DOF Connectivity

Programmer's Guide

Version 7.0

Table of Contents

Chapter 1: Introduction

Welcome to the *DOF Connectivity Programmer's Guide*. This guide contains the basic knowledge system developers need to use version 6.x of the DOF Object Access Libraries (OALs) to connect DOF nodes on a network.

The DOF application programming interface (API) currently includes versions of the OAL in Java, C, and C#. Although this guide provides basic information for all three languages, DOF is an object-oriented technology at its core, and it is most specifically modeled on Java language conventions. If you are unfamiliar with object-oriented programming in general or with specific Java definitions of terms such as "field" or "interface," you may want to familiarize yourself with these before working with any of the OALs. Even if you are not working in Java, a basic understanding of Java will help you to better understand DOF.

Since the Java and C# OALs are virtually identical, we have omitted C# sample code from this guide. You may assume that unless otherwise stated, samples in C# differ from the Java samples only in the conventional ways in which C# differs from Java, such as using an override "modifier" rather than an override "annotation."

How to Read This Guide

To get the most from this guide, you should read each chapter in order through to the end.

This guide can be used in conjunction with the Connectivity Training in the DOF Essentials SDK. This training packages the full source code shown in this guide and may be copied into your own development environment and used for practice.

Components of DOF Connectivity

To understand how to connect nodes in an DOF network, you must first understand the API components that form the basics of DOF connectivity. These API components, which are defined in this chapter, are as follows:

- DOF
- DOFSystem
- DOFConnection
- DOFServer

DOF

DOF (which stands for Distributed Object Framework) is the basic naming prefix used throughout the libraries. The DOF class itself represents an DOF network node, and every node on an DOF network needs an instance of DOF. The DOF instance is used to create the other connectivity components, and it manages the routing of network traffic between the components that it creates.

DOFSystem

A DOFSystem instance contains the application logic that initiates or responds to operations. It sends operations and responses to the DOF. Network traffic the DOF receives from an attached system are routed to all the following:

- Other instances of DOFSystem attached to the DOF
- Instances of DOFConnection attached to the DOF
- Connections accepted by instances of DOFServer attached to the DOF

DOFConnection and DOFServer

A DOFConnection instance can connect to a DOFServer instance on another node, based on the transport address (See "Transports" on page 4). Unicast connections only connect to a single server; however, unicast servers can accept multiple separate connections.

Once communication is successfully established between a connection and server, that communication is bidirectional. The DOF controls routing to the connections that it created (outbound connections) and to the connections accepted by the servers that it created (inbound connections). When a connection (regardless of direction) receives traffic from the DOF, it sends that traffic across the network to the other end of the connection.

By default, when a DOF receives traffic from a connection, it routes that traffic only to instances of DOFSystem it created. However, if the DOF is set to be a proxy node, it routes traffic to the other connections as well. This includes both outbound and inbound connections.

Transports

One of the advantages of DOF is that it is transport-independent. Achieving transport-indendence for DOF is a result of isolating DOF protocols from transport protocols, relying on two basic abstractions. This guide will use the typical DOF implementation of a transport for IP, and you will not need to implement your own transport; however, it is useful to have a basic understanding of the transport abstractions that DOF relies on.

Network Address Abstraction

The first abstraction is the network address. The DOF libraries include a DOFAddress API, which is an abstract representation of an address on the underlying transport layer. The OALs do not define how the DOFAddress should be created or handled, but routing relies on the DOFAddress instance. The transport implementation defines the methods for creating a DOFAddress instance and takes care of routing at the transport level.

Creating IPv4 Addresses

The standard transport implementation for IP (called the Inet Transport) offers the following method signatures for creating IPv4 addresses.

Using Java

```
DOFAddress InetTransport.createAddress(String host, int port)
```

Using C

```
DOFAddress InetTransport_CreateIPv4Address( const uint8 addr[4], uint16 port )
```

Creating a Connection to the IP localhost

The examples show how to create a DOFAddress instance that you could use to create a connection to the IP localhost address.

Using Java

```
DOFAddress address;

address = InetTransport.createAddress("127.0.0.1", InetTransport.DEFAULT_UNICAST_
         PORT);
```

Using C

```
uint8 sockAddress[4] = {127,0,0,1};
DOFAddress address;

address = InetTransport_CreateIPv4Address(sockAddress, INETTRANSPORT_DEFAULT_
         UNICAST_PORT);

...

DOFAddress_Destroy(address);
```

Note: The constants shown for the port in the createAddress methods are defined by the Inet Transport libraries and represent port 3567, which is the default port for DOF unicast traffic. This port is registered with the Internet Assigned Numbers Authority (IANA).

This guide will use the address variable shown above to create connections and servers. Since this address will only work for communications between applications that are on the same computer, to experiment with communications across your network, you will need to replace the localhost address with a) the IP address of the node itself when creating DOFServer instances or b) the IP address of the node you want to connect to when creating DOFConnection instances.

Connection Type Abstraction

The second abstract concept you will deal with for connections is the connection type. DOF categorizes connections into streaming connections or datagram connections and loosely defines a streaming connection as a lossless connection that delivers packets in order, while a datagram connection doesn't guarantee packet order and is potentially lossy.

It is up to the transport implementation to create a relationship between the DOF connection types and the transport's connection types. Some transports may only support one of the types. The Inet transport implementation supports both. If you create a streaming connection in DOF using this transport, it creates a TCP/IP connection. A DOF datagram connection creates a UDP/IP connection at the transport layer. (The DOF connection types are enumerated and will be discussed later in this guide.)

Note: The DOF OALs define a few other connection types, but understanding the streaming and datagram types provides a basic introduction to the Inet transport.

DOF Routing

In addition to understanding the basic flow of DOF routing, you should also have a basic understanding of two DOF concepts that change the basic routing tables: security domains and interest operations.

While the discussion of DOF security is outside the scope of this guide, you need to know that the DOF will only route traffic between components with compatible security. However, because all components we will create in this guide will be in the unsecured domain, they will be able to communicate freely with one another as long as they are on the same DOF or a network path is provided through DOFConnections and DOFServers.

Choosing when nodes should perform *interest* operations, and at what level, can be an important part of DOF system design. Therefore, it is important to understand that *interest* operations and matching *provide* operations form routing tables at the DOF level. Each DOF stores state that is related to these operations, so that subsequent traffic between *interested* nodes and *providing* nodes can be routed directly to the proper connectivity components. It is for this reason that persistent operations, such as *subscribe* and *register*, require that *interest* first be performed.

Operations that do not require *interest*, such as *get* and *set* operations, are flooded to every node on the network (each DOF routes them to all security-compatible components) if *interest* is not already established. *Interest* operations themselves are always flooded throughout the network, subject to security rules. If an *interest* operation has been performed, operations such as *get* and *set* are routed directly between interested and providing nodes. It is bandwidth considerations such as these that make *interest* operations an important part of system design, but it also helps to be aware of this as you learn about connectivity and experiment with sending operations over your network.

Working in Programming Teams

One of the features of the DOF OALs is that they simplify the division of labor across teams of programmers. It is assumed that operations programming may be performed by application developers, while connectivity and security programming may be performed by system developers. Since this guide was written for system developers, the following procedures are not discussed in this guide and must be managed in the applications:

- Application developers need a way to obtain instances of DOFSystem that you configure. Your code should provide a way for application developers to get DOFSystem instances. In most cases, application developers probably do not need instances of the other components of DOF connectivity.
- You must carefully manage memory deallocation within a team, especially if you are programming in C. All components of DOF connectivity must be destroyed when you are finished with them (and in a logical order). In the Java OAL, it should be sufficient to provide a shutdown routine that calls the DOF.destroy method. The Java OAL destroys all components associated with a DOF when it is destroyed. However, in the C OAL, all components must be destroyed manually. Between you and the application developer, you must ensure that the following are called in the proper order (see the API reference documentation for more information about each function call):
    - *DOF_SetNodeDown*
    - *DOFSystem_Destroy* for each DOFSystem instance
    - *DOFServer_Destroy* for each DOFServer instance
    - *DOFConnection_Destroy* for each DOFConnection instance
    - *DOF_Destroy*
- In addition, in the C OAL, DOF_Init must be called before you can use the library, and *DOF_Shutdown* must be called when you are finished with it. Usually, the simplest place to include these calls are at the beginning and end of main functions.

Chapter 2: Using the Builder Pattern

All DOF connectivity components are instantiated using a builder pattern. The builder pattern provides a simple way to construct objects with multiple configuration options and includes the following advantages:

- Avoiding too many constructors or too many constructor parameters
- Enabling validation of an object’s state prior to creation
- Enabling the resultant object to be immutable, and thus, inherently threadsafe and simpler to use and test
- Providing backward-compatibility in future versions of the library, because additional methods can be added to the builder to support new features, but old code simply won’t call them

Using the builder pattern is a simple four-step process:

1. Use the builder class’s *constructor* or *create* method to create a builder object.
2. Use the builder class’s *set* methods to alter specific configuration parameters in the builder object. All parameters have default settings so you can ignore any settings you do not need to change.
3. Use the builder class’s *build* method, which returns a configuration object with all the parameters you’ve chosen.
4. Pass the configuration object to the component’s *constructor* or *create* method.

The next chapter demonstrates how to use this pattern to create the components. The builder pattern is not unique to the DOF OALs, so if you need further help understanding this pattern or the samples in the following chapters, you can research it online.

Chapter 3: Creating Connectivity Components

In this chapter, you will learn how to create instances of the four connectivity components that were discussed earlier:

- DOF
- DOFSystem
- DOFServer
- DOFConnection

This chapter also discusses considerations for managing the division of labor between system developers (who would normally create the components in this chapter) and application developers.

DOF

A DOF instance is always the first thing your program must create, because it represents a node in an DOF network and must be used to create the other components. No OAL application can be DOF-enabled without an instance of DOF. The sections that follow show you how to create an instance of DOF that uses a default configuration and how to use the configuration builder to change one of the configuration options.

Building a DOF with a Default Configuration

The constructor for a DOF in the Java OAL and the DOF_Create function in the C OAL both take an instance of *DOF.Config* (representing the configuration) as their only input parameter. To create a DOF with a default configuration, you can pass a null value for this argument as shown below.

Using the Java OAL

```
DOF myDOF = new DOF(null);
```

Using the C OAL

```
DOF myDOF = DOF_Create(NULL);
```

Using the DOF Configuration Builder

The DOF has a number of configuration options. (For more information, see the API reference documentation.) To demonstrate how to use the DOF configuration builder, the following sections show you how to set the maximum number of simultaneous connections that a DOF can support.

In the default configuration, this parameter is set to its maximum value so that the maximum number of connections is limited only by the operating system. In our example, we'll set this to 5, which includes both outbound and inbound connections. This means that if the node has both an outbound connection that is always connected along with a server, the server will only be able to accept four connections at once.

DOF Configuration Builder Sample Code

The examples below show how to set the maximum connections to 5 using the Java and C OAL.

Using the Java OAL

```
DOF myDOF;
DOF.Config myDOFConfig;
myDOFConfig = new DOF.Config.Builder()
   .setMaxConnections((short)5)
   .build();
myDOF = new DOF(myDOFConfig);
```

Using the C OAL

```
DOF myDOF;
DOFConfig myDOFConfig;
DOFConfigBuilder myDOFConfigBuilder;

myDOFConfigBuilder = DOFConfigBuilder_Create();
DOFConfigBuilder_SetMaxConnections(myDOFConfigBuilder, 5);
myDOFConfig = DOFConfigBuilder_BuildAndDestroy(myDOFConfigBuilder);

myDOF = DOF_Create(myDOFConfig);
DOFConfig_Destroy(myDOFConfig);
```

Understanding the Sample Code

The sample code follows the following steps as outlined in the previous chapter:

1. We used the *DOF.Config.Builder* class constructor in the Java OAL and the *DOFConfigBuilder_Create* function in the C OAL to create an instance of the builder.
2. We used *the DOF.Config.Builder.setMaxConnections* method in Java and the *DOFConfigBuilder_SetMaxConnections* function in C to set the maximum connections to five. Setter methods in the Java OAL typically take only the value for the option you wish to set. In the C OAL, the first argument is always the instance of the builder.
3. We used the *DOF.Config.Builder.build* method in Java and the *DOFConfigBuilder_BuildAndDestroy* function in C to create an instance of a DOF configuration with options matching those we set in the builder. The C function simultaneously builds the DOFConfig instance and destroys the DOFConfigBuilder instance. The C OAL also has a *DOFConfigBuilder_Build* function. If you use this function, you must

manually destroy the builder. (Java garbage collection takes care of the builder instance in Java.)

4. We passed the configuration object to the DOF constructor in Java and the *DOF_Create* function in C. In C, you are then free to also destroy the DOFConfig instance.

DOFSystem

With the exception of naming a DOFSystem instance for logging purposes, which is an option only in the Java OAL, all the configuration options for DOFSystem are related to security. Because of this, this guide will show only the shortcuts for creating a system that uses the default configuration.

Using the Java OAL

The shortcut in the Java OAL is a method call that takes no arguments:

```
DOFSystem myDOFSystem = myDOF.createSystem();
```

Using the C OAL

Because the C programming language does not support function overloading, the *DOF_CreateSystem* function in the C OAL has the following input parameters:

- **DOF.** Pass the instance of DOF you created.
- **DOFSystemConfig.** To create a system with the default configuration, pass NULL.
- **Timeout.** A timeout is required when creating a secure system so that the system’s credentials can be validated by an Authentication Server. For an unsecured system, pass "zero."
- **Exception.** This argument is a pointer to an exception you declare. In this guide, we will pass NULL, because exceptions to this function are security-related. For more information look at the sample code for creating DOFServer and DOFConnection instances.

You can use the following to create a DOFSystem instance using the default configuration in the C OAL:

```
DOFSystem myDOFSystem = DOF_CreateSystem(myDOF, NULL, 0, NULL);
```

DOFServer

There is no shortcut to instantiate DOFServer without using a configuration builder. This is because there is no reliable default for the server type and address; thus, they must always be specified. In addition to being created, servers must be started before they will operate properly.

DOFServer Sample Code

The examples below show how to create and start a simple server without setting any configuration parameters other than the server type and address.

Using the Java OAL

```
DOFServer.Config myServerConfig;
myServerConfig = new DOFServer.Config.Builder(DOFServer.Type.STREAM, address)
   .build();
myServer = myDOF.createServer(myServerConfig);

try {
   myServer.start(TIMEOUT);
} catch (DOFException e) {
   /* Handle exception. */
}
```

Using the C OAL

```
DOFException ex;
DOFServerConfig myServerConfig;
DOFServerConfigBuilder myServerConfigBuilder;

myServerConfigBuilder = DOFServerConfigBuilder_Create (DOFSERVERTYPE_STREAM,
         address);
myServerConfig = DOFServerConfigBuilder_BuildAndDestroy (myServerConfigBuilder);
myServer = DOF_CreateServer(myDOF, myServerConfig);

DOFServerConfig_Destroy(myServerConfig);

DOFServer_Start(myServer, TIMEOUT, &ex);

if(ex){
   /* Handle exception. */
   DOFException_Destroy(ex);
}
```

Understanding the Sample Code

Note the following:

- We created a streaming server by using an enumeration in the library. You must pass one of these enumerated values for this argument, and there is also an enumeration for datagram servers. For a server to accept a connection, they must be of the same type; therefore, if you know that your node may accept both datagram and streaming connections, you must make two servers: one of each type.
- The address argument passed to the builder is the DOFAddress instance created in the introduction.
- The *start* methods take a timeout argument in milliseconds, which represents how long you are willing to have your program wait for the server to start. The TIMEOUT constant shown in the samples is not defined by the OALs. You must define your own waiting period. Because the *start* method is a synchronous call, you want to discover a good compromise based on your processing power, judging between giving the server enough time to start and blocking your thread. However, unlike the methods for connecting instances of DOFConnection that we will

examine in the next section, starting a server does not typically involve network traffic. If your start method is timing out, the exception will indicate the problem.
- The methods for starting the server may throw exceptions.
    - In Java, you must enclose this method in a try-catch block.
    - In C, the *DOFServer_Start* function returns a boolean, which is false if the server fails to start. The function also takes an exception pointer argument, so you can retrieve information about the error. This is the typical exception handling model in the C OAL.

DOFConnection

Creating a DOFConnection and connecting to a server is nearly identical to creating and starting a server. Sample code for this is shown below.

Note: The connection type must match the server type, and the transport address stored in the DOFAddress instances must also match.

The examples below show how to create and connect a simple connection without setting any configuration parameters other than the type and address.

Using the Java OAL

```
DOFConnection.Config myConnectionConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, address)
   .build();
myConnection = myDOF.createConnection(myConnectionConfig);

try {
   myConnection.connect(TIMEOUT);
} catch (DOFException e) {
   /* Handle exception. */
}
```

Using the C OAL

```
DOFException ex;
DOFConnectionConfig myConnConfig;
DOFConnectionConfigBuilder myConnConfigBuilder;

myConnConfigBuilder = DOFConnectionConfigBuilder_Create (DOFCONNECTIONTYPE_
         STREAM, address);
myConnConfig = DOFConnectionConfigBuilder_BuildAndDestroy (myConnConfigBuilder);
myConnection = DOF_CreateConnection(myDOF, myConnConfig);

DOFConnectionConfig_Destroy(myConnConfig);

DOFConnection_Connect(myConnection, TIMEOUT, &ex);

if(ex){
   /* Handle exception. */
   DOFException_Destroy(ex);
}
```

Chapter 4: Creating Proxies

In DOF topology, a proxy node is an intermediate node between two endpoints that desire to communicate. The proxy is capable of routing operations between the endpoints. Proxy nodes have security and permissions requirements in a secure system; however, creating a proxy in the unsecured domain is relatively simple.

When designing your topology, you must ensure that the proxy can communicate with both endpoint nodes. For example, the proxy might have a DOFServer instance, and both endpoint nodes connect to it. Another common model (especially for gateways) is for the proxy to have a DOFServer that multiple endpoints connect to, while the proxy also maintains a connection to an DOF Center Server. DOF provides flexibility for you to design your topology using multiple connection and server models.

If a proxy node does not itself provide any DOF interfaces, it may not need any instances of DOFSystem.

To create a proxy in an unsecured system, you need only create a DOF that has its router parameter set to "true." This parameter is "false" by default, so you must use the builder to create a DOF for a proxy node.

The following examples show how to create a proxy node.

Using the Java OAL

```
DOF myDOF;
DOF.Config myDOFConfig;
myDOFConfig = new DOF.Config.Builder()
   .setRouter(true)
   .build();
myDOF = new DOF(myDOFConfig);
```

Using the C OAL

```
static DOF myDOF;
DOFConfig myDOFConfig;
DOFConfigBuilder myDOFConfigBuilder;

myDOFConfigBuilder = DOFConfigBuilder_Create();
DOFConfigBuilder_SetRouter(myDOFConfigBuilder, TRUE);
myDOFConfig = DOFConfigBuilder_BuildAndDestroy(myDOFConfigBuilder);

myDOF = DOF_Create(myDOFConfig);
DOFConfig_Destroy(myDOFConfig);
```

Chapter 5: Using State Listeners

In section on creating DOFServer and DOFConnection instances, we used synchronous *start* and *connect* calls. However, the library also provides non-blocking asynchronous calls that you can use for this purpose.

You can also use connection and server state listeners to monitor the status of your components. These allow you to attempt to restart the server or reconnect the connection if they go down.

Note: This chapter shows sample code only for the connection state listener; however, the server state listener is similar.

State Listener Callbacks

The connection state listener is an interface with the following callbacks:

- **stateChanged.** This is called when the connected status of a connection changes. The connected status is represented by the isConnected Boolean in the DOFConnection.State. When a connection is connected, the isConnected Boolean is "true." When a connection becomes disconnected, the isConnected Boolean is "false."
- **removed.** The sample shows these methods as "empty," but they can be used to clean up resources associated with a state listener.

The samples also implement the connection operation listener interface, which has the following callback:

- **complete.** This is called whenever an attempt to connect has finished. It is called regardless of whether or not the attempt to connect was successful.

State Listener Implementation

When creating a connection and implementing the *DOFConnection.StateListener* and the *DOFConnection.ConnectOperationListener* interfaces, note the following:

- *DOFConnection.addStateListener* is used to associate the *StateListener* with the connection.
- The *ConnectOperationListener* is associated with the connect operation because it is passed as a parameter in the *beginConnect* method.

State Listener Sample Code

The following examples show how to create a connection and implement the DOFConnection.StateListener and the DOFConnection.ConnectOperationListener interfaces.

Using the Java OAL

```
DOFConnection.Config myConnectionConfig = new DOFConnection.Config.Builder
         (DOFConnection.Type.STREAM, address)
   .build();
myConnection = myDOF.createConnection(myConnectionConfig);

ConnectionListener connectionOperationListener = new ConnectionListener();

myConnection.addStateListener(new ConnectionStateListener());

private class ConnectionStateListener implements DOFConnection.StateListener {
   @Override
   public void stateChanged(DOFConnection connection, DOFConnection.State state)
            {
      while(connection.isConnected() == false){
         myConnection.beginConnect(TIMEOUT, connectionOperationListener, null);
      }
   }

   @Override
   public void removed(DOFConnection arg0, DOFException arg1) {
   }
}

private class ConnectionListener implements
         DOFConnection.ConnectOperationListener {
   @Override
   public void complete(DOFOperation operation, DOFException ex) {
      if(ex == null){
         DOFOperation.Connect connectionOp = (DOFOperation.Connect) operation;
         DOFConnection connection = connectionOp.getConnection();

         /* The following method, which is not shown in the sample, would
         * pause between connection attempts.
         */
         delayReconnection();
         if(!connection.isConnected()){
            connection.beginConnect(TIMEOUT, connectionOperationListener, null);
         }
      }
   }
}
```

Using the C OAL

```
DOFConnectionConfig myConnectionConfig;
DOFConnectionConfigBuilder myConnConfigBuilder;

static void removed (DOFConnectionStateListener self, DOFConnection connection,
         DOFException except);
static void stateChanged(DOFConnectionStateListener self, DOFConnection
         connection, DOFConnectionState state);
static const struct DOFConnectionStateListenerFns_t ConnectionStateListenerFns =
         { removed, stateChanged };
static DOFConnectionStateListener_t ConnectionStateListener = {
         &ConnectionStateListenerFns };
```

```
static void connectionComplete(DOFConnectionConnectCallback self, DOFConnection
         connection, void *context, DOFException except);
static const struct DOFConnectionConnectCallbackFns_t connectionCallbackFns = {
         connectionComplete };
static DOFConnectionConnectCallback_t connectionCallback = {
         &connectionCallbackFns };

myConnConfigBuilder = DOFConnectionConfigBuilder_Create (DOFCONNECTIONTYPE_
         STREAM, address);
myConnectionConfig = DOFConnectionConfigBuilder_BuildAndDestroy
         (myConnConfigBuilder);
myConnection = DOF_CreateConnection(myDOF, myConnectionConfig);

DOFConnection_AddStateListener(myConnection, &ConnectionStateListener);

DOFConnectionConfig_Destroy(myConnectionConfig);
DOFAddress_Destroy(address);

static void removed (DOFConnectionStateListener self, DOFConnection connection,
         DOFException except){
}

static void stateChanged(DOFConnectionStateListener self, DOFConnection
         connection, DOFConnectionState state){
   boolean connectionStatus = DOFConnectionState_IsConnected(state);

   if(connectionStatus == FALSE){
      DOFConnection_BeginConnect(connection, TIMEOUT, &connectionCallback, NULL);
   }
}

static void connectionComplete(DOFConnectionConnectCallback self, DOFConnection
         connection, void *context, DOFException except){
   boolean connectionStatus = DOFConnection_IsConnected(connection);

   if(connectionStatus == FALSE){
      /* In the C OAL, while it is still important to pause between
      * connection attempts, it is equally important not to pause
      * callback threads. Therefore, the following function spawns a new
      * thread, pauses for the length of time in the timout, and then
      * calls DOFConnection_BeginConnect(connection, TIMEOUT,
      * &connectionCallback, NULL) again.
      * /
      delayReconnection (TIMEOUT, connection);
   }
}
```

Understanding the Sample Code

Notice that the code does not attempt to connect the connection when it is initially instantiated. When the state listener is added, it prompts a call to the *stateChanged* method and all attempts to connect take place in the state listener.

The last parameter in the beginConnect calls allows you to pass a context.

The code follows this sequence of calls:

1. The connection is instantiated, prompting a call to the *stateChanged* callback.
2. *DOFConnection.beginConnect* is called in the *stateChanged* callback, causing the complete callback in the connection operation listener to be called, regardless of whether the connection successfully connected or not.
3. If the connection fails to connect, a pause is implemented to prevent churning, especially because *beginConnect* is an asynchronous call. After the pause, *beginConnect* is called again, using the same connect operation listener.
4. This cycle repeats until the connection is successful. After the connection is successful, both *complete* and *stateChanged* are called. However, at this point the *isConnected* state of the connection is "true," so the threads reach a stopping condition.
5. If the connection is disconnected, *stateChanged* is called again and the cycle repeats.

Chapter 6: Conclusion

After reading this guide, you should have the basic knowledge required to create components of DOF connectivity and connect a simple DOF network.

Future versions of this guide will contain more information on connecting multicast groups.