Playing on the Network

Playing against the computer isn't comparable to competing—or cooperating—with human players in the same virtual world. Just as computer games are as old as computers, multiplayer games are as old as the Internet. Networking is, however, not just an optional feature that you add to an existing game like a particle effect.

This optional chapter assumes that you have intermediate understanding of Java networking and serialization. You find the jMonkeyEngine networking support, colloquially referred to as SpiderMonkey, in the `com.jme3.network` package.

In this chapter you will learn:

- How players connect to a central game server
- How to send game events over the network
- How to choose between fast or reliable messaging
- How to keep a multiplayer game world synchronized

Ready to invite some fellow adventurers over into your virtual world and solve the next mission together?

Offline versus online

Not every game concept makes sense as a network game, but many benefit from multiplayer competition. Before you get your hands on some networking code, you should get an idea of what your options are:

- **Single-player desktop games**: These games are played exclusively offline. If you have never wrote a video game before, write an offline game first to get the hang of the game engine.

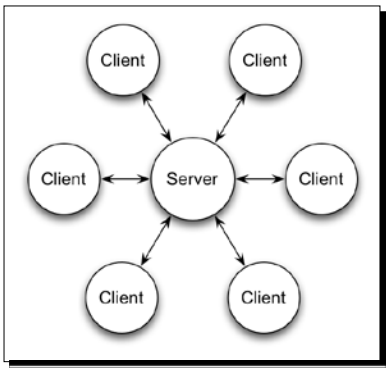
  Some single-player desktop and mobile games share highscores on a central server. This is the simplest type of game where you "compete against people online" without actually playing online. If this is your first network application, adding an online highscores feature to your existing desktop game is a good exercise.

- **Multiplayer games**: These games are played either on a LAN or over the Internet. Each player runs a game client locally. The players meet up by connecting to one game server that maintains the game state and tracks the score for all players. The game server is either hosted by one of the players, or provided by the game publisher. If you understand Java networking, threading, and serialization, developing a multiplayer game is an attainable goal.
- **Massively multiplayer online games (MMOG)**: In these games, thousands of players interact in one virtual world. MMOGs differ from multiplayer games in terms of volume: players trigger a massive amount of interactions, messages are synchronized in massive data centers, and the game is kept running by massive support departments. Unless you write multiplayer games for breakfast, a MMOG is not something that you and a few pals can pull out of a hat next to having a day job.

  Writing a multi player game is an achievable goal for experienced Java developers—and it gives you the "street cred" needed to start the MMOG team of your dreams afterward.

How to make your game a star

In multiplayer games, each player runs a client application, and connects to one central game server. This topology is referred to as a star network. One central game server communicates with several clients.



What's typical about a star network is that clients never send messages to other clients. All communication goes through one hub, the central server. This centralization decreases network traffic, because you can put the server in charge of only passing on messages that are really needed. A star network is also robust enough for applications such as online games, where clients constantly join and leave.

Apart from the star topology, multiplayer games follow an approach similar to the input-update-render loop that you are already familiar with from single-player games.

1. The client listens for player *input* and passes it up to the server.
2. The server *updates* the game state and generates global events.
3. The server broadcasts an update so that all clients know what to *render*.

You see that the three stages of input handling, updating, and rendering are still the same—only now they are distributed over the network.

Time for action – creating a headless server

For a network connection, you need one server and one client class. Both the client and server are based on the `com.jme3.app.SimpleApplication class`. Basically, we replace one `Main.java` with two files: `ClientMain.java` and `ServerMain.java`. You run one `ServerMain` for all players.

It takes only a few steps to create and run a server.

1. Make a copy of `Main.java` and name the class as `ServerMain.java`.
2. In the `main()` method, start the `SimpleApplication` class with the special `JmeContext.Type.Headless` constant as an argument.

```
public static void main(String[] args) {
  ServerMain app = new ServerMain();
  app.start(JmeContext.Type.Headless);
}
```

3. Create a `com.jme3.networking.Server` object. Initialize the server in `simpleInitApp()` to run on port 6143. The `createServer()` factory method throws an `IOException` that we need to try and catch.

```
private Server myServer;
public void simpleInitApp() {
  try {
myServer = Network.createServer(
           "My Cool Game", 1, 6143, 6143);
```

4. Still within the `try` block, we start the server.

```
    myServer.start();
    } catch (IOException ex) { }
}
```

5. Override the `destroy()` method of the `SimpleApplication` class to ensure that you close the network connection cleanly.

```
@Override
  public void destroy() {
    try {
      myServer.close();
    } catch (Exception ex) { }
    super.destroy();
  }
```

Run the code sample. When the server runs correctly, no errors appear on the terminal, no window opens, nothing visible happens—yet. Congrats, you have started your first jMonkeyEngine game server!

What just happened

The application that you just ran is a headless server. **Headless** means that this simple application does not open any window, it does not listen to user input, and it does not render any audio or video output. All this server application will do is maintain the game state, and run the main update loop.

When you create the server instance, you specify the game name (here `My Cool Game`), version (here `1`), and communication ports (here `6143`) in the factory method. In this example, we chose the port `6143` because it is the default for TCP and UDP network communication. You will learn more about network protocols later.

In general, you can choose any port number over 1024. We recommend you make a quick web search to confirm that your port is not yet taken by another very common service. More user-friendly applications let their users customize the port via a configuration file or a settings panel to avoid conflicts.

Time for action – client, meet server

Your game client is a normal instance of a `SimpleApplication`. That means clients are not headless. The game client renders video and audio, listens to input, and runs its own local update loop, just as you learned in the previous chapters. Each player runs their own instance of the `ClientMain` class.

For our client, we start a normal `SimpleApplication` with the `JmeContext.Type.Display` argument (which is also the default).

1. Make a copy of `Main.java` and name the class as `ClientMain.java`.
2. In the `main()` method, start the `SimpleApplication` with the `JmeContext.Type.Display` argument.

```
public static void main(String[] args) {
  ClientMain app = new ClientMain();
  app.start(JmeContext.Type.Display);
}
```

3. Create a `com.jme3.networking.Client` object with the same name, version, and port number as you specified for the server. The `connectToServer()` factory method throws an `IOException`; remember to try and catch it.

```
private Client myClient;
public void simpleInitApp() {
    try {
        myClient = Network.connectToServer(
                 "My Cool Game", 1, "localhost", 6143);
```

4. Still within the `try` block, start the client.

```
            myClient.start();
        } catch (IOException ex) { }
    }
```

5. Override the `destroy()` method of the `SimpleApplication` class to ensure that you close the connection cleanly.

```
    @Override
    public void destroy() {
        try {
            myClient.close();
        } catch (Exception ex) { }
        super.destroy();
    }
```

Let's test our two classes: run `ServerMain` first, and then `ClientMain`. The client and server connect on the same port. The client opens a jMonkeyEngine window. Apart from that, nothing visible happens yet.

If you get a `NoRouteToHostException` despite being online, your firewall is likely configured to block connections on the chosen communication port. Either reconfigure your server to communicate on an unblocked port, or change the firewall settings to open the port.

What just happened

When creating the client, you inform it of the server's game name (here `My Cool Game`), version (here the integer `1`), address (here `localhost`), and communication port (`6143`).

Note that if a client tries to connect to a server with a different name or version, the client closes and prints a `SEVERE` log message saying `Connection terminated, reason: Server client mismatch`.

In this example, we run both the client and server on localhost. It's okay to use localhost or an IP address during development or for testing—but for a real game, you will want to purchase a permanent domain name. Specify your server's hostname in place of localhost; for example, `mycoolgame.com`. Optimally, you read this hostname from a configuration file, so you don't have to recompile the game for every address change.

As we already mentioned for the server, always close network connections cleanly, at the latest when you quit, by overriding the server's and the client's `destroy()` methods. Make sure to call `myClient.close()` or `myServer.close()` respectively, as shown in the code sample, otherwise the network thread keeps running.

We recommend that you define global constants in one class that both the server and client can access. This is important for sharing the same version number, application name, ports, and network address, between the client and server builds.

Create a class named `Globals.java` with content similar to the following example:

```
public class Globals {
  public static final String NAME = "My Cool Game";
  public static final String DEFAULT_SERVER = "mycoolgame.com";
  public static final int VERSION = 1;
  public static final int DEFAULT_PORT = 6143;
  ...
}
```

Then, in your code, you can access these values with the `Globals.CONSTANT` syntax. For example, in the factory method, as follows:

```
myClient.connectToServer( Globals.NAME, Globals.VERSION,
Globals.DEFAULT_SERVER, Globals.DEFAULT_PORT);
```

Time for action – is anybody home?

You built and ran your server and a client. Assuming that your code did not throw any exceptions, the client has connected to the server. A bit of log output scrolls by, but apart from that, nothing visible happens. Is anybody home?

Let's quit both applications and have a peek at the server's connection status.

1. As a quick test, we add the following code to the update loop of `ServerMain.java`:

```
@Override
public void update() {
  System.out.println("Server connections: "
  + myServer.getConnections().size());
}
```

2. Start the server and one client again. You should see print lines saying `Server connections: 0` and `Server connections: 1`.
3. Start and quit some clients. Keep an eye on the numbers in the print lines.

You can now track that you have indeed established client-server connections with SpiderMonkey.

What just happened

The server's `getConnections().size()` method tells you the number of clients that are connected to the server. You can print this value as a quick and dirty test whether the server established the connection successfully. Later in the development process, after you have confirmed that the server recognizes the expected number of client connections, you remove this printline again.

Is there too much informational output scrolling past in your terminal? To be able to read our print lines, let's limit the logger to print warning messages only. Add the following line to the `main()` method:

```
java.util.logging.Logger.getLogger("").setLevel(Level.
WARNING);
```

`ClientListener` and `ServerListener` have access to a `source` object that identifies the sender of messages. Note that the `ServerListener` refers to clients as `HostedConnection`, while the `ClientListener` refers to itself as `Client`. The following information is available at runtime:

- `com.jme3.networking.Client`:
  - `isConnected()`: This returns a Boolean whether this client is ready.
  - `getID()`: This returns a unique positive number that identifies this client. The ID starts with `0` and increases for every connected client, until you restart the server.
- `com.jme3.networking.HostedConnection`:
  - `getServer()`: This returns the server instance that is hosting this connection.
  - `getAddress()`: This returns the server's IP address and port.
  - `getID()`: This returns a unique positive number that identifies this hosted connection.

- `setAttribute(s,o)` and `getAttribute(s)`: The server can maintain custom attributes for this `HostedConnection` session. An attribute is a string name, `s` and Java object, `o`.
- `attributeNames()`: This returns the current set of attribute strings for this `HostedConnection` (read-only).

- `com.jme3.networking.Server`:
  - `isRunning()`: This returns a Boolean whether this server is ready.
  - `getConnections()` and `getConnection(i)`: These return a `Collection` of all active `HostedConnections`, or one particular hosted connection identified by its ID `i`.

Time for action – please leave a message

Now that your client connects to the server, you want to send data from the client to the server, and from the server to the client. To send messages, we extend the `com.jme3.network.AbstractMessage` class, and to receive them, we implement the `com.jme3.network.MessageListener` interface.

The following example message sends a greeting as a string:

1. Create a class and name it `GreetingMessage.java`.

```
public static class GreetingMessage extends AbstractMessage {
```

2. Annotate the class as `com.jme3.network.serializing.Serializable`. Give each message class a unique ID (here we use `id=0` for the first message):

```
@Serializable(id=0)
```

3. Write getters and setters for the message. In this example, the data is simply a string.

```
  private String greeting = "Hello!"; // init your message data
  public  void   setGreeting(String s){greeting = s;}
  public  String getGreeting(){return greeting;}
```

4. Add an empty default constructor with no arguments. The empty constructor is required because we want this class to be `Serializable`.

```
  public GreetingMessage() {}  // empty default constructor
  public GreetingMessage(String s) { greeting = s; }
}
```

5. Register each message class to SpiderMonkey's `com.jme3.network.serializing.Serializer`. Add the following line to the `simpleInitApp()` methods of both `ServerMain.java` and `ClientMain.java`:

```
  public void simpleInitApp() {
  …
Serializer.registerClass(GreetingMessage.class);
  }
```

Serialization makes Java objects ready so that they can be sent across the network. Your first message class is now ready. You will create several different message classes for your actual game.

To be able to receive and respond to all types of messages, we now implement one message listener for the client, and one for the server.

1. Create a Java file, `ClientListener.java`, that implements the `com.jme3.network.MessageListener` interface.

```
public class ServerListener
implements MessageListener<HostedConnection> {}
```

2. Create a Java file, `ServerListener.java`, that implements the `com.jme3.network.MessageListener` interface.

```
public class ClientListener
implements MessageListener<Client> {}
```

3. Make the `ClientListener` print all `GreetingMessages` that it receives from the server. The `ClientListener` identifies itself to the `ServerListener` using `source.getId()`, which returns a unique number for each client.

```
public class ClientListener implements MessageListener<Client> {
  public void messageReceived(Client source, Message message) {
    if (message instanceof GreetingMessage) {
      GreetingMessage greetingMessage =
      (GreetingMessage) message;
      System.out.println("Client #"+source.getId()
        + " received the message: '"
        + greetingMessage.getGreeting() + "'");
    }
  }
}
```

4. Make the `ServerListener` print all `GreetingMessages` that it receives from clients. Additionally, the `ServerListener` identifies the sending client using `source.getId()`, changes the message, and returns it to the sender as confirmation.

```
public class ServerListener
implements MessageListener<HostedConnection> {
  public void messageReceived(HostedConnection source,
                              Message message) {
    if (message instanceof GreetingMessage) {
      GreetingMessage greetingMessage = (GreetingMessage) message;
      System.out.println("Server received '"
        + greetingMessage.getGreeting()
        + "' from client #" + source.getId() );
      // Send an answer
      greetingMessage.setGreeting("Welcome client #"
        + source.getId() + "!" );
      source.send(greetingMessage);
    }
  }
}
```

This demo shows how your client and server respond to messages that are instances of `GreetingMessage`, and how they call getters and setters on these messages.

Next, for each message class such as `GreetingMessage`, register the message listener to `ClientMain` and `ServerMain`.

1. Add the following line to the `simpleInitApp()` method of `ServerMain.java`:

```
public void simpleInitApp() {
    …
    myServer.addMessageListener(new ServerListener(),
    GreetingMessage.class);
}
```

2. Add the following line to the `simpleInitApp()` method of `ClientMain.java`:

```
public void simpleInitApp() {
    …
    myClient.addMessageListener(new ClientListener(),
    GreetingMessage.class);
}
```

Now you are ready to create and send your first message of type `GreetingMessage`.

1. Open `ClientMain.java`.
2. Create the message in the `simpleInitApp()` method.

```
public void simpleInitApp() {
    …
    Message message = new GreetingMessage("Hi server! "
    + "Do you hear me?");
```

3. Send the message from the client to the server:

```
    myClient.send(message);
}
```

4. Run the server, and then start two clients. Quit one client, and then the other. The output should look similar to the following:

```
Server connections: 0
Server connections: 1
Server received 'Hi server, do you hear me?' from client #0
Client #0 received the message: 'Welcome client #0!'
Server connections: 2
Server received 'Hi server, do you hear me?' from client #1
Client #1 received the message: 'Welcome client #1!'
Server connections: 1
Server connections: 0
```

This simple print line demo shows the gist of how the client and server exchange data—and how they can communicate about the game state.

The server can also send a message to all clients instead of just to one. Create your message as before, and broadcast it to all clients using the following line:

```
myServer.broadcast(message);
```

Use `com.jme3.network.Filters` to broadcast the message to an explicit list of clients, from `client1` to `client3`.

```
myServer.broadcast( Filters.in( client1, client2,
client3 ), message );
```

Broadcast to all clients except the listed client client4 by using:

```
myServer.broadcast( Filters.notEqualTo( client4 ),
message );
```

What just happened

Sooner or later, your client needs to inform the server of user input. Sooner or later, the server needs to inform the clients of global events, and also about what other clients are up to—it's multiplayer after all!

Data sent over the network is referred to as a message. In SpiderMonkey, your messages extend the `com.jme3.network.AbstractMessage` class. You also annotate each message as `@Serializable`, because we use standard Java serialization to package the data for sending. Specify a unique ID for each message to ensure that the application uses the correct message serializer, regardless of the order in which you register the messages.

```
@Serializable(id=0)
```

You will create individual message classes, for example, to transport a line of a chat conversation, to convey character navigation, to broadcast a game action and who triggered it, to notify the game of players joining or leaving, or whatever your particular game requires. You create one message class for each type of message that you want to send.

To be able to receive and respond to messages, you implement one message listener for the client, and one for the server. Extend these two listeners from the `com.jme3.network.MessageListener` interface. In the `Listener` classes, you implement how you want to respond when receiving a particular message. In this example, we printed text and sent a message back. In a real game, a message may cause a character to move, shoot, or otherwise interact with the world.

The last step of the process is creating and sending the message. You create an instance of the appropriate message class, and call the `send()` or `broadcast()` method on it.

Pop quiz – client and server

Which of the following is true?

1. A game client is a default `SimpleApplication` that renders a scene graph.
2. A game client is a headless `SimpleApplication` that renders a scene graph.
3. A game server is a default `SimpleApplication` that runs the main update loop.
4. A game server is a headless `SimpleApplication` that runs the main update loop.

Overview: client, server, and message

Here is a short overview of the client-server message creation process that you have just learned:

1. Create one server and one client class.
    1. Create a headless `SimpleApplication`, `ServerMain.java`, as the server. Specify this game's communication ports in the constructor.
    2. Create a standard `SimpleApplication`, `ClientMain.java`, as the client. Specify the server's IP address and communication ports in the constructor.
    3. For both the server and client, override the `SimpleApplication`'s `destroy()` method to `close()` the connections.
2. Prepare message types.
    1. For each message type, create a message class that extends `com.jme3.network.AbstractMessage`. Remember the `@Serializable` annotation and empty default constructor.
    2. For each message type, register the message class to the serializer in both the server and client using `registerClass()`.
3. Prepare responses to message types in the listeners.
    1. Create one `ClientListener.java` and one `ServerListener.java`. Make them extend `com.jme3.network.MessageListener`.
    2. For each message type, implement a response in both listeners' `messageReceived()` methods. Use `if(message instanceof SomeMessage)` to distinguish message types.
    3. For each message type, register a server listener to `ServerMain`, and a client listener to `ClientMain`, using `addMessageListener()`.
4. Write your game logic, and create and send messages.

Use multithreading, Luke!

You have learned to use client listeners and server listeners to specify how clients and the server respond after receiving a message. The listeners' `messageReceived()` methods are the places where you trigger game actions, such as attaching or detaching a spatial, changing a spatial's transformation, or changing an icon or text in the HUD. These actions modify the scene graph in the client.

The catch is that you mustn't modify the scene graph from the network thread. As you remember from *Chapter 3*, *Interacting with the User*, several threads access the game data in memory. You cannot jump in from another thread and change the scene graph when it's not your turn. If you want to modify the scene graph from the network thread, you must use standard Java concurrency.

Time for action – changing the scene graph

The simplest example of changing the scene graph of our networked game is to change the color of an attached cube every time a `CubeMessage` is received.

1. Attach a white cube to the scene graph in `ClientMain`.
2. Create a serializable `CubeMessage` with a getter that returns a `ColorRGBA` object.
3. Send an empty `CubeMessage` up to the server when the client connects. Add the following code to `ClientMain`:

```
public void clientConnected(Client client) {
  myClient.send( new CubeMessage() );
}
```

4. When the server receives a `CubeMessage`, it picks a color, and broadcasts the `CubeMessage` on to all clients. Add the following to the `ServerListener`:

```
public void messageReceived(HostedConnection source, Message
message) {
  if (message instanceof CubeMessage) {
    CubeMessage cubeMessage = (CubeMessage) message;
    /* tell all clients! */
    server.broadcast(new CubeMessage(ColorRGBA.randomColor()));
  }
}
```

5. For the `ClientListener` to be able to change the scene graph, it needs access to the `SimpleApplication`. Add the following constructor to the client listener:

```
private ClientMain app;
public ClientListener(ClientMain app) {
  this.app = app;
}
```

6. We use this new constructor in `ClientMain` (our `SimpleApplication`) when we add the message listeners for the `CubeMessage`. Add the following code to the `simpleInitApp()` method of `ClientMain`:

```
myClient.addMessageListener(
  new ClientListener(this),CubeMessage.class);
```

7. When a client receives a `CubeMessage`, it changes the color of the attached cube. Add the following code to the client listener:

```
public void messageReceived(Client source, Message message) {
  if (message instanceof CubeMessage) {
    final CubeMessage cubeMessage = (CubeMessage) message;
    app.enqueue(new Callable() {
      public Void call() {
        Material mat = new Material(app.getAssetManager(),
                          "Common/MatDefs/Misc/Unshaded.j3md");
        mat.setColor("Color", cubeMessage.getColor());
        app.getRootNode().getChild(0).setMaterial(mat);
        return null;
      }
    });
  }
}
```

Run `ServerMain` and `ClientMain` again. You should see a colored cube. Start another `ClientMain`—it shows the cube in a different color. Switch to the first `ClientMain` and note that the color of the first client's cube was updated.

What just happened?

In this simple demo, each client starts out with a `rootNode` that has a white cube attached.

Every time a client starts and connects, it sends a `CubeMessage` to the server. The server responds to the `CubeMessage` by broadcasting a new `CubeMessage` to all clients. In this demo, the `CubeMessage` transmits a specific `ColorRGBA` object to all clients.

Each client reacts to the `CubeMessage` broadcast by updating its scene graph. In this demo, the client simply changes a cube to the color contained in the message.

```
app.getRootNode().getChild(0).setMaterial(mat);
```

We use `getChild(0)` to find the cube because it is the only child of the `rootNode`—in a larger game, you would access the spatial by name.

When you find yourself writing anything such as `app.getRootNode().get…`, you are modifying the scene graph. Remember to wrap such calls into a `Callable()` and to enqueue them:

```
app.enqueue(new Callable() {
  public Void call() {
    …
    app.getRootNode().getChild(0).setMaterial(mat);
```

```
        return null;
      }
        });
```

Enqueuing the `Callable` ensures that the desired modification is performed in sync with other threads. Proper multithreading prevents hard-to-track glitches that may occur when your application grows more complex.

So many things I have to tell you

The default serializer can translate all Java data types (such as string and int) to bytes. In the previous examples, you used the default serializer, but you may want to transmit custom Java objects. To serialize custom data types, you write a custom serializer.

Time for action – serializing custom messages

When you write a custom serializer, you should take the opportunity and optimize the translation of the data so the message is as short as possible. The following example shows how you write a serializer for a `java.net.InetAddress` object.

1. We create a class named `InetAddressSerializer.java`. Make it extend the general serializer class.

```
public class InetAddressSerializer extends Serializer {
```

2. Implement the `Serializer` methods `readObject(ByteBuffer, Class<T>)` and `writeObject(ByteBuffer, Object)`. The most compact way to store an IP address is four bytes.

```
  @Override
  public <T> T readObject(ByteBuffer data, Class<T> c)
         throws IOException {
    byte[] address = new byte[4];
    data.get(address);
    return (T) InetAddress.getByAddress(address);
  }

  @Override
  public void writeObject(ByteBuffer buffer, Object object)
        throws IOException {
    InetAddress address = (InetAddress) object;
    buffer.put(address.getAddress());
  }
}
```

3. Go to `ServerMain.java` and register you custom `InetAddressSerializer` with the `InetAddress` class.

```
Serializer.registerClass(InetAddress.class,
  new InetAddressSerializer());
```

4. Let's create a message that uses the custom serializer. Create a serializable class, `InetAddressMessage.java`, with a unique ID:

```
@Serializable(id=1)
public class InetAddressMessage extends AbstractMessage {
  public InetAddress addr;
  public InetAddressMessage(InetAddress addr)
  { this.addr = addr; }
  public InetAddressMessage() { }
}
```

5. Remember to register the message class in `ServerMain.java`:

```
InetAddressSerializer.registerClass(InetAddressMessage.class);
myServer.addMessageListener(new ServerListener(),
  InetAddressMessage.class);
```

Now you can transmit an IP address in an efficient, compact message.

What just happened?

The process shown here is standard Java serialization to convert objects to bytes. Although reading and writing a serialized object is technically easy, you should find the most concise message format for a network application.

This means you should not serialize fields that merely store precalculated results or cache global constants. Instead you write `readObject()` so it restores these values by, for instance, calculating them again. This way you save several bytes per message, which is an important part of optimizing your networked game.

Time for action – responding to all kinds of messages

Look at your message listener code again. Your listeners currently expect only `GreetingMessages`. Now, it's time to change the code to distinguish between `GreetingMessage` and the newly added `InetAddressMessage`:

1. In the client listener and server listener, add an `else` clause to the condition of the `messageReceived()` methods.

```
if (message instanceof GreetingMessage) {
    …
} else if (message instanceof InetAddressMessage) {
```

```
        InetAddressMessage addrMessage =
        (InetAddressMessage) message;
        // do something, e.g. print
        System.out.println("The server received the IP address "
        + addrMessage.getAddress() + "from client #" +
        source.getId());
    }
```

2. Go to the end of the `simpleInitApp()` method of `ClientMain` and send a test message from the client to the server.

```
    try {
        Message message = new InetAddressMessage(
            InetAddress.getByName("jmonkeyengine.org"));
        myClient.send(message);
    } catch (UnknownHostException ex) { }
```

Quit `ServerMain` and `ClientMain`, and rerun them. The server responds to a new client by printing the IP address of `jmonkeyengine.org`.

What just happened?

You now know how to send and respond to several types of messages, and you know how to use serialization to transmit custom data efficiently.

Time for action – clients come and go

In a networked game, clients can connect or drop off any time. To be able to introduce itself to the server, your clients must implement the `com.jme3.network.ClientStateListener` interface.

1. Edit `ClientMain.java` and make the class implement the `ClientStateListener` interface:

```
public class ClientMain extends SimpleApplication
  implements ClientStateListener {
```

2. Implement the `clientConnected()` and `clientDisconnected()` methods. Here we define what the client does right after it connects or disconnects.

```
  public void clientConnected(Client client) {
    System.out.println("Client #" + client.getId()
    + " is ready.");
  }
  public void clientDisconnected(Client client,
```

```
    DisconnectInfo info) {
      System.out.println("Client #" + client.getId()
      + " has left.");
    }
…
}
```

3. Remember to register the `ClientStateListener` in the `simpleInitApp()` method of `ClientMain`:

```
myClient.addClientStateListener(this);
```

Similarly, the `com.jme3.network.ConnectionListener` notifies the server every time when a client has established a connection to the server.

1. Edit `ServerMain` and make the class implement the `ConnectionListener` interface:

```
public class ServerMain extends SimpleApplication
  implements ConnectionListener {
```

2. Implement the `connectionAdded()` and `connectionRemoved()` methods. Here we define what the server does right after a client connects or disconnects.

```
  public void connectionAdded(Server server,
  HostedConnection client) {
    System.out.println("Server knows that client #"
    + client.getId() + " is ready.");
  }

  public void connectionRemoved(Server server,
  HostedConnection client) {
    System.out.println("Server knows that client #"
    + client.getId() + " has left.");
  }
    ...
}
```

3. Remember to register the `ConnectionListener` in the `simpleInitApp()` method of `ServerMain`:

```
myServer.addConnectionListener(this);
```

Rerun `ServerMain` and `ClientMain`. When clients come and go, the client-server communication looks as follows:

```
Server connections: 0
Server connections: 1
Client #0 is ready.
Server knows that client #0 is ready.
Server received 'Hi server, do you hear me?' from client #0
Client #0 received the message: 'Welcome client #0!'
...
Server connections: 2
Client #1 is ready.
Server knows that client #1 is ready.
Server received 'Hi server, do you hear me?' from client #1
Client #1 received the message: 'Welcome client #1!'
...
Server connections: 1
Client #0 has left.
Server knows that client #0 has left.
...
Server connections: 0
Client #1 has left.
Server knows that client #1 has left.
```

This example simply prints out status messages, so you see that the methods were indeed triggered.

What just happened?

The `com.jme3.network.ClientStateListener` is notified as soon as the client has connected to the server, and again when the client disconnects. The `connectionAdded()` and `connectionRemoved()` methods are where you define what the client should do after it connects or disconnects. If your game requires a login, `connectionAdded()` is the method where the client sends the username and password to the server. Connecting may take a few seconds, so the server is interested in knowing exactly when the connection is ready.

On the server, a similar `com.jme3.network.ConnectionListener` fires whenever a client comes or goes. You write the `clientConnected()` and `clientDisconnected()` methods to define how the server reacts when a client connects or disconnects. This is the method where you attach players to the game world, or detach them, respectively.

Time for action – fast or reliable, pick one

To decide whether you should send a message reliably or not, look at a series of messages of this type, and ask yourself the following questions:

- Does the next message of this type make the previous one obsolete? Can the game recover if one of these messages is lost? For example, the world state and player states are constantly updated to absolute values: you constantly broadcast new absolute locations, directions, and so on. If a character misses one step, the worst that happens is a little jerk in the moment when it catches up with the next location message. In these cases, you can send unreliable messages using UDP.
- Does the meaning of these messages change if they arrive in a different order? Do the messages convey unique events? For example, whether a player wielded a weapon before or after attacking makes all the difference between killing the dragon and enraging it with a shovel. Whether a player casts a protective spell on himself/herself before or after the bat bites him/her makes all the difference between a career as a paladin or vampire. When players trade, they do not accept that the cheque was lost in the mail. In these cases, always send reliable messages using TCP.

You switch between UDP or TCP by setting the reliable flag on the message.

```
message1.setReliable(true);  // reliable, slow (TCP)
message2.setReliable(false); // unreliable, fast (UDP)
```

By default, all messages are sent reliably (true).

What just happened?

The SpiderMonkey API offers both TCP and UDP transport protocols. The distinction is important, as the use of a protocol depends on the message's use case.

Protocol	Pros	Cons
UDP	This is fast.	This is unreliable. Messages can get lost, be duplicated, or can arrive out of order.
TCP	This is reliable. Messages arrive in the correct order. Each message arrives only once. Lost messages are resent.	This is slow. One lost message stalls the whole channel.

The specified port is used for both TCP and UDP communication, but if desired, you can also specify two different ports in the client and server constructors. If you set the server's UDP port to `-1`, the server routes all traffic over TCP. You can switch to TCP if you know that your players have fast network connections. TCP communication is more succinct, because it does not flood the network with packets.

Pop quiz – listeners

Which of the following is true?

1. You write one `Message` per client listener.
2. You write one `ServerListener` per game.
3. You write one `ClientMain` per player.
4. You write one `ClientListener` per message.

Summary

You have completed the core of the beginner's introduction to the jMonkeyEngine API. In this last development chapter, you learned how to share a game world with other players.

You sent and received messages, you devised more efficient messages, and you chose between fast versus reliable messages. Combining that with the previous chapters, you now know everything that you need to start writing your first 3D video game. If you have open questions about best practices, check the appendix for extra networking tips.