protected Vector<ManagedReference> items;

  public Inventory() {
     items = new Vector<ManagedReference>();
  }

  public void addItem(Item i) {
```

```
    DataManager dm = AppContext.getDataManager();
    ManagedReference ref = dm.createReference(i);
    items.add(ref);
  }

  public Item getItem(int ix) {
    ManagedReference ref = items.get(ix);
    return ref.get(Item.class);
  }

  public boolean has(Item i) {
    for (int i=0;i<items.size();i++) {
      if (items.get(i).get().equals(i))
        return true;
    }
    return false;
  }
}
```

The Challenge of Concurrency

In any multiplayer game, there are many different clients simultaneously accessing and potentially modifying the game state. For example, in a MMORPG, two players may be in the same store looking at the inventory of the store. Imagine what might happen if both players attempt to buy the same sword at the *exact* same time. Imagine that the algorithm for buying a sword looks like this:

1. Check that sword is still available from the shop
2. Remove sword from store inventory
3. Add sword to character possessions
4. Pay shopkeeper

The table shows a perfectly reasonable algorithm for purchasing a sword, but think about what happens when two (or more) players run it at the same time.

Time	Player 1	Player 2
1	Checks if the sword is available	Checks if the sword is available
2	Removes sword from store inventory	Removes sword from store inventory
3	Adds sword to possessions	Adds sword to possessions
4	Pays shopkeeper	Pays shopkeeper

In time-step one, both players check if the sword is in the store's inventory. Since it hasn't been removed yet, this check returns true. Then both players remove the sword from the store's inventory. This is a bug, since it is impossible for the same

sword to be removed from the store's inventory twice. But even worse, in time-step three, both players add the sword to their possessions. So now two players posses the *same* sword. Finally, the shopkeeper happily accepts two payments for the same sword. Clearly this violates all of the rules of physics and economics! This is an example of a *race* condition, and it is one of the most insidious bugs in all of programming.

One possible solution to this problem is to put all the players in a single-file line. If any one player is shopping, no other players can buy anything. While this approach is safe, it is extremely inefficient. It forces every player to wait for everyone in front of them, even if they aren't buying the same item. Further, a player who is simply reading the list of available items, prevents anyone else from purchasing something. We need to provide safe concurrent access to different game objects, but we also need to allow multiple players to simultaneously examine a game object so long as no player is modifying that object. Fortunately, Darkstar provides services for implementing a safe and efficient solution.

The paradigm that Darkstar uses is to solve problems of safe concurrent access is reader/writer locks. You may have seen reader/writer locks in the context of operating systems or other concurrent programming. The semantics of reader/writer locks are straightforward. Any simultaneous thread of execution can acquire a lock on an object as either a *reader* or a *writer*. A thread which acquires a lock as a reader can call object methods which do not modify the object's state. For example calling a `get` method which reads an instance variable. A thread which acquires a lock as a writer can call any method on the object. As many threads as want to can obtain locks as a reader so long as no thread has acquired the lock as a writer. Only one thread can obtain a lock as a writer.

> **Note**
>
> These rules are *not* enforced by Darkstar or Java. It is up to the programmer to know which type of lock they have obtained and behave appropriately.

In Darkstar, each `ManagedObject` which is contained by the `DataManager` has a separate reader/writer lock attached to it. When a user makes a call to `DataManager.getBinding(...)`, he obtains a reader lock on that object. A reader lock may also be obtained from a `ManagedReference` using the `get(...)` method. Writer locks can be obtained using the `getForUpdate(...)` method of the `ManagedReference` class. A reader lock may be upgraded to a writer lock using `DataManager.markForUpdate(...)`.

Note

People who have worked with locks before will note that there is no explicit method in Darkstar "unlock" or give up a lock once it has been obtained. Because Darkstar is an event driven server, locks are always obtained in the context of handling some event (for example, the server calls your `recevied Message(...)` method in a `ClientListener`. Once a lock is obtained, it is kept for the duration of the event handling method. When the method ends, the lock is released. Locks in Darkstar are also reentrant, thus you don't need to worry about making multiple calls to `get(...)` or `getForUpdate(...)` in the process of handling an event.

As a concrete example of concurrency control, this is an implementation of a `Player` class that uses concurrency control to safely purchase items from the store:

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.Vector;

public class Shopper implements ManagedObject, Serializable {
    protected int money;
    protected Vector<ManagedReference> stuff;

    public Shopper() {
      this.money = 500;
      this.stuff = new Vector<ManagedReference>();
    }

    public int getMoney() {
      return money;
    }

    public int getItemCount() {
      return stuff.size();
    }

    public Item getItem(int ix) {
      return stuff.get(ix).get(Item.class);
    }

    public boolean buy(Item item, Inventory inv) {
      DataManager dm = AppContext.getDataManager();
      int price = item.getPrice();
      dm.markForUpdate(this);
      if (price < money) {
        dm.markForUpdate(inv);
```

```
        if (inv.has(item)) {
           stuff.add(dm.createReference(item));
           inv.removeItem(item);
           money -= price;
           return true;
        }
      }
      return false;
    }
}
```

Generating Action with Tasks

So far everything in our game server has been reactive. A client sends a message, the server handles that message with an appropriate response. This works fine for certain games like card games or board games where user action is the only thing that changes the state of the game. However, in many types of games, the game itself modifies the game state as time passes, even in the absence of user input. For example, in a space simulation, the spaceships, asteroids and aliens all have velocities. As time passes, all of these things move, regardless of whether the clients have sent commands or not. Furthermore, in these action oriented games, the state of the game world is sent back to the client periodically (usually 30 times per second or so), so that the client can display the evolving game state. These sorts of server-initiated actions are called "tasks" in Darkstar. The application server provides support for easily creating either one-time or periodic tasks. Tasks are used in Darkstar instead of user-created threads. Letting the server manage concurrency allows the server to optimize scheduling and possibly distribute load onto different CPUs or servers.

Creating and Running Tasks

The `Task` interface is quite similar to `java.lang.Runnable`. They both require a user to implement a single `run()` method which takes no parameters. However, there are several important differences between `Task` objects and `Runnable` objects. First, `Task`'s run method can throw any `Exception`, this exception can indicate that `Task` execution has temporarily failed, but it should be retried at a later time. The second, much more significant, difference between `Runnables` and `Tasks`, is that the expected lifespan of a task is quite short. If the execution of a `Task` lasts longer than a preset time threshold, the execution of that `Task` is terminated. This feature of Darkstar provides resilience. It helps the server to prevent a malicous client or a software bug from monopolizing server resources at the expense of other players. The time threshold is maintained in the `com.sun.sgs.txn.timeout` property. This property defaults to 60,000 milliseconds (1 minute). It is important to note that

the timeout is only enforced when you make a call to a Darkstar function which may throw `TransactionException` (such as `ClientSession.send(...)` or `AppContext.getDataManager(...)`). If you put your game into a tight loop (e.g., `while (true) ;`) your `Task` will not be terminated by Darkstar.

`Task` objects must also implement the `Serializable` interface. Darkstar will persist Tasks to disk and restart them if your game server ever crashes and needs to be restarted. Consequently, any member variable in a `Task` that is a `ManagedObject` must be through a `ManagedReference` object, rather than a direct `ManagedObject`. Despite being persisted by Darkstar, `Tasks` are not required to implement the `ManagedObject` interface. Darkstar will manage the the `Task` and remove it from persistent storage when it has completed execution. However, if the `Task` does implement `ManagedObject`, then Darkstar assumes that it is being actively managed by your game and your game must actively remove the `Task` from the `DataManager`.

As a simple example, here is a `Task` which simply sends a message to a client without the client needing to send a message to the server.

```
import com.sun.sgs.app.*;

public class MessageTask implements Task, java.io.Serializable {
   private ClientSession client;
   public MessageTask(ClientSession client) {
     this.client = client;
   }

   public void run() {
     client.send("Hello from the server".getBytes());
   }
}
```

Another difference between the `Runnable` interface and `Tasks` is how a `Task` is queued for execution. Rather than creating a new `Thread` object, each `Task` is queued for execution with a centralized `TaskManager`. The `TaskManager` is a core component of the Darkstar application server. Because it knows about all `Tasks` that are simultaneously active in the server, the `TaskManager` can make intelligent choices about `Task` scheduling that improve efficiency. The `TaskManager` can be obtained by calling `AppContext.getTaskManager()`. It provides two methods for starting a `Task`:

`public void scheduleTask(Task t)`

Schedules a task for immediate execution.

`public void scheduleTask(Task t, long delay)`

Schedules a task for execution after a specified delay in milliseconds.

It is important to note, that like the `start()` method in `java.lang.Thread`, these methods are *asynchronous*. They immediately return after the `Task` has been scheduled for execution, but possibly before the `Task` has actually been run. Once scheduled, the `Task` is run by a separate component in the Darkstar server. Here is an example of scheduling a `Task`:

```
ClientSession client;
...
MessageTask mt = new MessageTask(client);
// Schedule a task for execution in 1 second.
AppContext.getTaskManager().scheduleTask(mt, 1000);
...
```

Periodic Tasks

The most common `Tasks` in games are periodic tasks that execute at a specific period, or time-interval and perform some sort of maintenance or updating on the server. Examples of these types of behaviors include updating the state of objects obeying simulated physics and broadcasting information back to a client. To support these sorts of activities, you can also schedule `Tasks` for repeated execution at a given period:

`public PeriodicTaskHandle schedulePeriodicTask(Task t, long delay, long period)`

Schedules a `Task` for repeated execution at a given interval (in milliseconds). The first execution of the `Task` occurs after a (possibly zero) delay (also in milliseconds). Returns a `PeriodicTaskHandle` object that has a single cancel() method which is used to stop subsequent periodic executions of the Task.

There are two important caveats for the `TaskManager`'s execution of periodic threads. First, scheduling a `Task` for periodic execution does not guarantee that a `Task` will be run. Darkstar makes every possible effort to ensure that the periodic task is run at the specified interval, but if the server is overloaded tasks may not be run. If this happens, subsequent attempts to run the task at the next periodic interval will still occur. Second, Darkstar does not guarantee that a previous call to a periodic task's `run()` method has finished before it begins executing a second call to the `run()` method. If your `Task`'s `run()` method takes longer to execute than the periodic interval, then executions of your task will overlap.

For example of a periodic task, here is a `Task` that simulates a bouncing ball subject to gravity.

```
import com.sun.sgs.app.*;

public class GravityTask implements Task, java.io.Serializable {
  Ball b;
  long timeDelta;

  public GravityTask(Ball b, long timeDelta) {
    this.b = b;
    this.timeDelta = timeDelta;
  }

  public void run() {
    //  Position += velocity*time + 1/2 acceleration * time squared.
    b.incrementPosition(b.getVelocity()*timeDelta+0.5*9.8*timeDelta*timeDelta);
    //  new Velocity += acceleration * time
    b.incrementVelocity(9.8*timeDelta);
  }
}
```

Here is an example of scheduling this `Task`:

```
Ball b;
...
long period = 100;
GravityTask gt = new GravityTask(b, period);
TaskManager tm = AppContext.getTaskManager();
PeriodicTaskHandle handle = tm.schedulePeriodicTask(gt, 0, period);
```

Tasks and Server Performance

Creating `Tasks` of appropriate granularity is an important consideration for performance tuning a game in Darkstar. In general, for maximum performance, your `Tasks` should be as short and modular as possible. Because contention for shared resources significantly limits performance, `Tasks` should access as few `ManagedObjects` as possible. As an example, consider the task of animating a collection of "moveable" objects in the game. Suppose the Moveable interface provides a single method: `move()`, which updates a `Moveable` object's position. One possible way to implement moving all objects is to have a single `Task` that iterates over a collection of moveable objects, as in the following code:

```
public class MoveableTask implements Task, java.io.Serializable {
  protected Vector<ManagedReference> moveables;
  ...
  public void run() {
    for (int i=0;i<moveables.size();i++) {
      Moveable m = moveables.get(i).getForUpdate(Moveable.class);
      m.move();
    }
  }
}
```

This code is a natural way to implement moving all objects in a game, however, it doesn't scale very well. There could be hundreds of thousands, or even millions of objects in an MMORPG, think about trying to apply the laws of physics move them all with a single task. Clearly, that task couldn't finish within the threshold for task execution. Also, this `Task` grabs writer access to all of the objects in the world, preventing any other access to the game state until *all* objects have finished moving. Most importantly, because there is only a single task involved in moving objects, it can't be distributed across multiple nodes in a multiprocessor or cluster system. Thus, this single task approach can't take full advantage of the server's resources.

An alternate approach to this 1-to-n architecture is a 1-to-1 approach that creates a different `Task` object for every single object in the world. In this approach, each `Task` is only responsible for moving one object. For example:

```
public class MoveableTask implements Task, java.io.Serializable {
  protected ManagedReference moveable;
  ...
  public void run() {
    Moveable m = moveable.getForUpdate(Moveable.class);
    m.move();
  }
}
```

Clearly, this `Task` can run quite quickly. More importantly, it only obtains writer access for a single object, and immediately releases it when the task completes. Thus access to other objects are permitted even while the `Task` is running. Finally, because many different `Tasks` are responsible for moving different objects, they can be distributed onto multiple processors or even multiple computers to balance load as the number of objects in the game state increases.

There are two potential downsides to the 1-to-1 approach. First, you are effectively doubling the memory size of every moveable object in your game state. You need a certain amount of memory to store the object itself, and a similar amount of memory to store the `Task` responsible for moving that object. Of course, memory is cheap, so this is a relatively small concern. More importantly, you are eliminating a central, synchronized clock from your game. For certain games this is no problem at all. However, games that rely on time-dependent information to make decisions (such as collision detection between two ships in an arcade game) this has a significant impact. The consequences of an unsynchronized clock and algorithms for overcoming it will be discussed in greater detail in the example application at the end of this book.

Task Exceptions and Retry

When your `Task`'s run method is called, it is always possible that an error may occur. Many of these errors are fatal and your `Task` should terminate and never be run again, but sometimes these errors are temporary and you want to make a subsequent attempt to run the `Task` successfully. For example, your `Task` might be attempting to connect to a network server which is temporarily unreachable (due to a temporary network failure, for example). In this case, you want to signal to Darkstar that your `Task` has temporarily failed, but that you would like the application server to make subsequent attempts to re-run your `Task` at a later time. To accomplish this, your `Task` can throw an `Exception` that implements the `Exception RetryStatus` interface. If your `Task` throws such an exception, and that exception's `shouldRetry()` method returns true, then Darkstar will attempt to reschedule the task at a later time. Darkstar also reserves the right to reschedule any `Task` that throws an exception for later execution. In particular, if your `Task` throws an exception just before a server is terminated, then it may be run again when the server is restarted. Consequently, you can not use throwing an exception to terminate the execution of a `Task`.

Efficient Client-Client Communication Through Channels

In some cases you want to send information from one client to another client. For example, you might provide chat communications between clients on the same team, or all players in the same room in a MMORPG. Using the communication methods previously discussed, all of these communications would have to first be sent into the server and then broadcast back out to the appropriate other clients. This approach makes the server a single bottleneck in communication. In general, your game server has much better things to do with its CPU resources than relay communications from one client to another. A more efficient approach is to have clients communicate directly with each other. In Darkstar, this direct communication is provided by channels. Channels provide a publish/subscribe paradigm for communication. Clients are subscribed to one or more named channels. A client can then send arbitrary messages to the channel. Any message sent by a client to a particular channel is received by all of the other subscribers to the channel. The sender does *not* receive a copy of his own message. A comparison between Darkstar's two different communication methods are shown in **Figure 5**.

In Darkstar, channels are created and managed by the `ChannelManager`. Like the other manager objects, the `ChannelManager` can be obtained using the static method `AppContext.getChannelManager()`.

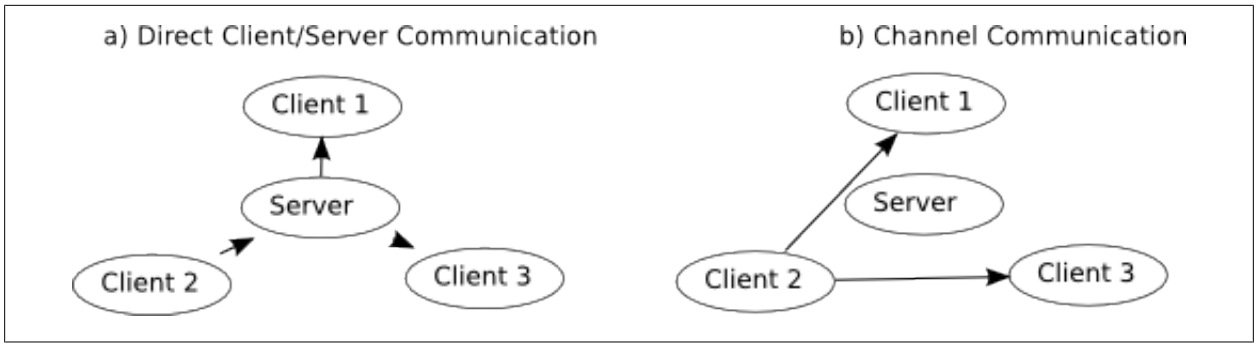

Figure 5. Comparing Client 2 sending a message to Clients 1 and 3 using direction communication through the server or channel communication

Creating Channels

In Darkstar, communication channels can only be created on the server. To allow clients to create channels, the server must be programmed to respond to a developer designed "channel create" message which makes the server create the channel on behalf of the client. The `ChannelManager` provides two methods for creating and managing channels:

`public Channel createChannel(String name, ChannelListener listener, Delivery delivery)`

> Creates a channel with the specified name. If listener is non-null, attach the specified server-side channel listener to the channel. The delivery object specifies the delivery requirements for the channel. The method will throw `NameExistsException` if a channel with the given name already exists.

`public Channel getChannel(String name)`

> Obtains a pre-existing channel by its name. Throws `NameNotBoundException` if there is no channel bound to the specified name.

When creating a `Channel`, the server provides two parameters used to define the `Channel`. The first is simply a `String` that gives the `Channel` a name. The second is a `Delivery` object, which specifies the message delivery requirements for the `Channel`. In Darkstar there are three different `Delivery` requirements that are possible for any given `Channel`, they are specified by three static singleton `Delivery` objects:

`Delivery.RELIABLE`

> Message delivery is guaranteed. Messages are delivered in the order in which they are sent.

`Delivery.ORDERED_UNRELIABLE`

Message delivery is not guaranteed. However, messages that do arrive will arrive in order.

`Delivery.UNRELIABLE`

Neither message delivery nor message order is guaranteed.

Regardless of the delivery requirements, messages are guaranteed to be delivered at most once (i.e., there is no possibility of duplicate messages). The choice of delivery requirements has an impact on the performance of your game's channel communication. You should always choose the least stringent delivery requirements necessary for your game to perform correctly. However, it is never a good idea to choose a less stringent requirement and then work around it in your code (for example by using unreliable delivery and then implementing your own version of acknowledgements and retransmission.) It is always more efficient to rely on Darkstar's communication methods to implement the delivery requirements necessary for your game.

When a `Channel` is created, the server also has the option of providing an object that implements the `ChannelListener` interface. This is an event handling interface which receives messages sent to the `Channel`.

> **Note**
>
> Any custom implementation of `ChannelListener` passed to `createChannel(...)` must also implement the `java.io.Serializable` interface so that it can be persisted to file storage.

Here is a simple implementation of the `ChannelListener` interface:

```
import com.sun.sgs.app.*;
import java.io.Serializable;

public class MyChannelListener
   implements ChannelListener, Serializable
{
   public void receivedMessage(Channel channel, ClientSession client, byte[] msg) {
       System.out.println("Received: "+new String(msg)+" from "+client+" on "+channel);
   }
}
```

Here is an example of a server that instantiates two Channels at start-up: a reliable channel that it listens to and an unreliable channel that it doesn't:

```
import com.sun.sgs.app.*;
import java.util.Properties;
```

```
import java.io.Serializable;

public class ChannelDemoMain
    implements Serializable, AppListener
{
    String  channelNames = new String[] {"Team Rockstar", "King Cobras"};
    Channel[] channels;

    public void initialize(Properties init) {
        channels = new Channel[channelNames.length];

        ChannelManager cm = AppContext.getChannelManager();
        channel[0] =
            cm.createChannel(channelNames[0], new MyChannelListener(), Delivery.RELIABLE);
        channel[1] =
            cm.createChannel(channelNames[1], null, Delivery.UNRELIABLE);
    }

    public ClientSessionListener loggedIn(ClientSession session)
    {
        return new ChannelClientListener(session, channels);
    }
}
```

Managing Channel Participants

The server has sole control over when clients join or leave a channel. To support users joining/leaving channels in your game, you must implement special channel join or leave messages which the client sends to the server in order to join or leave a channel. The `Channel` object returned by `ChannelManager.createChannel(...)` or `ChannelManager.getChannel(...)` has the following methods for managing the members of a channel:

`public void join(ClientSession client, ChannelListener listener)`

Adds a particular client to a channel. If listener is non-null then its `receivedMessage(...)` method will be called whenever this client sends a message to the channel. The listener will not be called if the server or other clients send messages to the Channel.

`public void leave(ClientSession client)`

Removes a particular client from a channel.

`public void leaveAll()`

Removes all current listeners from a particular channel.

`public Set<ClientSession> getSessions()`

Get session objects for all of the clients that are currently members of the channel.

`public boolean hasSessions()`

Returns true if there is at least one client subscribed to the channel, returns false otherwise.

Here is a simple implementation of a server which handles creating, joining and leaving a channel:

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.*;

public class ChannelClientListener
    implements Serializable, ClientSessionListener
{
    protected ClientSession session;
    protected Vector<Channel> client_channels;

    public ChannelClientListener(ClientSession session)
    {
         this.session = session;
         this.my_channels = Vector<Channel>();
    }

    public Channel getChannel(String name) {
         try {
            return AppContext.getChannelManager().getChannel(name);
         } catch (NameNotBoundException ex) {
            return null;
         }
    }

    public void receivedMessage(byte[] message)
    {
        String msg = new String(message);
        if (msg.startsWith("join")) {
            // format is "join <channel-name>"
            String channelName = msg.substring(5);
            Channel c = getChannel(channelName);
            if (c != null) {
                // Join this user
                c.join(session, null);
                // Confirm join.
                session.send("OK".getBytes());
                my_channels.add(c);
            }
```

```
            else {
                session.send("No such channel".getBytes());
            }
        }
        else if (msg.startsWith("leave")) {
            // format is "leave <channel-name>"
            String channelName = msg.substring(6);
            Channel c = getChannel(channelName);
            if (c != null) {
                // Leave the channel
                c.leave(session);
                // Confirm leave
                session.send("OK".getBytes());
                my_channels.remove(c);
            }
            else {
                session.send("No such channel".getBytes());
            }
        }
        else if (msg.startsWith("create")) {
            // format is "create <channel-name>"
            String channelName = msg.substring(7);
            try {
                ChannelManager cm =
                    AppContext.getChannelManager();
                cm.createChannel(chanelName, null, Delivery.RELIABLE);
                session.send("OK".getBytes());
            } catch (NameExistsException ex) {
                session.send("Channel exists.".getBytes());
            }
        }
        else {
            session.send("Unknown message".getBytes());
        }
    }

    public void disconnected(boolean graceful) {
        Stirng disconnect;
        if (graceful)
            disconnect = "Graceful disconnect.";
        else
            disconnect = "Hard disconnect";
        System.out.println(session+" disconnected: "+disconnect);
    }
}
```

Because the client can only be added to or removed from channels by the server, the client side of channel participation consists of a series of event notifications.

The `ServerSessionListener` interface requires an event handling method which receives notifications when the client is added to a channel:

`public ClientChannelListener joinedChannel(ClientChannel channel)`

This method is called when the client is joined to a channel specified by the `ClientChannel` parameter.

Here is an implementation of the `ServerListener`, which adds support for `Channels`, it extends the `MySimpleClientListener` class.

```
import com.sun.sgs.client.*;
import com.sun.sgs.client.simple.*;

public class ChannelServerListener
  extends MySimpleClientListener
{
    protected ClientChannel channel;
    protected ChannelClient client;

    public ChannelServerListener(ChannelClient client) {
        this.client = client;
    }

    public void receivedMessage(byte[] msg) {
        System.out.println(new String(msg));
    }

    public ClientChannelListener joinedChannel(ClientChannel channel) {
        this.channel = channel;
        client.joinedChannel(channel);
        return new MyClientChannelListener(client);
    }
}
```

This class' `joinedChannel` method returns a non-null implementation of the `ClientChannelListener`. `ClientChannelListener` is an event-handling interface which is sent events related to the channel. The interface has two event-handling methods:

`public void leftChannel(ClientChannel channel)`

Notifies the client that the server has removed it from the specified channel.

`public void receivedMessage(ClientChannel channel, SessionId sender, byte[] msg)`

This method is discussed in detail in the following section on sending and receiving data.

Here is a simple `ClientChannelListener` that prints notifications when the client is added or removed from a channel:

```
import com.sun.sgs.client.*;

public class MyClientChannelListener
    implements ClientChannelListener
{
    protected ChannelClient client;

    public MyClientChannelListener(ChannelClient client) {
        this.client = client;
    }

    public void leftChannel(ClientChannel channel) {
        System.out.println("Left channel: "+channel.getName());
        client.leftChannel();
    }

    public void receivedMessage(ClientChannel channel,
                                SessionId sender,
                                byte[] msg)
    {
        System.out.println(sender+":"+new String(msg));
    }
}
```

Channel Communication

Once a Channel has been created, communication is quite similar to direct client/server communication. On both the client and the server, there are three methods for sending data that have nearly identical signatures. On the server side, they are part of the `Channel` object, on the client side they are part of the `ClientChannel` object.. Regardless of the message sending method, the sender does not receive a copy of the message through its listener.

> **Note**
>
> In all cases, the byte array used for sending the message must not be modified after the message is sent. If the byte array is modified, unpredictable errors may occur. The easiest way to work around this restriction is to clone the array prior to sending the message.

The methods for sending through the server-side `Channel` object are:

`public void send(byte[] msg)`

Sends a message to all members of this channel.

`public void send(ClientSession client, byte[] msg)`

Sends a message to a single specific client participating in the channel. If the client is not currently in the channel, no action is taken.

`public void send(Set<ClientSession> clients, byte[] msg)`

Sends a message to a subset of the clients participating in the channel. If some members of the `Set` are not participating in the channel, then no action is taken and those `ClientSessions` are ignored.

On the client side, the methods for sending through the `ClientChannel` object are:

`public void send(byte[] msg)`

Sends a message to all members of this channel.

`public void send(SessionId client, byte[] msg)`

Sends a message to a single specific client participating in the channel. If the client is not currently in the channel, no action is taken.

`public void send(Set<SessionId> clients, byte[] msg)`

Sends a message to a subset of the clients participating in the channel. If some members of the `Set` are not participating in the channel, then no action is taken then those `ClientSessions` are ignored.

Data sent through these methods is received through the `ChannelListener` and `ClientChannelListener` event handling interfaces. Again, the signatures for the methods are nearly identical.

On the server, the `ChannelListener` interface has a single method for receiving data:

`public void receivedMessage(Channel channel, ClientSession sender, byte[] msg)`

Receives a message sent on the given channel, sent by the given sender.

On the client, the `ClientChannelListener` interface has a similar method for receiving data:

`public void receivedMessage(ClientChannel channel, SessionId sender, byte[] msg)`

Receives a message sent on the given channel, sent by the given sender. If sender is null, then the game server itself is the sender of the message.

Here is an updated version of the `ChannelClientListener` class, which sends status messages back to clients on channels when clients leave or join a channel. When a client joins, it sends an announcement to all of the clients which are currently in

the channel, and then sends a separate welcome message to the client that has joined. When a client leaves a channel, it waits until that client has been removed and then sends an announcement of the departure to the remaining clients. Finally, it keeps track of all the channels a particular client is part of, so that it can send appropriate departure messages when the client is disconnected, either gracefully or forced.

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.*;

public class ChannelClientListener
    implements Serializable, ClientSessionListener
{
    protected ClientSession session;
    protected Vector<Channel> my_channels;

    public ChannelClientListener(ClientSession session)
    {
         this.session = session;
         this.my_channels = Vector<Channel>();
    }

    public Channel getChannel(String name) {
         try {
            return AppContext.getChannelManager().getChannel(name);
         } catch (NameNotBoundException ex) {
            return null;
         }
    }

    public void receivedMessage(byte[] message)
    {
        String msg = new String(message);
        if (msg.startsWith("join")) {
            // format is "join <channel-name>"
            String channelName = msg.substring(5);
            Channel c = getChannel(channelName);
            if (c != null) {
                // Get the current members.
                Set<ClientSession> current = c.getSessions();
                // Join this user
                c.join(session, null);
                // Confirm join.
                session.send("OK".getBytes());
                // Send joined message to previous members
                c.send(current,
                       (session.getName()+
                        " has joined the channel").getBytes());
```

```
            // Send welcome message to client
            c.send(session,
                   ("Welcome to "+
                    channels[i].getName()).getBytes());
            my_channels.add(c);
            return;
        }
        else {
            session.send("No such channel".getBytes());
        }
    }
    else if (msg.startsWith("leave")) {
        // format is "leave <channel-name>"
        String channelName = msg.substring(6);
        Channel c = getChannel(channelName);
        if (c != null) {
            // Leave the channel
            c.leave(session);
            // Confirm leave
            session.send("OK".getBytes());
            // Send leave notification to remaining members.
            c.send((session.getName()+
                    " left the channel").getBytes());
            my_channels.remove(c);
            return;
        }
        else {
            session.send("No such channel".getBytes());
        }
    }
    else if (msg.startsWith("create")) {
        // format is "create <channel-name>"
        String channelName = msg.substring(7);
        try {
            ChannelManager cm =
                AppContext.getChannelManager();
            cm.createChannel(chanelName, null, Delivery.RELIABLE);
            session.send("OK".getBytes());
        } catch (NameExistsException ex) {
            session.send("Channel exists.".getBytes());
        }
    }
    else {
        session.send("Unknown message".getBytes());
    }
}

public void disconnected(boolean graceful) {
    Stirng disconnect;
    if (graceful)
```

```
            disconnect = "Graceful disconnect.";
        else
            disconnect = "Hard disconnect";
        // Send departure messages
        for (Channel c : my_channels) {
            c.send((session.getName()+
                    " left the channel with a "+disconnect).getBytes());
        }
    }
}
```

Here is a client application which uses the previously introduced classes to join, leave and communicate over channels.

```
import com.sun.sgs.client.ClientChannel;
import com.sun.sgs.client.simple.SimpleClient;
import java.util.Properties;
import java.io.*;

public class ChannelClient
{
    protected String host;
    protected int port;
    protected SimpleClient sc;
    protected ClientChannel channel;

    public ChannelClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void joinedChannel(ClientChannel channel) {
        System.out.println("Joined Channel: "+channel);
        this.channel = channel;
    }

    public void leftChannel() {
        System.out.println("Left Channel: "+channel);
        this.channel = null;
    }

    public void connect() {
        // Create a client listener and client
        ChannelServerListener listen = new ChannelServerListener(this);
        sc = new SimpleClient(listen);

        // Create the login properties and login.
        Properties connectionProperties = new Properties();
        connectionProperties.setProperty("host", host);
        connectionProperties.setProperty("port", Integer.toString(port));
        try {
```

```
            sc.login(connectionProperties);
        } catch (IOException ex) {
            ex.printStackTrace();
            System.exit(1);
        }
        // Wait for login to complete.
        try {
            listen.waitForLogin();
        } catch (InterruptedException ignore) {}
            try {
            sc.send("inventory".getBytes());
            sc.send("money".getBytes());
            sc.send("stuff".getBytes());
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public void run() {
        try {
            BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
            while (true) {
                String line = in.readLine();
                if (line.length() > 0) {
                    if (line.startsWith("join") ||
                        line.startsWith("leave"))
                        sc.send(line.getBytes());
                    else if (channel != null) {
                        channel.send(line.getBytes());
                    }
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ChannelClient cc=new ChannelClient(args[0], Integer.parseInt(args[1]));
        cc.connect();
        cc.run();
    }
}
```

Space Game!

The following section describes the complete implementation of a simple multi-player space combat game. Players are pilots engaged in simple combat with each other. **Figure 6** shows two opponents playing the game.

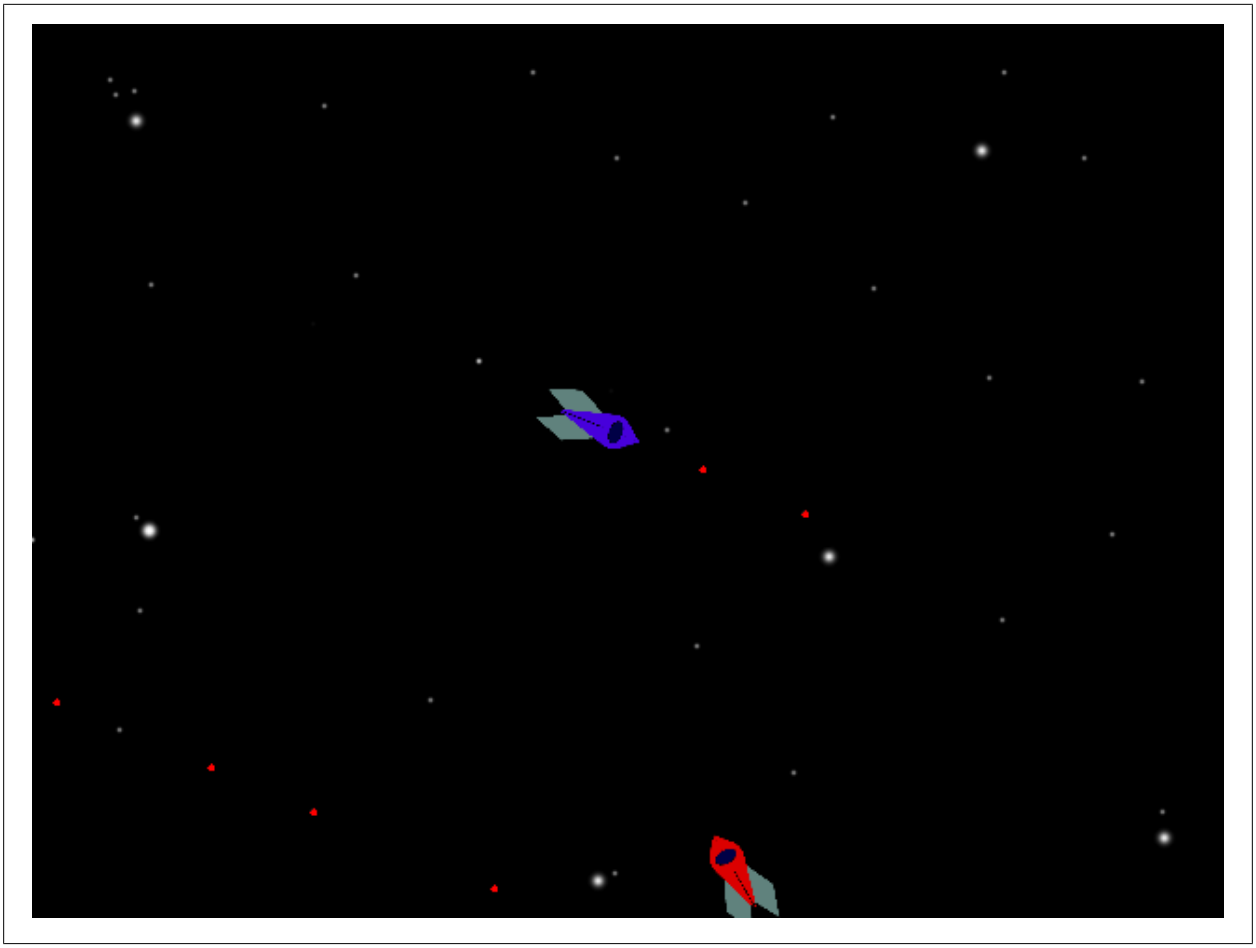

Figure 6. A screen shot of Space Game! in action.

This section is organized a follows: first the data structures for the game state are described, then the main game application, finally the client application is briefly discussed.

Game State

Space Game! is a relatively simple game. There are basically only three types of objects which make up the internal state of the game: Ships, Bullets and the game clock. Each ship obviously represents a particular client in the game. Each client can fire multiple Bullets in an attempt to destroy other player's ships. The game clock is the central arbiter of time in the game. It is used to synchronize action. The class diagram for this game state is shown in **Figure 7**.

The first thing to notice is that all of the game state objects implement a shared moveable interface. The moveable interface simply requires two methods:

```
public interface Moveable {
   public void move();
   public int getTicks();
}
```

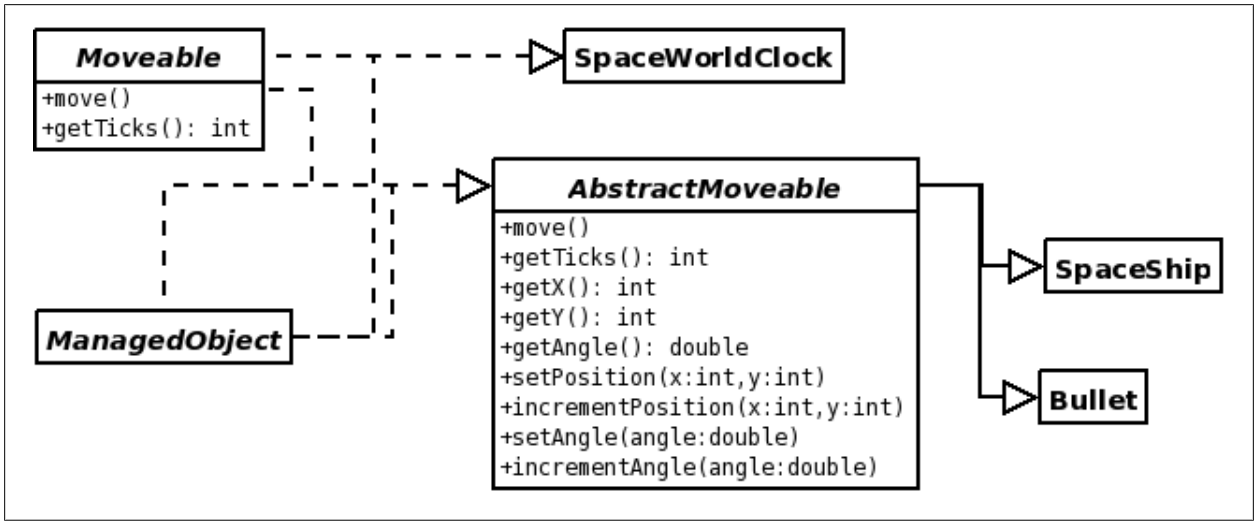

Figure 7. The object model for game state

The `move()` method actually moves the moveable object. The `getTicks()` function returns the current local time of the move object. For all moveable objects except the `SpaceWorldClock`, this timer is synchronized with the global `SpaceWorldClock`. The `SpaceWorldClock` is a simple implementation of the moveable interface that simply updates the clock tick each time its move method is called:

```
import com.sun.sgs.app.ManagedObject;
import java.io.Serializable;

public class SpaceWorldClock
    implements ManagedObject, Serializable, Moveable
{
    protected int ticks;

    public SpaceWorldClock() {
        ticks = 0;
    }

    public void move() {
        ticks++;
    }

    public int getTicks() {
        return ticks;
    }
}
```

In Space Game! bullets and space ships are each represented as five values: the (x,y) position of the object, the angular orientation of the object, the forward velocity of the object and an identifier that uniquely specifies the object. Both bullets and space ships move in the same manner: The forward velocity is broken down

into delta_x and delta_y components using trigonometry and orientation of the object, and these deltas are used to modify the objects (x,y) position. Because both Bullets and SpaceShips share many of the same methods (for example getting and setting position), these methods were abstracted out into the `AbstractMoveable` super class.

```
import com.sun.sgs.app.*;

public class AbstractMoveable
    implements Moveable, java.io.Serializable
{
    protected int tick;
    protected double vel;
    protected int id;

    protected static int state_id = 0;

    protected ManagedReference state;

    public AbstractMoveable(int id, double x, double y, double angle,
                            double vel, int tick)
    {
       this.id = id;

       State state = new State(x, y, angle, id);
       DataManager dm = AppContext.getDataManager();
       dm.setBinding("state_"+state_id++, state);
       this.state = dm.createReference(state);

       this.vel = vel;
       this.tick = tick;
    }

    public void move() {
       State st = state.getForUpdate(State.class);
       st.incrementPosition(Math.cos(st.getAngle())*vel,
                            Math.sin(st.getAngle())*vel);

       tick++;
    }

    public int getTicks() {
       return tick;
    }

    public int getId() {
       return id;
    }
```

```
    public int getX() {
       return state.get(State.class).getX();
    }

    public int getY() {
       return state.get(State.class).getY();
    }

    public int getAngle() {
       State st = state.get(State.class);
       int ang = (int)Math.toDegrees(st.getAngle());
       while (ang < 0)
           ang+=360;
       return ang;
    }
}
```

Internally, the `AbstractMoveable` class uses a `State` class to contain the actual state of the moveable object. This distinction is made to reduce contention for the object. When the moveable object is moved, only the state is locked for updating. This enables other objects to access the `AbstractMoveable` object for other purposes (for example updating the velocity) without introducing contention. The use of this pattern for contention reduction is suggested by the developers of the Darkstar application server.

```
import com.sun.sgs.app.ManagedObject;
import java.io.Serializable;

public class State
    implements ManagedObject, Serializable
{
    protected double x, y;
    protected int id;
    protected double angle;

    public State(double x, double y, double angle, int id) {
       this.x = x;
       this.y = y;
       this.id = id;
       this.angle = angle;
    }

    public double getAngle() {
       return angle;
    }

    public int getX() {
       return (int)x;
    }
```

```
    public int getY() {
       return (int)y;
    }

    public int getId() {
       return id;
    }

    public void setAngle(double angle) {
       this.angle = angle;
       while (angle < 0) angle += 2*Math.PI;
       while (angle > 2*Math.PI) angle -= 2*Math.PI;
    }

    public void incrementAngle(double inc) {
       setAngle(angle+inc);
    }

    public void setPosition(double x, double y) {
       this.x = x;
       this.y = y;

       if (this.x < 0)
           this.x = SpaceConstants.WIDTH-1;
       else if (this.x >= SpaceConstants.WIDTH)
           this.x = 0;

       if (this.y < 0)
           this.y = SpaceConstants.HEIGHT-1;
       else if (this.y >= SpaceConstants.HEIGHT)
           this.y = 0;
    }

    public void incrementPosition(double dx, double dy) {
       setPosition(x+dx, y+dy);
    }

    public String toString() {
       return id+" @ "+x+","+y;
    }
}
```

Both `Bullet` and `SpaceShip` subclass `AbstractMoveable` and add some functionality that is specific to the subclass. The `Bullet` subclass holds both its own id and the id of the client that fired it, ensuring that clients can not shoot themselves. The `Bullet` class also contains a time-to-live (TTL) variable which is used to delete the bullet after a specific time interval has passed. The `Bullet` class also holds on to

the periodic task that actually animates the bullet. This task is also terminated when the bullet's time to live expires.

```
import com.sun.sgs.app.*;

public class Bullet
    extends AbstractMoveable
{
    protected int client_id;
    protected int ttl;
    protected PeriodicTaskHandle task_handle;
    protected ManagedReference bullet_list;

    public Bullet(double x, double y, double angle,
                int client_id, int id, int ticks)
    {
       super(id, x, y, angle, SpaceConstants.BULLET_SPEED, ticks);
       this.client_id = client_id;
       this.ttl = SpaceConstants.BULLET_LIFE;
    }

    public void setTaskHandle(PeriodicTaskHandle h) {
       this.task_handle = h;
    }

    public void setBulletList(ManagedReferenceList<Bullet> bl) {
       bullet_list = AppContext.getDataManager().createReference(bl);
    }

    public int getClientId() {
       return client_id;
    }

    public int getTimeToLive() {
       return ttl;
    }

    public void move() {
       super.move();

       ttl--;
       if (ttl <= 0) {
           task_handle.cancel();
           ManagedReferenceList<Bullet> bl =
              bullet_list.getForUpdate(ManagedReferenceList.class);
           bl.remove(this);
           DataManager dm = AppContext.getDataManager();
           dm.removeBinding("bullet_"+client_id+"_"+id);
           dm.removeObject(this);
       }
```

```
    }
}
```

The `SpaceShip` class adds a number of methods to respond to enable it to respond to client input and game events like getting shot by another client's bullet.

```
import com.sun.sgs.app.*;
import java.util.Random;

public class SpaceShip
    extends AbstractMoveable
{
    int bullet_id;
    Random r;

    public SpaceShip(int x, int y, int id, int ticks) {
       super(id, x, y, 0, 0, ticks);
       this.r = new Random();
    }

    public void explode() {
       State s = state.getForUpdate(State.class);
       s.setPosition(r.nextInt(SpaceConstants.WIDTH),
                     r.nextInt(SpaceConstants.HEIGHT));
       s.setAngle(r.nextInt(360));
       this.vel = 0;
    }

    public void fire() {
       State s = state.get(State.class);
       Bullet b = new Bullet(s.getX(),s.getY(),s.getAngle(),
                             id, bullet_id, tick);
       DataManager dm = AppContext.getDataManager();
       dm.setBinding("bullet_"+id+"_"+bullet_id, b);
       ManagedReferenceList<Bullet> bullets
           = dm.getBinding(SpaceConstants.BULLET_LIST,
                           ManagedReferenceList.class);
       bullets.add(b);

       TaskManager tm = AppContext.getTaskManager();

       PeriodicTaskHandle pth =
           tm.schedulePeriodicTask(new MoveableTask(b),0,
                                   SpaceConstants.GAME_PERIOD);
       b.setTaskHandle(pth);
       b.setBulletList(bullets);

       bullet_id++;
    }
```

```
    public void incrementVelocity(int inc) {
       this.vel += inc;
       vel = Math.min(SpaceConstants.MAX_SPEED, vel);
       vel = Math.max(-SpaceConstants.MAX_SPEED, vel);

    }

    public void incrementAngle(double inc) {
       state.getForUpdate(State.class).incrementAngle(inc);
    }
}
```

Together, these classes make up the entire game state of Space Game!. However, these classes are merely static data structures, the main server application is required to drive the action of the game.

Game Server

The game server's classes implement the core application logic which implements Space Game!. This code handles connections from clients, client server communication, and animation tasks, which cause space ships and bullets to move through the world. There are four major components to the game server code:

- Lists of the currently active clients and other managed objects
- The game server "main" that accepts client connections
- The per-client listener that receives client communication
- Periodic tasks that move the game forward through time

In the game, there are many different instances of the same type of object. For example, multiple clients connected to the server, each corresponding to multiple space ships flying across the screen, and multiple bullets fired by each individual ship. To keep track of all of this, the Space Game! uses two different list objects. Because it exists in the Darkstar architecture, each of these lists is itself a `ManagedObject`, this enables multiple different pieces of the Space Game! server to access the same data. The first list object, `ManagedReferenceList<T>` is a templatized generic class which contains a homogeneous list of `ManagedObjects`:

```
import com.sun.sgs.app.*;
import java.util.List;
import java.util.Vector;

public class ManagedReferenceList<T extends ManagedObject>
    implements ManagedObject, java.io.Serializable
{
    protected List<ManagedReference> store;
    protected Class<T> t_class;
```

```
    public ManagedReferenceList() {
       this(new Vector<ManagedReference>());
    }

    public ManagedReferenceList(List<ManagedReference> store) {
       this.store = store;
    }

    public void add(T obj) {
       DataManager dm = AppContext.getDataManager();
       store.add(dm.createReference(obj));
       if (t_class == null)
           t_class = (Class<T>)obj.getClass();
    }

    public void remove(T obj) {
       DataManager dm = AppContext.getDataManager();
       store.remove(dm.createReference(obj));
    }

    public int size() {
       return store.size();
    }

    public void clear() {
       store.clear();
    }

    public T get(int ix) {
       return store.get(ix).get(t_class);
    }
}
```

Unfortunately, though there are in fact managed by the application server, `ClientSession` objects do not implement the `ManagedObject` interface. Consequently, the game server needs a separate list class to maintain the list of currently connected clients:

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.Vector;

public class ClientList
    implements ManagedObject, Serializable
{
    Vector<ClientSession> clients;

    public ClientList() {
       this.clients = new Vector<ClientSession>();
    }
```

```
    public void add(ClientSession client) {
       this.clients.add(client);
    }

    public void remove(ClientSession client) {
       this.clients.remove(client);
    }

    public int getClientCount() {
       return clients.size();
    }

    public ClientSession getClient(int ix) {
       return clients.get(ix);
    }
}
```

The rest of the code in the implementation of the game server uses these lists to manage the data stored in the current game state.

Client communication with the Space Game! server begins with server implementation of the `AppListener` interface that initializes the server and accepts connections from client. When the server is initialized it creates and binds all of the global game state variables. It also creates and starts two periodic tasks, one that broadcasts status information to each client and another that maintains the global time clock. When a client connects, the server adds the new client to the client list and also creates a corresponding SpaceShip instance and adds it to the list of SpaceShips. The server then sends the client its identifying number. Finally, the server returns a new `lSpaceClientListener` object that receives communication from the client. Here is the implementation of the `SpaceWorldMain` class:

```
import com.sun.sgs.app.*;
import java.util.Properties;
import java.util.Random;
import java.io.Serializable;


public class SpaceWorldMain
    implements AppListener, Serializable
{
    Random rand;
    int id;
    PeriodicTaskHandle statusHandle, clockHandle;
    ManagedReference clientlist;
    ManagedReference shiplist;
    ManagedReference clock;
```

```
public void initialize(Properties p) {
   DataManager dm = AppContext.getDataManager();
   ClientList cl = new ClientList();
   ManagedReferenceList<SpaceShip> sl =
       new ManagedReferenceList<SpaceShip>();
   ManagedReferenceList<Bullet> bl =
       new ManagedReferenceList<Bullet>();
   SpaceWorldClock clock = new SpaceWorldClock();

   dm.setBinding(SpaceConstants.CLIENT_LIST, cl);
   dm.setBinding(SpaceConstants.SHIP_LIST, sl);
   dm.setBinding(SpaceConstants.BULLET_LIST, bl);
   dm.setBinding(SpaceConstants.CLOCK, clock);

   clientlist = dm.createReference(cl);
   shiplist = dm.createReference(sl);
   this.clock = dm.createReference(clock);
   this.rand = new Random();

   TaskManager tm = AppContext.getTaskManager();
   StatusTask stat = new StatusTask();
   statusHandle =
       tm.schedulePeriodicTask(stat, 0, SpaceConstants.GAME_PERIOD);

   id = 0;

   System.out.println("Space World Initialized.");

   MoveableTask mt = new MoveableTask(clock, false);
   clockHandle =
       tm.schedulePeriodicTask(mt, 0, SpaceConstants.GAME_PERIOD);
}

public ClientSessionListener loggedIn(ClientSession session)
{
   DataManager dm = AppContext.getDataManager();
   ClientList cl = clientlist.getForUpdate(ClientList.class);
   cl.add(session);
   ManagedReferenceList<SpaceShip> sl =
       shiplist.getForUpdate(ManagedReferenceList.class);
   SpaceShip ss = new SpaceShip
       (Math.abs(rand.nextInt()%SpaceConstants.WIDTH),
        Math.abs(rand.nextInt()%SpaceConstants.HEIGHT), id,
        clock.get(SpaceWorldClock.class).getTicks());
   sl.add(ss);

   dm.setBinding(session.getName()+"_ship", ss);


   TaskManager tm = AppContext.getTaskManager();
```

```
        MoveableTask mt = new MoveableTask(ss);
        PeriodicTaskHandle pth =
            tm.schedulePeriodicTask(mt, 0, SpaceConstants.GAME_PERIOD);

        byte[] msg = new byte[5];
        msg[0] = SpaceConstants.ID;
        SpaceUtils.writeInt(id, msg, 1);
        session.send(msg);
        id++;
        return new SpaceClientListener(session, ss, sl, pth);
    }
}
```

The SpaceClientListener handles client messages and disconnects. There are three different types of messages that can be sent from the client: adjusting the speed of the ship (SpaceConstants.DX), adjusting the rotation of the ship (SpaceConstants.DX) and firing a bullet (SpaceConstants.FIRE). The SpaceClientListener determines the type of the message, parses the message data and then calls the appropriate function on the SpaceShip object. When a client disconnects, the SpaceClientListener performs clean up: cancelling the periodic task which animates the space ship, removing the client from this list of all clients, and removing the corresponding SpaceShip object from the list of all SpaceShips. Here is the SpaceClientListener class:

```
import com.sun.sgs.app.*;
import java.io.Serializable;
import java.util.Random;

public class SpaceClientListener
    implements ClientSessionListener, Serializable
{
    ClientSession client;
    PeriodicTaskHandle taskHandle;
    ManagedReference ship;
    ManagedReference ship_list;

    public SpaceClientListener(ClientSession client, SpaceShip ss,
                               ManagedReferenceList<SpaceShip> sl,
                               PeriodicTaskHandle th)
    {
        this.client = client;
        this.taskHandle = th;
        this.ship = AppContext.getDataManager().createReference(ss);
        this.ship_list = AppContext.getDataManager().createReference(sl);
    }

    public void disconnected(boolean force) {
        taskHandle.cancel();
```

```
        DataManager dm = AppContext.getDataManager();
        ClientList cl =
            dm.getBinding(SpaceConstants.CLIENT_LIST, ClientList.class);
        cl.remove(client);

        SpaceShip ss = ship.get(SpaceShip.class);

        ManagedReferenceList<SpaceShip> sl =
            ship_list.getForUpdate(ManagedReferenceList.class);
        sl.remove(ss);

        dm.removeBinding(client.getName()+"_ship");
        dm.removeObject(ss);
    }

    public void receivedMessage(byte[] msg) {
        SpaceShip ss = ship.getForUpdate(SpaceShip.class);

        switch(msg[0]) {
        case SpaceConstants.DX:
            if (msg[1] == SpaceConstants.UP)
               ss.incrementVelocity(1);
            else if (msg[1] == SpaceConstants.DOWN)
               ss.incrementVelocity(-1);
            break;
        case SpaceConstants.ROTATE:
            if (msg[1] == SpaceConstants.UP)
               ss.incrementAngle(0.1);
            else if (msg[1] == SpaceConstants.DOWN)
               ss.incrementAngle(-0.1);
            break;
        case SpaceConstants.FIRE:
            ss.fire();
            break;
        default:
            System.err.println("Unknown message type: "+msg[0]);
        }
    }
}
```

The remaining components of the game server implementation are the periodic tasks that animate Moveable objects and broadcast status information to the clients. Every instance of a Moveable object that is created in the game has a corresponding periodic MoveableTask that is responsible for animating it. The `MoveableTask` is only responsible for moving that one object. The game was designed in this manner to reduce contention. Because each method that calls `ManagedReference.getForUpdate(...)` locks that object for the duration of the

method. Having a single method which updated the location of all moveable objects would lock all of the Moveable objects in the game, even after they had been moved, preventing others from accessing the objects. Further, a single mover task could not be distributed across multiple CPUs or multiple servers if such resources were available. Associating a different task with each moveable object reduces contention and maximizes use of computational resources. The `MoveableTask` also optionally keeps itself synchronized with the global `SpaceWorldClock`. This ensures that all moveable objects are synchronized at the same time and makes collision checking between bullets and ships feasible.

```
import com.sun.sgs.app.*;

public class MoveableTask
    implements Task, java.io.Serializable
{
    ManagedReference clock;
    ManagedReference mover;
    Class clazz;
    boolean timecheck;

    public MoveableTask(Moveable mover) {
       this(mover, true);
    }

    public MoveableTask(Moveable mover, boolean timecheck)
    {
       DataManager dm = AppContext.getDataManager();
       this.mover = dm.createReference(mover);
       this.clock = dm.createReference
           (dm.getBinding(SpaceConstants.CLOCK, SpaceWorldClock.class));
       this.clazz = mover.getClass();
       this.timecheck = timecheck;
    }

    public void run() {
       try {
           Moveable ss = (Moveable)mover.get(clazz);
           SpaceWorldClock cl = clock.get(SpaceWorldClock.class);
           if (timecheck)
              while (cl.getTicks() > ss.getTicks())
                  ss.move();
           else
              ss.move();
       } catch (Exception ex) {
           ex.printStackTrace();
       }
    }
```

```
}
```

In contrast to the `MoveableTask`, there is just a single `StatusTask` that is responsible for broadcasting game state information back to the clients. This is because broadcasting state information only requires reader access to an object and consequently does not introduce any significant contention for resources into the system. Also, send a single large message across the network is more efficient that sending a series of small messages because of the network overhead associated with sending each individual message. Because the `StatusTask` already scans through both the list of Bullets and the list of SpaceShips, the `StatusTask` also determines when a bullet has collided with a ship. When a ship is hit, `SpaceShip.explode()` is called and the ship is moved to a new location prior to sending its status to the client.

```
import com.sun.sgs.app.*;

public class StatusTask
    implements Task, java.io.Serializable
{
    ManagedReference clients;
    ManagedReference shiplist;
    ManagedReference bulletlist;

    byte[] status;

    protected static final int HIT_THRESHOLD=400;

    public StatusTask() {
       this.status = new byte[1024];
       DataManager dm = AppContext.getDataManager();
       ClientList cl =
           dm.getBinding(SpaceConstants.CLIENT_LIST, ClientList.class);
       clients = dm.createReference(cl);

       ManagedReferenceList<SpaceShip> sl =
           dm.getBinding(SpaceConstants.SHIP_LIST, ManagedReferenceList.class);
       shiplist = dm.createReference(sl);

       ManagedReferenceList<Bullet> bl =
           dm.getBinding(SpaceConstants.BULLET_LIST,
                         ManagedReferenceList.class);
       bulletlist = dm.createReference(bl);
    }

    protected boolean isHit(Bullet b, SpaceShip ss)
    {
       while (b.getTicks() < ss.getTicks())
           b.move();
```

```
    while (ss.getTicks() < b.getTicks())
        ss.move();

    double dx = ss.getX()-b.getX();
    double dy = ss.getY()-b.getY();
    double dist = dx*dx+dy*dy;
    return (dist < HIT_THRESHOLD);
}


public void run() {
    try {
        ManagedReferenceList<SpaceShip> sl =
           shiplist.get(ManagedReferenceList.class);
        ManagedReferenceList<Bullet> bullets
           = bulletlist.get(ManagedReferenceList.class);

        // Resize as needed
        while (sl.size()*SpaceConstants.STATUS_SIZE+
              bullets.size()*SpaceConstants.STATUS_SIZE+20 >
              status.length)
           {
               status = new byte[status.length*2];
           }

        // Header
        status[0] = SpaceConstants.STATUS;
        SpaceUtils.writeInt(sl.size(), status, 1);

        // Write info for each ship
        for (int i=0;i<sl.size();i++) {
           SpaceShip ss = sl.get(i);
           for (int j=0;j<bullets.size();j++) {
               Bullet b = bullets.get(j);
               if (b.getClientId() != ss.getId() &&
                  isHit(b, ss))
                  {
                      ss.explode();
                      bullets.remove(b);
                  }
           }
           SpaceUtils.writeInt(ss.getId(), status,
                             5+i*SpaceConstants.STATUS_SIZE);
           SpaceUtils.writeInt(ss.getX(), status,
                             5+i*SpaceConstants.STATUS_SIZE+4);
           SpaceUtils.writeInt(ss.getY(), status,
                             5+i*SpaceConstants.STATUS_SIZE+8);
           SpaceUtils.writeInt(ss.getAngle(), status,
                             5+i*SpaceConstants.STATUS_SIZE+12);
        }
```

```
            int base = 5+sl.size()*SpaceConstants.STATUS_SIZE;

            SpaceUtils.writeInt(bullets.size(), status, base);
            base += 4;

            for (int i=0;i<bullets.size();i++) {
               Bullet b = bullets.get(i);
               SpaceUtils.writeInt(b.getId(), status,
                                   base+i*SpaceConstants.STATUS_SIZE);
               SpaceUtils.writeInt(b.getX(), status,
                                   base+i*SpaceConstants.STATUS_SIZE+4);
               SpaceUtils.writeInt(b.getY(), status,
                                   base+i*SpaceConstants.STATUS_SIZE+8);
               SpaceUtils.writeInt(b.getAngle(), status,
                                   base+i*SpaceConstants.STATUS_SIZE+12);
            }

            ClientList cl = clients.get(ClientList.class);
            int count = cl.getClientCount();
            for (int i=0;i<count;i++) {
               ClientSession cs = cl.getClient(i);
               if (cs.isConnected())
                   cs.send(status);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Game Client

The code for the Space Game! client is significantly simpler than the server, there are only five classes which make up the client. The main client application class is `SpaceClient`, it creates the GUI for the game, handles client key-presses and sends appropriate messages to the server.

```
import com.sun.sgs.client.ClientChannel;
import com.sun.sgs.client.simple.SimpleClient;
import java.util.Properties;
import java.io.*;
import javax.swing.*;
import java.awt.event.*;

public class SpaceClient
    implements KeyListener
{
    protected String host;
    protected int port;
    protected int id;
```

```
protected SimpleClient sc;
protected ClientChannel channel;
protected SpacePanel panel;
protected byte[] msg = new byte[2];

public SpaceClient(String host, int port) {
   this.host = host;
   this.port = port;
   this.panel = new SpacePanel();
}

public SpacePanel getPanel() {
   return panel;
}

public void joinedChannel(ClientChannel channel) {
   this.channel = channel;
}

public void leftChannel() {
   this.channel = null;
}

public void fire()
   throws IOException
{
   msg[0] = SpaceConstants.FIRE;
   sc.send(msg);
}

public void turnLeft()
   throws IOException
{
   msg[0] = SpaceConstants.ROTATE;
   msg[1] = SpaceConstants.DOWN;
   sc.send(msg);
}

public void turnRight()
   throws IOException
{
   msg[0] = SpaceConstants.ROTATE;
   msg[1] = SpaceConstants.UP;
   sc.send(msg);
}

public void speedUp()
   throws IOException
{
   msg[0] = SpaceConstants.DX;
```

```
        msg[1] = SpaceConstants.UP;
        sc.send(msg);
    }

    public void slowDown()
        throws IOException
    {
        msg[0] = SpaceConstants.DX;
        msg[1] = SpaceConstants.DOWN;
        sc.send(msg);
    }

    public void connect() {
        // Create a client listener and client
        SpaceServerListener listen = new SpaceServerListener(this);
        sc = new SimpleClient(listen);

        // Create the login properties and login.
        Properties connectionProperties = new Properties();
        connectionProperties.setProperty("host", host);
        connectionProperties.setProperty("port", Integer.toString(port));
        try {
            sc.login(connectionProperties);
        } catch (IOException ex) {
            ex.printStackTrace();
            System.exit(1);
        }
        // Wait for login to complete.
        try {
            listen.waitForLogin();
        } catch (InterruptedException ignore) {}
    }

    public void run() {
        while (true) {
            try {
               Thread.currentThread().sleep(1000);
            } catch (InterruptedException ex) {}
        }
    }

    public void setId(int id) {
        this.id = id;
        if (panel != null)
            this.panel.setId(id);
    }

    public void keyPressed(KeyEvent ev) {
        try {
            switch (ev.getKeyCode()) {
```

```
            case KeyEvent.VK_SPACE:
               fire();
               break;
            case KeyEvent.VK_LEFT:
               turnLeft();
               break;
            case KeyEvent.VK_RIGHT:
               turnRight();
               break;
            case KeyEvent.VK_UP:
               speedUp();
               break;
            case KeyEvent.VK_DOWN:
               slowDown();
               break;
            case KeyEvent.VK_ESCAPE:
               sc.logout(true);
               System.exit(0);
               break;
            }
         } catch (IOException ex) {
             ex.printStackTrace();
         }
      }

      public void keyTyped(KeyEvent ev) {}
      public void keyReleased(KeyEvent eve) {}

      public static void main(String[] args) {
         SpaceClient sc=new SpaceClient(args[0], Integer.parseInt(args[1]));
         sc.connect();

         JFrame mainframe = new JFrame("Space Game!");
         mainframe.getContentPane().add(sc.getPanel());
         mainframe.pack();
         mainframe.addKeyListener(sc);
         mainframe.setVisible(true);

         sc.run();
      }
}
```

The Space Game! client uses the `SpaceServerListener` class to receive messages from the server. The client only ever receives two types of messages: an ID message that specifies the client's ID number and a STATUS message, which provides the current state of the game. The ID message is only sent once, when the client connects. The STATUS message is sent repeatedly as the game state changes.

```
import com.sun.sgs.client.*;
import com.sun.sgs.client.simple.*;
```

```
public class SpaceServerListener
  extends MySimpleClientListener
{
    protected ClientChannel channel;
    protected SpaceClient client;
    protected GameStatus gs;

    public SpaceServerListener(SpaceClient client) {
       this.client = client;
       this.gs = new GameStatus();
    }

    public void receivedMessage(byte[] msg) {
       switch(msg[0]) {
       case SpaceConstants.STATUS:
           gs = new GameStatus();
           gs.load(msg);
           client.getPanel().setStatus(gs);
           client.getPanel().repaint();
           break;
       case SpaceConstants.ID:
           int id = SpaceUtils.readInt(msg, 1);
           client.setId(id);
           break;
       default:
           System.err.println("Received unknown message: "+msg[0]);
           break;
       }
    }

    public ClientChannelListener joinedChannel(ClientChannel channel) {
       this.channel = channel;
       client.joinedChannel(channel);
       return null;
    }
}
```

When the client receives a STATUS message, it is parsed into a `GameStatus` object. The current `GameStatus` is maintained as two lists of Position objects, one for space ships and one for bullets.

```
import java.util.Vector;

public class GameStatus {
    Vector<Position> ship_positions;
    Vector<Position> bullet_positions;

    public GameStatus() {
       ship_positions = new Vector<Position>();
```

```
        bullet_positions = new Vector<Position>();
    }

    public Vector<Position> getShipPositions() {
        return ship_positions;
    }

    public Vector<Position> getBulletPositions() {
        return bullet_positions;
    }

    protected Position readPosition(byte[] data, int start) {
        int id = SpaceUtils.readInt(data, start);
        int x = SpaceUtils.readInt(data, start+4);
        int y = SpaceUtils.readInt(data, start+8);
        int a = SpaceUtils.readInt(data, start+12);

        return new Position(x, y, Math.toRadians(a), id);
    }

    public void load(byte[] msg) {
        ship_positions.clear();
        bullet_positions.clear();

        int count = SpaceUtils.readInt(msg, 1);
        for (int i=0;i<count;i++) {
            ship_positions.add
               (readPosition(msg, 5+i*SpaceConstants.STATUS_SIZE));
        }
        int base = 5+count*SpaceConstants.STATUS_SIZE;
        count = SpaceUtils.readInt(msg, base);
        base += 4;
        for (int i=0;i<count;i++) {
            bullet_positions.add
               (readPosition(msg, base+i*SpaceConstants.STATUS_SIZE));
        }
    }

    public String toString() {
        StringBuffer buff = new StringBuffer("Status: \nShips:");
        for (int i=0;i<ship_positions.size();i++)
            buff.append(ship_positions.get(i));
        for (int i=0;i<bullet_positions.size();i++)
            buff.append(bullet_positions.get(i));
        return buff.toString();
    }
}

public class Position {
    protected int x, y, id;
    protected double angle;
```

```
    public Position(int x, int y, double angle, int id) {
       this.x = x;
       this.y = y;
       this.id = id;
       this.angle = angle;
    }

    public double getAngle() {
       return angle;
    }

    public int getX() {
       return x;
    }

    public int getY() {
       return y;
    }

    public int getId() {
       return id;
    }

    public String toString() {
       return id+" @ "+x+","+y;
    }
}
```

Whenever a new `GameStatus` is received, it is passed to the `SpacePanel`, which graphically displays the updated game state for the user.

```
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.util.Vector;
import java.awt.geom.AffineTransform;

public class SpacePanel extends JPanel {
    protected Dimension d;
    protected GameStatus status;
    protected int id;
    protected BufferedImage ship, my_ship;
    protected BufferedImage background;
    protected AffineTransform trans;
    protected int off_x, off_y;
    protected Vector<Position> ships;
```

```
protected Vector<Position> bullets;

public SpacePanel() {
   d = new Dimension(SpaceConstants.WIDTH, SpaceConstants.HEIGHT);
   status = null;
   try {
       ship = javax.imageio.ImageIO.read
          (new java.io.FileInputStream("ship.png"));
       my_ship = javax.imageio.ImageIO.read
          (new java.io.FileInputStream("my_ship.png"));
       background = javax.imageio.ImageIO.read
          (new java.io.FileInputStream("stars.png"));
       off_x = (int)(ship.getWidth()*0.2);
       off_y = (int)(ship.getHeight()/2*0.2);
   }
   catch (java.io.IOException ex) {
       ex.printStackTrace();
   }
   trans = new AffineTransform();
}

public Dimension getPreferredSize() {
   return d;
}

public Dimension getMinimumSize() {
   return d;
}

public void setStatus(GameStatus status) {
   ships = status.getShipPositions();
   bullets = status.getBulletPositions();
   this.status = status;
}

public void setId(int id) {
   this.id = id;
   System.out.println("ID is "+id);
}

public void paint(Graphics gc) {
   Graphics2D g = (Graphics2D)gc;
   g.drawImage(background, 0, 0, this);
   if (status != null) {
       for (Position p : ships) {
          trans.setToTranslation(p.getX(),p.getY());
          trans.rotate(p.getAngle());
          trans.translate((int)(-0.5*off_x), -off_y);
          trans.scale(0.2,0.2);
          if (p.getId() == id)
```

```
                g.drawRenderedImage(my_ship, trans);
            else
                g.drawRenderedImage(ship, trans);
        }
        g.setColor(Color.red);
        for (Position p : bullets) {
            g.fillOval(p.getX()-2, p.getY()-2, 4, 4);
        }
    }
  }
}
```

A. Serialization

Serialization is the process of taking the in-memory representation of an object and storing it (persisting) to a file so that it survives even after the program which created it exits. In comparison to other languages, Java provides straightforward support for serialization. However, there are several steps necessary for successful serialization.

First, your object must implement the `java.io.Serializable` interface. This interface simply marks your object as semantically able to be serialized. Not every type of Java object can be serialized. Its easy to imagine how a `String` can be written to and recovered from a file. After all, its just a list of characters, that's basically what a file is anyway. However, its much more difficult to imagine how an object like a `Thread` could be serialized. A `Thread` is much more than just data, its a running programming with its own context. As you might imagine, in Java `Strings` implement the `Serializable` interface, `Threads` do not.

Second, all of the instance variables in your object must be serializable. Fortunately, most of the data objects (`Strings`, `Vectors`, etc) in the main Java library are serializable. As I mentioned before, some, like `Thread` are not. When in doubt, look at the documentation. But what if your class needs to have some non-serializable object as an instance variable? This is unlikely in Darkstar, but perfectly possible. In that case, you need to mark that object as `transient`. For example:

```
protected transient Thread myThreadInstance;
```

The `transient` keyword indicates to Java that an instance variable should not be serialized to disk. When an object is read in from disk (de-serialized), transient variables are set to null (for Objects) or their default value (for primitives like int and boolean).

The third consideration in creating serializable objects is version control. As you develop a game, your game objects are likely to change and evolve. This means that between running your code to test your game, recompiling and re-running the

code you may have changed the objects which are persistently stored to disk. For example, you might add a new instance variable to an object. When this happens, the version of the object stored on the disk is out of sync with the object used in your program. By default, Java treats a modified object as incompatible. It will throw an exception when it attempts to deserialize the older version. In order to prevent this, you need to put an instance variable in your object of type long named `serialVersionUID`. This field should be declared `static` and `final`. For example:

```
static final long serialVersionUID = 1L;
```

As long as two versions of a class have the same `serialVersionUID`, Java will attempt to deserialize an older version of an object into a newer version. This deserialization will work so long as the changes you have made to the object are compatible. Compatible changes include adding or removing fields or methods. Changes to the object hierarchy result in incompatible changes, which will still throw exceptions. If you add a field which is not present in the older version of the object it will be instantiated with its default value.

If you make incompatible changes and your server will not start because it crashes when deserializing objects, you may need to clear the entire object store of your server. This is also sometimes useful to do when you want to start a brand new server without any record of your previous gaming. To accomplish this you need to delete the contents of the dsdb directory in your game's data folder.