ADOBE® INDESIGN®

ADOBE INDESIGN SERVER SOLUTIONS

Adobe

Adobe® InDesign® Server Solutions

Chapter Update Status		
CS6	Edited	Removed chapters related to Java and CORBA. Other chapters as noted.

Contents

Introduction 4

Before you begin 4

Terminology 4

2 Converting Java Code to Scripts 6

Overview 6

Basic differences between Java and JavaScript 7

Converting Java to JavaScript 7

Differences between Java API and InDesign Scripting API 10

Working with InDesign Server SOAP 15

Terminology 15

InDesign Server WSDL (Web Services Description Language) 16

The InDesign Server SOAP method 18

InDesign Server SOAP messages and responses 19

SOAP client implementations 25

Debugging tips 27

Frequently asked questions 27

References 28

4 Scalability and Performance 30

Defining scalability and performance 30

Guilt by elimination 31

CPU bound or I/O bound? 32

What to do if you are I/O bound 36

What to do if you are CPU bound 38

InDesign Performance Metrics API 40

Performance-testing tools 40

Benchmark Charts Console 47

Introduction

Chapter Update Status		
CS6	Edited	Removed content related to Java and CORBA. Updated terminology. Other content not guaranteed to be current.

This document offers step-by-step instructions to developers embarking on Adobe® InDesign® Server development tasks. It includes references to other SDK documentation, tools, and samples, and it helps developers determine which InDesign API to use for different tasks.

This document includes:

- Chapter 2, "Converting Java Code to Scripts," which compares Java to JavaScript and also shows how to convert many Java code snippets to JavaScript (or other scripting languages).
- Chapter 3, "Working with InDesign Server SOAP," which describes the SOAP implementation for InDesign Server (the *RunScript, BeginSession, and EndSession* methods).
- Chapter 4, "Scalability and Performance," which provides information about tools you can use to quantify your system's performance, explains how to interpret the results, gives advice on how to improve your configuration to get the best performance from InDesign Server, and includes benchmarking tools.

Before the information in this document is of much value, you will need to become familiar with the basic architecture and general concepts of programming for InDesign Server. If you are new to InDesign Server development, we recommend that you start with *Introduction to Adobe InDesign Server Development,* which provides an overview of the SDK.

Before you begin

Before you begin, you need to make sure that your development environment and documentation are available.

Install the InDesign Server SDK. In this document, the directory where you install the SDK is referred to as *<IDS SDK>*. It is fine to move this directory after setting up.

Terminology

The following terms and acronyms are used throughout this document:

- *API*—Application programming interface.
- *CORBA*—Common Object Request Broker Architecture, a language-independent, distributed object model. CORBA is an industry standard for enabling computer languages to speak to one other.
- *IDS*—InDesign Server.

- <*IDS*>—The directory where you installed InDesign Server.
- <*IDS SDK*> — The directory where you installed the InDesign Server SDK.
- *InDesign plug-In SDK* — The C++ SDK for InDesign, Adobe InCopy®, and Adobe InDesign Server plug-in development.
- *InDesign scripting DOM*—Scripting document object model. This refers to the objects within InDesign and their hierarchy, as accessed through scripting.
- *InDesign scripting SDK*—The API for accessing the scripting DOM.
- *InDesign Server SDK* —Includes various technologies for working with InDesign Server, such as SOAP, examples of scripting and SOAP for use with InDesign Server, and InDesign Server documentation.
- <*Scripting SDK*>—The directory where you installed the scripting SDK.
- <*SDK*>—The directory where you installed the InDesign plug-in SDK.

2 Converting Java Code to Scripts

Chapter Update Status		
CS6	New	All new content.

Earlier versions of InDesign Server supported Java development. With InDesign Server CS6, Java is no longer supported, so you must convert your existing InDesign Java applications to another language.

This chapter briefly describes your options for alternatives to Java, discusses some of the similarities and differences between Java and the available scripting languages, and provides some guidance in porting Java applications to JavaScript scripts.

Overview

InDesign Server supports three languages for scripting:

- AppleScript for Mac OS
- VBScript for Windows
- JavaScript for either platform.

Although the scripting languages differ, the ways they work with InDesign Server are very similar. Both Java and InDesign Server's scripting solutions provide a way to access InDesign' Server's scripting document object model (DOM), which in turn is a hierarchy that parallels InDesign's object model.

Which scripting language should you use?

If you have written scripts before, use whatever language you know. If you have never written scripts before or if you need to make your scripts work on both the Mac OS and Windows versions of InDesign Server, use JavaScript. If you need to communicate with other, non-Adobe applications on your system, use the appropriate platform-standard language (AppleScript on Mac OS or VBScript in Windows).

Basic differences between Java and JavaScript

Java	JavaScript
An object oriented programming language	A scripting language
Applications run in a virtual machine or browser.	Runs in a browser only.
Java code needs to be compiled.	JavaScript code is all text.
Has static typing and strong type checking.	Is loosely typed; that is, you can set a variable to any type of data and the variable's type is interpreted based on context.
Has a compile-time system of classes built by declarations.	Supports a run-time system based on a small number of data types representing numeric, Boolean, and string values.

Converting Java to JavaScript

As can be seen from the samples in this section, there is one-to-one correspondence between Java APIs and JavaScript methods and properties. This is because both essentially operate on the same underlying scripting DOM. So, if you understanding the InDesign object model and the corresponding InDesign scripting objects, it should not be difficult to migrate an existing Java solution to a scripting solution.

To get started with InDesign Server scripting, refer to the *Adobe InDesign Scripting Tutorial*. If you have never created a script before, this document shows you how to get started. The document also covers how to install and run an InDesign script and describes what InDesign scripting can and cannot do. It discusses the software you need to start writing your own scripts.

After you learn the basics of InDesign scripting, you can move on to the *Adobe InDesign Scripting Guide*, which explores scripting in more depth. The *Adobe InDesign Scripting Guide* contains hundreds of tutorial scripts covering topics such as text formatting, finding and changing text, drawing objects on a page, and exporting documents.

This section provides a few samples of basic InDesign Server operations in both Java and JavaScript.

Getting the Application object

Java

```
// get the path to the IOR file
String iorFile = args[0]; // represents path to IOR file
// read the first line of the IOR file using a buffered reader
BufferedReader iorIn = new BufferedReader(new FileReader(iorFile));
String application_IOR = iorIn.readLine();
// Assuming the info we read from the IOR represents an Application
// object, use the ORB class to convert the info string into an object
ORB orb = ORB.init(new String [] {iorFile}, null);
Object object = orb.string_to_object(application_IOR);
// convert the object initialized from the IOR to an IDS Application object
Application myApp = ApplicationHelper.narrow(object);
```

JavaScript

In JavaScript, the application object is readily available for use: `app`.

Creating a document

Java

```
Document myDocument = myApp.addDocument(OptArg.noDocumentPreset());
```

JavaScript

Because JavaScript supports the use of default arguments, no parameter is required if you want to add a document with the default settings:

```
var myDocument = app.documents.add();
```

Opening a document

Java

```
VariableType vtDocument = myApp.open(VariableTypeUtils.createFile(filePath));
Document myDocument = DocumentHelper.narrow(vtDocument.asObject());
```

JavaScript

```
var myDocument = app.open("c:\\pathToDocument.indd");
```

Closing a document

Java

```
Document myDocument = myApp.getNthChildDocument(0);
if (myDocument != null)
{
    // specify no save by sending in noSaveOptionsEnum()
    myDocument.close(OptArg.noSaveOptionsEnum(),OptArg.noFile(),
    OptArg.noString(), OptArg.noBoolean());
}
```

JavaScript

```
var myDocument = app.documents.item(0);
//Close the document.
myDocument.close();
```

Saving a document

Java

```
Document myDocument = myApp.getNthChildDocument(0);
String filepath = "c:\\ServerTestFiles\\SaveDocumentAs.indd";
myDocument.save(OptArg.makeFile(filepath),
    OptArg.noBoolean(), // default option for save as stationary
    OptArg.noString(), // default option for version string
    OptArg.noBoolean());// default option for force save
```

JavaScript

```
var myDocument = app.documents.item(0);
myDocument.save(new File("/c/ ServerTestFiles / SaveDocumentAs.indd "));
```

Defining page size and document length

Java

```
// Add a document with no preset
Document myDocument = myApp.addDocument(OptArg.noDocumentPreset());
// Set the desired preferences for this document
DocumentPreference docPrefs = myDocument.getDocumentPreferences();
docPrefs.setPageHeight(UnitUtils.createString("800pt"));
docPrefs.setPageWidth(UnitUtils.createString("600pt"));
docPrefs.setPageOrientation(kPageOrientationLandscape.value);
docPrefs.setPagesPerDocument(16);
```

JavaScript

```
var myDocument = app.documents.add();
myDocument.documentPreferences.pageHeight = 800;
myDocument.documentPreferences.pageWidth = 600;
myDocument.documentPreferences.pageOrientation = PageOrientation.landscape;
myDocument.documentPreferences.pagesPerDocument=16;
```

Setting page margins and columns

Java

```
Document myDocument = myApp.addDocument(OptArg.noDocumentPreset());
Page myPage = myDocument.getNthChildPage(0);
MarginPreference myMarginPref = myPage.getMarginPreferences();
myMarginPref.setColumnCount(3);
myMarginPref.setColumnGutter(UnitUtils.createString("1p"));
myMarginPref.setTop(UnitUtils.createString("4p"));
myMarginPref.setBottom(UnitUtils.createString("6p"));
```

JavaScript

```
var myDocument = app.documents.add();
var myPage = myDocument.pages.item(0);
myPage.marginPreferences.columnCount = 3;
myPage.marginPreferences.columnGutter = 1;
myPage.marginPreferences.top = 4;
myPage.marginPreferences.bottom = 6;
```

Creating a text frame

Java

```
Document myDocument = myApp.addDocument(OptArg.noDocumentPreset());
// create a text frame on page 1
TextFrame myTextFrame = myDocument.getNthChildPage(0).addTextFrame(
    OptArg.noLayer(),
    OptArg.noLocationOptionsEnum(),
    OptArg.noVariableType());
// set the bounds of the text frame
myTextFrame.setGeometricBounds(
    UnitUtils.createDouble(new double[] {72, 72, 288, 288}));
// enter text in the text frame
myTextFrame.setContents(VariableTypeUtils.createString("This is example text."));
```

JavaScript

```
var myDocument = app.documents.add();
var myTextFrame = myDocument.pages.item(0).textFrames.add();
myTextFrame.geometricBounds = ["72p", "72p", "288p", "288p"];
myTextFrame.contents = " This is example text.";
```

Differences between Java API and InDesign Scripting API

The InDesign Server Java API closely resembles the InDesign Scripting API; however, there are some key differences between the scripting languages (AppleScript, JavaScript, and Visual Basic) and Java. To deal with these differences, the old InDesign Java API contained some special elements, discussed in this section.

VariableType

The scripting languages supported by the scripting API are loosely typed; that is, you can set a variable to any *type* of data, and the variable's type is interpreted based on context. On the other hand, Java is strongly typed, so variables must be declared as a specific type before using them. To deal with this difference, some member data and method parameters in the Java API were implemented using the *VariableType* class.

The *VariableType* class acted as a wrapper for an object, allowing the object to be treated as loosely typed while maintaining its strongly typed data internally. A *VariableType* object could represent any of a variety of types, including Boolean, String, int, and SObject.

Creating a *VariableType* object required using the *VariableTypeUtils* class, which contained *createX()* methods for all its implemented types. For example, to create a *VariableType* object to hold a String, Java did the following:

```
VariableType vtString = VariableTypeUtils.createString("Hello");
```

You also can use the *VariableTypeUtils* class to pass an object directly into a method:

```
textFrame.setContents(VariableTypeUtils.createString("Hello"));
```

In JavaScript, the first case isn't needed at all—you don't need to create the text object—and the second case becomes simply:

```
textFrame.contents = "Hello";
```

Helpers and holders

For almost every class within the IDS Java API, there existed a *Helper* and a *Holder* class. The *Helper* handled type-casting, and the *Holder* implemented streaming.

A *Helper* was used when you needed to convert from a *VariableType* object to a specific object type. Here is an example where an *Application* object's *open()* method is called. The *open* method returns a *VariableType* object, but that object can actually represent a Document, Book, Library, or Array of Documents, Books, or Libraries. In our example, we know it represents a document, so Java used the *DocumentHelper narrow()* method to return a *Document* object:

```
VariableType vtDocument = application.open(
VariableTypeUtils.createFile("c:\\myDoc.indd"));
Document document = DocumentHelper.narrow(vtDocument.asObject());
```

Helper and holders are not needed in scripts. such as in JavaScript:

```
var document = app.open(File(c:\\myDoc.indd));
```

OptArg

The InDesign scripting API supports optional arguments, which means that you are not forced to pass a value for every parameter in a function's parameter list. In Java, however, there is no support for optional arguments. To maintain consistency with the scripting API, the IDS Java API contained the OptArg type. OptArg was used in parameter lists in places where the scripting API allows an optional argument. When writing the code to call a method that requires an OptArg, Java applications used the *OptArg.createX()* method to pass in a value, and *OptArg.noX()* to pass in no value. Here are a few examples:

```
Document doc = myApp.addDocument(OptArg.noDocumentPreset());
TextFrame textFrame = doc.addTextFrame(OptArg.noLayer(),
    OptArg.noLocationOptionsEnum(), OptArg.noVariableType());
    Guide myGuide = myPage.addGuide(OptArg.makeLayer(myGuideLayer));
```

In JavaScript, the unused arguments don't need to be specified at all, and nothing special needs to be done with optional arguments that contain a value.

enums

Enum support was introduced in Java 1.5, but the IDS Java API supported back to Java 1.4.2, so the IDS Java API did not use enums. To maintain close similarity to the InDesign scripting API (which relies heavily on enums), the IDS Java API provided a complete set of "enum" interfaces whose names were based on

enums from the scripting DOM. Each of these interfaces had a static final member named "value" that returned the enum value. For example, the scripting enum SaveOptions.yes was implemented in the IDS Java API as kSaveOptionsYes.value; likewise, the scripting enum OTFFigureStyle.proportionalOldstyle becomes kOTFFigureStyleProportionalOldstyle.value in IDS Java. Here is an example of a call to a method using an enum class:

```
myGuide.setOrientation(kHorizontalOrVerticalVertical.value);
```

Look for these and simplify them in JavaScript to

```
myGuide.orientation = HorizontalOrVertical.horizontal
```

Units of measurement

The IDS Java API used its *Unit* class to represent values in *get* and *set* methods such as *setPageHeight()*. A Unit could be created using the *UnitUtils* class. UnitUtils had several methods for creating a Unit, including *createString(), createDouble()*, etc. For example, a String value of "3i," representing 3 inches, was created by calling UnitUtils.createString("3i"). When using a double value in a set method, the unit of measurement corresponded to the document's view-preferences' measurement units. So, when passing in double values to *set* methods, you needed to be aware of the current measurement units. This sample code demonstrates the use of both String and doubles:

```
// extracted from: UnitsOfMeasurement.java

Document myDocument = myApp.getNthChildDocument(0);

// Use UnitUtils.createString to pass in a string value -
// this allows you to specify what the units are. For example,
// 7i = 7 inches, 7p = 7 picas, 7pt = 7 points. Using the
// string values lets you avoid setting view preference measurement
// units to specify what units to use.
DocumentPreference docPref = myDocument.getDocumentPreferences();
docPref.setPageWidth(UnitUtils.createString("7i"));
docPref.setPageHeight(UnitUtils.createString("9i"));

// An alternative is to use the ViewPreferences to
// specify what units to use...

// if this is an existing document, I might not be sure what
// the preferred units of measurement are. Let's find out...
ViewPreference viewPrefs = myDocument.getViewPreferences();
int hUnits = viewPrefs.getHorizontalMeasurementUnits();
int vUnits = viewPrefs.getVerticalMeasurementUnits();

// set my measurement units to inches
viewPrefs.setHorizontalMeasurementUnits(kMeasurementUnitsInches.value);
viewPrefs.setVerticalMeasurementUnits(kMeasurementUnitsInches.value);

// set the page size to 7 inches x 9 inches
docPref = myDocument.getDocumentPreferences();
docPref.setPageWidth(UnitUtils.createDouble(7));
docPref.setPageHeight(UnitUtils.createDouble(9));

// reset my measurement units
viewPrefs.setHorizontalMeasurementUnits(hUnits);
viewPrefs.setVerticalMeasurementUnits(vUnits);
```

In JavaScript, we can either use a string directly to specify what the units are, or set preferred units of measurement, then set the number value. For example:

```
var myDocument = app.documents.item(0);
//Use string directly specify what the units are
with(myDocument.documentPreferences){
    pageHeight = "9i";
    pageWidth = "7i";
    }
// if this is an existing document, I might not be sure what
// the preferred units of measurement are. Let's find out...
with(myDocument.viewPreferences){
       var hUnit = horizontalMeasurementUnits;
       var vUnit = verticalMeasurementUnits;
    horizontalMeasurementUnits = MeasurementUnits.INCHES;
        verticalMeasurementUnits = MeasurementUnits.INCHES;
    }
// set the page size to 7 inches x 9 inches
with(myDocument.documentPreferences){
    pageHeight = 9;
    pageWidth = 7;
    }
// reset my measurement units
with(myDocument.viewPreferences){
    horizontalMeasurementUnits = hUnit;
        verticalMeasurementUnits = vUnit;
    }
```

Get and set methods

In the scripting DOM, objects are accessed directly. For instance, to obtain the zeroth document owned by the application using JavaScript:

```
// JavaScript
var myDocument = myApp.documents.item(0);
var layer1 = myDoc.layers.item("Layer1");
var layer1Name = layer1.name;
app.marginPreferences.top = 0.0;
```

The Java API hid direct access to contained objects, so to access objects like Documents, Layers, and Guides, applications used the *getX()* and *setX()* methods of the owning class, where X is some member data. For example:

```
Document myDoc = myApp.getNthChildDocument(0);
Layer layer1 = myDoc.getNamedChildLayer("Layer1");
String layer1Name = layer1.getName();
MarginPreference marginPrefs = myApp.getMarginPreferences().setTop(0.0);
```

Errors and exception handling

Many methods within the IDS Java API threw exceptions. To gracefully handle these exceptions, applications put a try/catch block around the code. The *IdsException* class provided a suite of methods for handling errors. Here is a sample try/catch block:

```
try {
    // attempt to get a document and use it - this code will throw if no
    // documents are open, because myDocument will be set to null, and
    // dereferencing null will throw an exception.
    Document myDocument = myApp.getNthChildDocument(0);
    myDocument.setActiveLayer(myDocument.getNamedChildLayer("Layer9"));
}
catch(IdsException e) {
    System.err.println("Exception #" + e.errorCode + ": " + e.errorMsg);
    e.printStackTrace();
}
```

Notice the code in the catch block. The println writes the error code and message to the current system-error window, and e.printStackTrace() prints the stack trace to the current system-error window. The identity of the system-error window depends on where you initiate the code. For example, if you are running the code from a shell window, your shell window receives the output. If you are running from Eclipse, the Eclipse console receives the output. You could also use the *IdsException* method *printStackTrace* to write errors to a PrintStream or PrintWriter.

In JavaScript, a try/catch block is also a graceful way to handle code that might throw exceptions. For example:

```
try{
    var myDocument = app.documents.item(0);
    myDocument.activeLayer = myDocument.layers.itemByName("Layer9");
    }
catch(myError){
    alert(myError);
    }
```

Working with InDesign Server SOAP

Chapter Update Status		
CS6	Edited	Content not guaranteed to be current. Only the following changed: • Removed content related to Java and CORBA. • Added information about the BeginSession, EndSession, and BeginSessionResponse elements.

Adobe® InDesign® Server opens communication channels via SOAP. This chapter describes the SOAP implementation for InDesign Server, namely, the `RunScript` method.

This chapter contains:

- "Terminology" on page 16
- "InDesign Server WSDL (Web Services Description Language)" on page 17
- "The RunScript method" on page 19
- "InDesign Server SOAP messages and responses" on page 20
- "SOAP client implementations" on page 26
- "Debugging tips" on page 28
- "Frequently asked questions" on page 28
- "References" on page 29

Terminology

SOAP (Simple Object Access Protocol) is an XML-based syntax and protocol specification for exchanging data between applications in a networked environment. SOAP provides the framework by which application-specific data may be conveyed in an extensible manner, as well as a full description of the required actions taken by a *SOAP node* on receiving a *SOAP message*.

A *SOAP message* is defined as "The basic unit of communication between SOAP nodes." A *SOAP node* is defined as "The embodiment of the processing logic necessary to transmit, receive, process and/or relay a SOAP message, according to the set of conventions defined by this recommendation. A SOAP node is responsible for enforcing the rules that govern the exchange of SOAP messages (see below). It accesses the services provided by the underlying protocols through one or more SOAP bindings." (Definitions are from http://www.w3.org/TR/soap12-part1/#terminology.)

InDesign Server WSDL (Web Services Description Language)

What is WSDL?

WSDL is an XML-formatted language used to describe a Web service's capabilities as collections of communication endpoints capable of exchanging messages. In this context, a WSDL file is an XML file that defines the types, method(s), parameters, and result structures of InDesign Server's SOAP implementation.

There are many programming languages that can be used to implement a SOAP solution, and there are a variety of toolkits and APIs that can be used depending on the language you choose for your solution. In general, when using a statically typed language (like Java, C++, and C#), the toolkit provides a tool that generates source code based on the WSDL. For example, Apache Axis provides the `wsdl4j` tool, which generates Java code based on a WSDL. You then include the generated source code in your solution. For dynamically typed languages (like PHP and ASP.NET), the WSDL is used by the API dynamically at runtime.

Location of the InDesign Server WSDL

There are two ways to access the InDesign Server WSDL:

- Use the `IDSP.wsdl` file located in the `docs/references` folder of the InDesign Server SDK. The InDesign Server WSDL contains a `<SOAP:address location>` element that defines the default instance of InDesign Server with which a client using the WSDL will interact. The default location is `http://localhost:80`. You can modify this default by editing your version of `IDSP.wsdl` to use the desired instance of InDesign Server. The location element is located near the end of the file.
- Use HTTP to request the WSDL from a running instance of InDesign Server. For example: `http://localhost:12345/service?wsdl`. When accessing the WSDL through HTTP, the location element gets defined as the instance of InDesign Server used to generate the WSDL. In this example, InDesign Server was started with the -port option set to 12345; therefore, the instance is `http://localhost:12345`.

What is in the InDesign Server WSDL?

The syntax of the primary types defined in the InDesign Server WSDL are given here:

- **`IDSP-ScriptArg`** — A sequence containing strings for name and value. The `IDSP-ScriptArg` type is used within the `RunScriptParameters` type.

```
<complexType name="IDSP-ScriptArg">
    <sequence>
        <element name="name" type="xsd:string" minOccurs="1" maxOccurs="1" />
        <element name="value" type="xsd:string" minOccurs="1" maxOccurs="1" />
    </sequence>
</complexType>
```

- **`RunScriptParameters`** — A sequence containing `scriptText`, `scriptLanguage`, `scriptFile`, and `scriptArgs` elements. The `RunScriptParameters` type is used as a parameter to the `RunScript` method.

```
<complexType name="RunScriptParameters">
    <sequence>
        <element name="scriptText" type="xsd:string"
            minOccurs="0" maxOccurs="1" nillable="true" />
        <element name="scriptLanguage" type="xsd:string"
            minOccurs="0" maxOccurs="1" nillable="true" />
        <element name="scriptFile" type="xsd:string"
            minOccurs="0" maxOccurs="1" nillable="true" />
        <element name="scriptArgs" type="IDSP:IDSP-ScriptArg"
            minOccurs="0" maxOccurs="unbounded" />
    </sequence>
</complexType>
```

- **`RunScript`** — The method for running a script in InDesign Server.

```
<element name="RunScript">
    <complexType>
        <sequence>
        <element name="runScriptParameters" type="IDSP:RunScriptParameters"
            minOccurs="0" maxOccurs="1" nillable="true" />
        </sequence>
    </complexType>
</element>
```

- **`RunScriptResponse`** — The return type of the `RunScript` method.

```
<element name="RunScriptResponse">
    <complexType>
        <sequence>
            <element name="errorNumber" type="xsd:int" minOccurs="1" maxOccurs="1"/>
            <element name="errorString" type="xsd:string"
                minOccurs="0" maxOccurs="1" nillable="true"/>
            <element name="scriptResult" type="IDSP:Data"
                minOccurs="0" maxOccurs="1" nillable="true" />
        </sequence>
    </complexType>
</element>
```

- **`BeginSession`** — The method for starting a session in InDesign Server.

```
<element name="BeginSession">
    <complexType>
        <sequence>
        </sequence>
    </complexType>
</element>
```

- **`BeginSessionResponse`** — The return type of the `BeginSession` method.

```
<element name="BeginSessionResponse">
    <complexType>
        <sequence>
            <element name="sessionID" type="IDSP:SessionID"
                minOccurs="1" maxOccurs="1"/>
        </sequence>
    </complexType>
</element>
```

- **`EndSession`** — The method for ending a session in InDesign Server.

```
<element name="EndSession">
    <complexType>
        <sequence>
            <element name="sessionID" type="IDSP:SessionID"
                minOccurs="1" maxOccurs="1"/>
        </sequence>
    </complexType>
</element>
```

The InDesign Server SOAP method

The Adobe InDesign Server SOAP implementation provides three methods, `RunScript`, `BeginSession`, and `EndSession`.

The RunScript method

The `RunScript` method, its parameter list, and its result structure are defined by the InDesign Server WSDL. The `RunScript` method passes an InDesign Server-compatible script to InDesign Server. InDesign Server executes the script and returns the result to the caller.

The script can be written in JavaScript, AppleScript, or VBScript. It must be based on the InDesign Server scripting DOM (Document Object Model). Information about the InDesign Server scripting DOM is in *Adobe InDesign Server Scripting Guide*, located in the `scripting` folder within the InDesign Server SDK.

How you call `RunScript` depends on the language with which you are building your solution and what toolkit or API you are using. Generally, you use a SOAP API to create a *client* object that can call the methods defined in the WSDL. In interpreted languages (for example, PHP and ASP), you use a reference to the InDesign Server WSDL to create the client at runtime. In compiled languages (for example, C++, C#, and Java), you must first generate client code based on the WSDL, include that code in your project, and then instantiate a *client* object using the generated code.

For examples of how to create a *client* object and call `RunScript`, look at the sample client projects included with the InDesign Server SDK. There are samples for Java, Flex, C#, PHP, and ASP.NET, and in the InDesign plug-in SDK, there is a C++ example, SampleClient.

The BeginSession method

If you want to run a script in a session context, call this before calling the `RunScript` method. The `BeginSession` method's parameter list and result structure are defined by the InDesign Server WSDL. The `BeginSession` method notifies InDesign Server to open a session for the script execution and to return the session ID to the client.

For examples of how to begin a session, look at the sample client projects included with the InDesign Server SDK. There are samples for Java, C#, and C++.

The EndSession method

The `EndSession` method's parameter list and result structure are defined by the InDesign Server WSDL. The `EndSession` method notifies InDesign Server to close a specific session for the script execution based on the passed session ID.

For examples of how to end a session, look at the sample client projects included with the InDesign Server SDK. There are samples for Java and C#, and in the InDesign plug-in SDK, there is a C++ example, SampleClient.

InDesign Server SOAP messages and responses

SOAP works by passing XML messages between applications. The XML message that is sent from a client to InDesign Server is called a *SOAP request*, and the XML message that InDesign Server returns to the client is called a *SOAP response*. A SOAP XML message contains the following:

- The root element, of type `SOAP-ENV:Envelope`.
- The envelope element contains at least one element, of type `SOAP-ENV:Body`, and response messages also precede the body with an element of type `SOAP-ENV:Header`.
- The body element contains elements that describe method requests or the responses to method requests. For example, if the message is a request being sent to InDesign Server to run a script, the first element within the body element is a `RunScript` element. If the message is a response being returned from InDesign Server's `RunScript` method, the first element within the body element is a `RunScriptResponse` element.

The RunScript request envelope

The body of the request envelope for running a script contains one element, `RunScript`, which contains an element named `runScriptParameters`. The `runScriptParameters` element contains the following four elements, which tell InDesign Server all it needs to know to run your script:

- `scriptText` — A string passed to InDesign Server, containing the entire script to be executed. This parameter is ignored if the `scriptFile` parameter has a value.
- `scriptLanguage` — One of `javascript`, `applescript`, or `visual basic`.
- `scriptFile` — The path to the script to be executed by InDesign Server. The path can take one of two forms:
  - An absolute path to the script based on the file system of the targeted InDesign Server instance. For example:

    Windows: `c:/myScriptsFolder/myScript.jsx`

    Mac OS: `/myScriptsFolder/myScript.jsx`
  - A relative path to either the InDesign Server application scripts folder or the InDesign Server user's scripts folder. You must use a colon (:) as the path separator, on both Windows and Mac OS.

    The path prefix `Scripts:Application:` represents the application scripts folder:

    Windows: `C:\Program Files\Adobe\Adobe InDesign CS6 Server\Scripts\`

    Mac OS: `/Applications/Adobe InDesign CS6 Server/Scripts/`

    The path prefix `Scripts:User:` represents the user's scripts folder:

    Windows:
    ```
    C:\Documents and Settings\myName\Application Data\Adobe\
        InDesign Server\Version 8.0\en_US\configuration_12345\Scripts\
    ```

Mac OS: `/Users/myName/Library/Preferences/Adobe InDesign Server/Version 8.0/en_US/configuration_12345/Scripts/`

Examples:

`Scripts:Application:`*`myScriptsFolder`*`:myScript.jsx`
`Scripts:User:`*`myScriptsFolder`*`:myScript.jsx`

- `scriptArgs` — A series of elements, where each element contains `name` and `value` elements. ScriptArgs are stored by InDesign Server. They are accessed from within your script by accessing the InDesign Server `scriptArgs` object. Each of the following examples assigns a local variable to the value of a `scriptArg` named `argOne`.

JavaScript:
```
if (app.scriptArgs.isDefined("argOne")) {
    var myArgValue = app.scriptArgs.getValue("argOne");
    }
```

Applescript:
```
tell script args
    if is defined name "argOne" then
        set myArgValue to get value name "argOne"
    end if
end tell
```

VBScript:
```
if myApp.ScriptArgs.IsDefined("argOne") then
    myArgValue = myApp.ScriptArgs.GetValue("argOne")
end if
```

Example The following is an example of a SOAP request that tells InDesign Server to run a JavaScript located at `c:\examplefiles\test.jsx`. Two `scriptArgs` are passed: arg0 = 88, and arg1 = "some text."

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">

    <SOAP-ENV:Body>

        <IDSP:RunScript>
            <IDSP:runScriptParameters>
                <IDSP:scriptText></IDSP:scriptText>
                <IDSP:scriptLanguage>javascript</IDSP:scriptLanguage>
                <IDSP:scriptFile>c:\examplefiles\test.jsx</IDSP:scriptFile>
                <IDSP:scriptArgs>
                    <IDSP:name>arg0</IDSP:name>
                    <IDSP:value>88</IDSP:value>
                </IDSP:scriptArgs>
                <IDSP:scriptArgs>
                    <IDSP:name>arg1</IDSP:name>
                    <IDSP:value>some text</IDSP:value>
                </IDSP:scriptArgs>
            </IDSP:runScriptParameters>
        </IDSP:RunScript>

    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The RunScript response envelope

The `RunScript` response envelope contains the result of the `RunScript` SOAP call. This response can be the return value of the script passed to InDesign Server, the value of an error caused by the script passed to InDesign Server, or the value of a SOAP fault.

If the result is a SOAP fault, the Body element contains one `Fault` element, which contains `faultcode` and `faultstring` elements.

If there was no SOAP fault, the Body of the response envelope contains one element, `RunScriptResponse`, which contains the following elements:

- `errorNumber` — The value of the error. If no error occurred, the value is 0.
- `errorString` — A string containing the error message returned from InDesign Server. This element is present only if errorNumber is not 0.
- `scriptResult` — The data returned from the script run by InDesign Server. In JavaScript and AppleScript, the script's return value is the last value encountered in the script. In VBScript, you set a variable named `returnValue` to the return value for the script. Each of the following examples returns the name of the document at the first index.

  JavaScript:
  ```
  var documentName = app.documents.item(0).name;
  documentName;
  ```

  AppleScript:
  ```
  tell application "InDesignServer"
      set documentName to name of document 1
  end tell
  documentName
  ```

  VBScript:
  ```
  Set myApp = CreateObject("InDesignServer.Application.CS6")
  documentName = myApp.Documents.Item(1).Name
  returnValue = documentName
  ```

The following are simple examples of each of these types of response envelopes.

Example The following is an example of a `RunScriptResponse` that contains a return value. In this case, the JavaScript passed to InDesign Server returned an array containing these elements: `"1"`, `"2"`, `10`, `12`.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">

    <SOAP-ENV:Body>
        <IDSP:RunScriptResponse>
            <errorNumber>0</errorNumber>
            <scriptResult>
                <data xsi:type="IDSP:List">
                    <item><data xsi:type="xsd:string">1</data></item>
                    <item><data xsi:type="xsd:string">2</data></item>
                    <item><data xsi:type="xsd:long">10</data></item>
                    <item><data xsi:type="xsd:long">12</data></item>
                </data>
            </scriptResult>
```

```
        </IDSP:RunScriptResponse>
    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Example The following is an example of a `RunScriptResponse` containing an error (25) caused by a parse error ("Expected: ;") in the JavaScript:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">
    <SOAP-ENV:Body>
        <IDSP:RunScriptResponse>
            <errorNumber>25</errorNumber>
            <errorString>Expected: ;</errorString>
            <scriptResult></scriptResult>
        </IDSP:RunScriptResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example The following is an example of a `RunScriptResponse` containing a SOAP fault caused by a file-not-found error:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>SOAP-ENV:Server</faultcode>
            <faultstring>The script file specified can not be found</faultstring>
            <detail>None</detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The BeginSession request envelope

The body of the request envelope to open a session context contains one element, `BeginSession`, which contains no elements.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">
    <SOAP-ENV:Body>
        <IDSP:BeginSession/>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The BeginSession response envelope

The `BeginSession` response envelope contains the result of the `BeginSession` SOAP call. This response can be either the newly created session ID requested by the `BeginSession` SOAP call or the value of a SOAP fault.

If the result is a SOAP fault, the Body element contains one `Fault` element, which contains `faultcode` and `faultstring` elements.

If there was no SOAP fault, the body of the response envelope contains one element, `BeginSessionResponse`, which contains the following element:

- `sessionID` — The identifier of the newly created session, which is an integer such as 2.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">

    <SOAP-ENV:Body>
        <IDSP:BeginSessionResponse>
            <sessionID>2</sessionID>
        </IDSP:BeginSessionResponse>
    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The EndSession request envelope

The header of the request envelope to close a session context contains one element:

- `sessionID` — The identifier of the current session, which is an integer such as 2.

The body of the request envelope contains one element, `EndSession`. The `EndSession` element contains the following element, which tells InDesign Server to close the specified session:

- `sessionID` — The identifier of the current session, which is an integer such as 2.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">

    <SOAP-ENV:Header>
        <IDSP:sessionID>2</IDSP:sessionID>
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
        <IDSP:EndSession>
            <sessionID>2</sessionID>
        </IDSP:EndSession>
    </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

The EndSession response envelope

The `EndSession` response envelope contains the result of the `EndSession` SOAP call, which specifies whether a fault occurred when closing the specified session.

The header of the response envelope contains one element:

- `sessionID` — The identifier of the session being ended, which is an integer such as 2.

If the result is a SOAP fault, the `Body` element contains one `Fault` element, which contains `faultcode` and `faultstring` elements. See "The RunScript response envelope" on page 22 for an example of a returned SOAP fault.

If there was no SOAP fault, the body of the response envelope contains one empty `Result` element.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:IDSP="http://ns.adobe.com/InDesign/soap/">

    <SOAP-ENV:Header>
        <IDSP:sessionID>2</IDSP:sessionID>
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
        <IDSP:Result/>
    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Message serialization and deserialization

Most Web Service toolkits provide message serialization and deserialization. This allows you to represent your request and response messages using objects that are native to the programming language, rather than XML strings.

For example, the `RunScript` method accepts a parameter of type `RunScriptParameters`. To call `RunScript`, you first set up the data required to initialize the `RunScriptParameters` structure. The way you set up this data depends on the programming language and toolkit you use. In general, though, you set up `RunScriptParameters` as an object that is native to the language, like an array, structure, or class. When a call to `RunScript` is made, the toolkit serializes `RunScriptParameters` and uses the resulting XML string to create the request message.

Likewise, the `RunScriptResponse` result of the `RunScript` method is returned to you as an object that is native to the programming language, not as an XML string. The toolkit takes care of the deserialization from a response message XML string to an object.

SOAP client implementations

The InDesign Server installation includes a sample SOAP client application named `sampleclient`. This command-line application allows you to easily send a script to InDesign Server via SOAP, and receive the result returned from the script. `sampleclient` is written in C++, and the source code is included in the InDesign plug-in SDK.

The InDesign Server SDK contains samples demonstrating how to create a similar sample client using a variety of technologies, including Java, C#.NET, ASP.NET, PHP, and Flex.

The following sections contain brief descriptions of the technology behind the sample clients. For more detailed information, see the samples in the SDKs and *Getting Started With the Adobe InDesign Server SDK*, located in the `<IDS SDK>/docs/guides` folder.

Java

The Java sample is a command-line application that employs the Apache Axis Web Service framework. The Axis framework provides the `wsdl2java` tool, used to generate source code based on a WSDL. The generated code is then packaged into a `.jar` file for use by the client application. The client application can be built using Eclipse or Ant, on Windows® or Mac OS®.

For more information on Apache Axis, go to http://ws.apache.org/axis/.

The sample project is located at `<IDS SDK>/samples/sampleclient-java-soap.`

C++

This sample is located in the InDesign plug-in SDK. The C++ sample is a command-line application that employs the gSOAP Web Service framework. The framework provides the `wsdl2h` and `soapcpp2` tools to generate source code based on a WSDL. The generated code is then included in the client application's project file. The project is built using Visual Studio on Windows and XCode on Mac OS.

For more information on gSOAP, go to http://gsoap2.sourceforge.net/.

Instructions on how to generate and compile the sample are in:

`<IDS SDK>/samples/sampleclient-cplusplus-soap/Readme.txt`

C# .NET

The C# sample is a command-line application that employs the .NET framework, which is a Windows-only technology. The .NET framework provides Web Reference technology to generate a proxy class representing the WSDL. This proxy class is instantiated by the client, allowing access to the types and methods within the WSDL. The client application is built using Visual Studio 2008.

For more information on C# .NET, go to http://msdn.microsoft.com/en-us/vcsharp/default.aspx.

The project is located at `<IDS SDK>/samples/sampleclient-csharp-soap`.

ASP.NET

ASP.NET is a server-side scripting technology requiring IIS (Internet Information Services) Web server. An ASP.NET project is developed and hosted using only Windows, but the client Web page can be accessed from any platform. The .NET framework provides Web Reference technology to generate a proxy class representing the WSDL. This generated file is added to the client Web page project by the Web Reference. The sample client is built using Visual Studio .NET and accessed from a Web browser.

For more information on ASP.NET, go to http://www.asp.net/.

For more information on IIS, go to http://www.microsoft.com/WindowsServer2003/iis/default.mspx.

The project is located at `<IDS SDK>/samples/sampleclient-aspnet-soap`.

PHP

PHP is a server-side scripting technology requiring a Web Server (IIS or Apache). PHP can be written using any text editor, and it is cross platform. This sample employs two of the major frameworks use to develop for SOAP in PHP: PHP:SOAP and NuSoap. Each framework provides an API for accessing a WSDL and generating a client object based on the WSDL. The PHP sample client is accessed from a Web browser.

For more information on:	Go to:
PHP	http://www.php.net/
PHP:SOAP	http://www.php.net/soap
NuSOAP	http://www.sourceforge.net/projects/nusoap/
IIS	http://www.microsoft.com/WindowsServer2003/iis/default.mspx
Apache Web Server	http://httpd.apache.org/

The project is located at `<IDS SDK>/samples/sampleclient-php-soap`.

Flex

The Flex sample client is an Adobe Flex Builder 3 project. It uses the Flex WebService API (mx.rpc.soap.WebService) to load the InDesign Server WSDL at runtime. The WebService API handles all serialization and deserialization of the SOAP packets that are sent to and received from InDesign Server. This sample is deployed in a browser, and has a simple User Interface allowing you to configure all `RunScript` parameters.

For more information on Flex and the Flex WebService, go to:

http://www.adobe.com/devnet/flex/

http://livedocs.adobe.com/flex/3/

http://livedocs.adobe.com/flex/3/langref/mx/rpc/soap/mxml/WebService.html

The project is located at `<IDS SDK>/samples/sampleclient-flex-soap`.

Other technologies

There are even more languages and frameworks available for developing SOAP clients for InDesign Server, including the following:

- Mac OS X Developer Tools CD — `WSMakeStubs`. Located in the `/Developer/Tools/` folder. Generates stubs based on the WebServicesCore.framework for AppleScript, C++ and Objective C.
- Java™ — IBM Web Services and Sun™ Microsystems Web Services. Contains tools and APIs for use with Java.
- Perl — SOAPLite for Perl. Provides Perl modules for writing SOAP client scripts.

Debugging tips

You may find the following tips helpful when debugging your client:

- Use a packet sniffer to monitor the XML data sent to and from InDesign Server. There are many available; for example, Charles (http://www.xk72.com/charles/).
- When writing PHP code, use print statements to trace information to the Web page.
- Have your script print output to the InDesign Server console. The following are examples of how to write to the InDesign Server console:

JavaScript:
```
app.consoleout("my message");
```

AppleScript:
```
tell application "InDesignServer"
    consoleout message "my message"
end tell
```

VBScript:
```
Set myApp = CreateObject("InDesignServer.Application.CS6")
myApp.ConsoleOut("my message");
```

Frequently asked questions

What can I do in the script that I pass into the sample client?

Basically, anything you can do in a regular InDesign Server script. If you have an InDesign Server plug-in that provides specific features, you need to provide scripting support so you can automate your feature using InDesign Server. For more details on how to add scripting support for your plug-in, see *Making Your Plug-in Scriptable*, a technical note included with the InDesign plug-in SDK.

When I send a script to InDesign Server using the sample client, I get a message containing an error code. What does this mean?

The error displayed in the sample client comes from InDesign Server after calling the InDesign Server `RunScript` method. The response usually contains a string associated with the error, giving more details; sometimes, however, the error string is not passed back to the client, but is written to the InDesign Server console.

For more information on InDesign errors, look at the error code listing in the InDesign plug-in SDK at `<SDK>/docs/references/errorcodes.htm`.

What happens to InDesign Server when a client terminates?

InDesign Server continues to run, waiting for more instructions.

Can a client communicate across multiple instances of InDesign Server?

A client communicates with only one instance of InDesign Server at a time; however, this does not mean that you cannot develop an application that communicates across multiple instances of InDesign Server. Keep in mind that each instance of InDesign Server (distinguished by TCP/IP port number) has its own set of "InDesign Defaults" and "InDesign Saved Data" files. These correspond to the databases represented by kWorkspaceBoss and kSessionBoss, respectively. If your client application works with multiple instances of InDesign Server, and it depends on this data, consider employing a strategy to keep these databases synchronized. For details, se "InDesign Server Plug-in Techniques" in *Getting Started with Adobe InDesign Plug-in Development*.

References

Publications

- "Scriptable Plug-in Fundamentals" in *Adobe InDesign Plug-In Programming Guide*, Adobe Systems Incorporated
- *Getting Started with Adobe InDesign Plug-in Development*, Adobe Systems Incorporated
- *Adobe InDesign Scripting Guide*, Adobe Systems Incorporated
- *Adobe InDesign Server Scripting Guide*, Adobe Systems Incorporated
- Apple® Developer Connection, *Web Services,* http://developer.apple.com/mac/library/navigation/index.html
- Cover Pages Technology Reports, Web Services Description Language (WSDL), http://xml.coverpages.org/wsdl.html
- Microsoft® Developer Network, Web Services Developer Center, http://msdn.microsoft.com/webservices/
- Microsoft Developer Network, *Understanding SOAP*, http://msdn2.microsoft.com/en-us/library/ms995800.aspx
- Microsoft Developer Network, *Understanding WSDL*, http://msdn2.microsoft.com/en-us/library/ms996486.aspx
- W3C SOAP technical reports, http://www.w3.org/TR/soap
- W3Schools SOAP Tutorial, http://www.w3schools.com/soap/

Tools

- Apache Axis, http://ws.apache.org/axis/
- gSOAP, http://sourceforge.net/projects/gsoap2/

- IBM Web Services (r)evolution, http://www-106.ibm.com/developerworks/webservices/library/ws-peer4/
- Microsoft Visual Studio, wsdl.exe, http://msdn.microsoft.com/en-us/library/7h3ystb6.aspx
- SOAPLite for Perl, http://www.soaplite.com/
- Sun Microsystems Web Services, http://java.sun.com/webservices/index.jsp

4 Scalability and Performance

Chapter Update Status		
CS6	Edited	Rearranged misordered paragraphs in "Mac OS tools" on page 35. Other content not guaranteed to be current.

This chapter discusses the issues surrounding Adobe® InDesign® Server scalability and performance. It provides information about tools you can use to quantify your system's performance, explains how to interpret the results, and gives advice on how to improve your configuration to get the best performance from InDesign Server. Benchmarking tools are also described here.

This chapter is not intended to be used as a final recommendation on system configuration. There are far too many details about your requirements that we cannot predict, and therefore we can only present this chapter as a guide to help you discover inefficiencies in your system.

Defining scalability and performance

For any system, *scalability* is complex, involving a very large number of factors. For systems that include InDesign Server as a component, major factors include disk performance, network performance, and CPU performance. Scalability also can depend on details like RAID stripe size and background processes.

Measuring the *performance* of a system allows you to determine whether your desired output is being met, and if not, where to modify your system to achieve the desired output. For a system using InDesign Server, quantifying performance can give you guidance as to whether you need to scale your system, either horizontally or vertically, or whether you need to modify your code to achieve the desired output.

This chapter discusses the performance of InDesign Server on a single machine, performing a single task, therefore isolating the performance of InDesign Server from other components such as network I/O.

Before following the steps in this chapter, you should design an easily repeatable test that demonstrates your performance problem. You may be able to use the SDK's Performance Testing Kit ("Performance-testing tools" on page 41) to build your test, or you might consider using the InDesign Performance Metrics API ("InDesign Performance Metrics API" on page 41) in your script or plug-in.

Expectation guidelines

This chapter contains general guidelines for the performance and scalability of a system involving InDesign Server. The remainder of this chapter will help you diagnose and solve issues where your system is not performing or scaling in a manner consistent with these expectations.

Different people define performance and scalability differently and have different expectations of how a system should perform. We define the *performance* of a system as its ability to handle a single task. The *scalability* of a system is its ability to handle multiple, simultaneous tasks without suffering decreases in performance. In other words, a scalable system is expected to have constant performance as it handles increases in load.

Software systems are expected to accommodate some level of load on any single machine. If you increase the power of the hardware on that machine, you expect the system to handle more load. This is commonly

called *vertical scalability*. Software systems also are expected to be *horizontally scalable*, meaning they can accommodate increases in load through additional machines. A vertically or horizontally scalable system handles these increases in load with no decrease in performance: any individual task completes in the same amount of time, regardless of the system's load.

Single-instance performance

One instance of InDesign Server can be expected to perform any task at least as fast as the desktop version of InDesign running on the same hardware. A scalability problem is a performance problem that is apparent only on simultaneous runs of a test, possibly on multiple machines. If one run of your test on one instance shows unsatisfactory performance, performance is an issue, but scalability is not a problem.

Single-machine scalability and multiple instances

InDesign Server can be scaled on a single machine using multiple instances, and the overall system should exhibit increases in throughput. Generally, the performance of incremental instances is slightly less than that of the original instance. So, for instance, two instances on one machine will not perform twice as fast as one instance on one machine. This is because the instances must share the machine's resources. The most important resources to InDesign Server are the CPU and the disk, so if you have a machine with multiple processors and multiple disks (and you direct each instance of InDesign Server to read/write from its own disk), you are most likely to achieve increases in performance proportional to increases in instances.

Horizontal scalability and multiple instances

InDesign Server also can be scaled on multiple machines using multiple instances and the overall system should exhibit increases in throughput. Generally, the performance of incremental instances is approximately the same as the original instance. So, for example, two instances of InDesign Server on two separate machines should perform twice as fast as one instance on one machine.

Sequential performance runs should be similar

The performance of multiple, sequential runs of a test should be similar. They will almost never be identical. This is because, at any time, the operating system behaves slightly differently (for example, because of what it is doing in the background), as does the disk (for example, because a file is written to a slightly different location or the disk is fuller). Over many tests, however, results should be similar.

Guilt by elimination

Often, InDesign Server is one piece of a larger system that involves many pieces. For example, a variable data publishing system might use InDesign Server for layout, but it also might include a Web server, application server, asset manager, and print queue.

To begin investigating any performance issue involving InDesign Server, eliminate all ancillary pieces of the system. For example, if your test involves InDesign Server using networked resources, such as files from a file server, rerun the test using local copies of those files. If your test uses a plug-in that communicates with another system, rerun the test using cached results from the other system.

Also pay attention to what other processes are running on the system. You can do this using the Task Manager (Windows®) or Activity Monitor (Mac OS®). If you see other processes using CPU, determine what

those processes are and whether they are necessary on that machine. If they are not necessary, remove them or shut them down. Especially for performance or scalability testing, run with as few other processes as possible, to focus only on InDesign Server.

If you run only the InDesign Server portion of the test, and running with only local resources no longer shows performance problems, you need to track down the performance issue in the appropriate ancillary system. In that case, this chapter is not relevant to you.

CPU bound or I/O bound?

Typically, InDesign Server is CPU bound or I/O bound, and the limitations of either the processor or the disk prevent InDesign Server from scaling the way you want it to. Operating system resources are very helpful in determining which one is the bottleneck for your test.

Windows tools

Performance Monitor

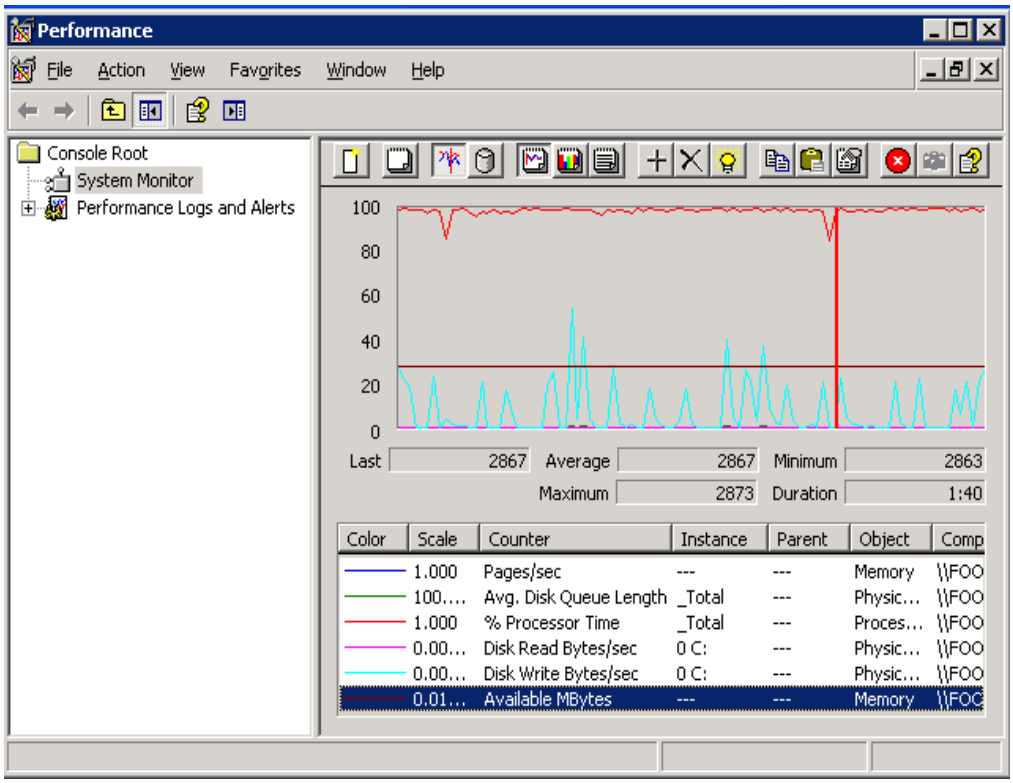
The Performance Monitor, PerfMon, can track many details about your system. Usually you start PerfMon at Start > Control Panel > Administrative Tools > Performance. If you are using the "Category View" of the Control Panel, you will find Administrative Tools under "Performance and Maintenance." By default, PerfMon shows CPU usage (% Processor Time) and disk usage (Avg. Disk Queue Length). It is highly customizable, but usually the default view is sufficient to determine whether you are CPU bound or I/O bound.

CPU bound

Run your test with only one instance of InDesign Server, and watch the Performance Monitor while the test runs. Watch the peaks of CPU usage; these are highly dependent on the number of processors on your machine. Think about seeing N times the CPU usage as you run N instances of InDesign Server. Would that be more than 100%? For example, if your machine has four single-core processors and you plan to run five instances of InDesign Server, are there any times during your test where CPU usage would be over 20%? If so, those are times when the scaling of InDesign Server would be limited by the CPU—when you would be CPU-bound.

Run your test again, this time against multiple instances of InDesign Server, and watch the Performance Monitor while you recreate your performance problem. If you see CPU usage consistently at 100%, you are CPU bound.

The following screen shot shows the Performance Monitor during a CPU-bound test (CPU usage is the red line).



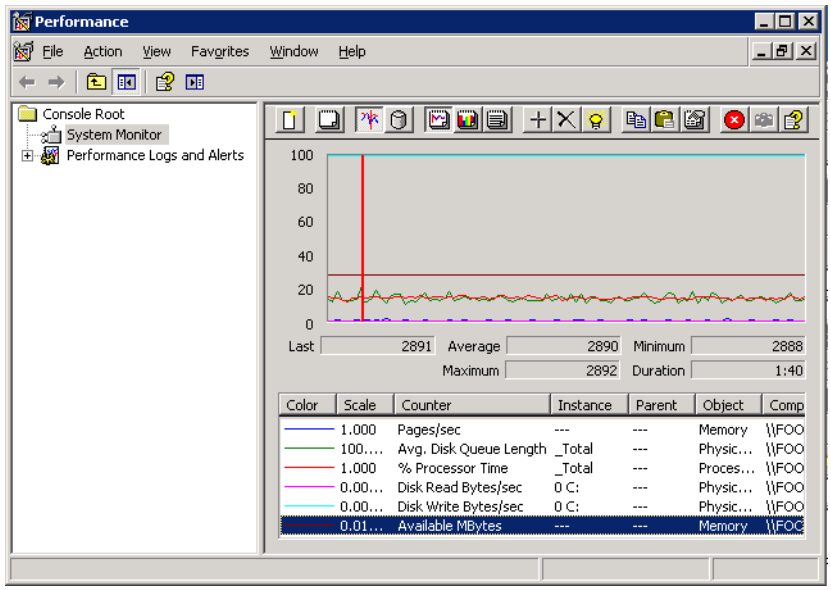
If it appears that you are CPU-bound, see "What to do if you are CPU bound" on page 39.

I/O bound

Run your test with one instance of InDesign Server, and watch the disk-queue length; this is the number of disk requests waiting in the queue. In an ideal world, this always is zero, as each disk request is completely serviced before the next one arrives. In the real world, however, this rarely occurs. Whenever the disk-queue length is greater than 0, some task is I/O-bound, as it is waiting on the disk. Do not be alarmed when you see a queue length greater than zero, as it is common to be I/O bound for short periods of time. Consistent I/O-bound situations, however, are more problematic and should be addressed.

Run your test again, using multiple instances of InDesign Server, and watch the Performance Monitor while you recreate your performance problem. If the disk-queue length is consistently above 0, you are I/O bound.

The following screen shot shows the Performance Monitor during an I/O-bound test. The disk-queue length is the green line running at around 20.



Note in the preceding test how low CPU usage is, as shown by the red line running also at about 20. In a test that exhibits this behavior, a more powerful processor will not enhance scalability. Note also that we are tracking the disk write rate on the light blue line. This is the number of bytes written to the disk per second. The disk-write rate can help you determine whether you are I/O bound in cases when the disk-queue length is not consistently high.

To chart additional metrics, right-click on the graph and choose "Add Counters...".

For each disk, there is a maximum read/write rate. While it is easy to watch current read/write rates using the Performance Monitor, it is harder to determine your maximum read or write rate. Typically, the I/O bottleneck is writing to disk. The SDK includes a tool called *maxwrite* that continuously writes to disk, enabling you to determine your max write rate. While running maxwrite, watch the max write rate on the Performance Monitor; this is your disk's maximum write rate. Stop maxwrite, run your InDesign Server test, and continue watching the write rate. If you are hitting the maximum write rate while running InDesign Server, you are I/O bound.

If either of these tests show that you're I/O-Bound, see "What to do if you are I/O bound" on page 37.

Read "Maximum disk-write tool" on page 42 for more information on maxwrite.

Mac OS tools

The best available tool depends on which version of OS X you are using:

- For OS X 10.5 (Leopard), a highly customizable performance tool, Instruments, is available from the Apple Developer Tools.

  For information on Instruments, go to: http://developer.apple.com/technology/tools.html

- In OS X 10.4, the Activity Monitor suffices as a tool to show you whether you are CPU bound or I/O bound.

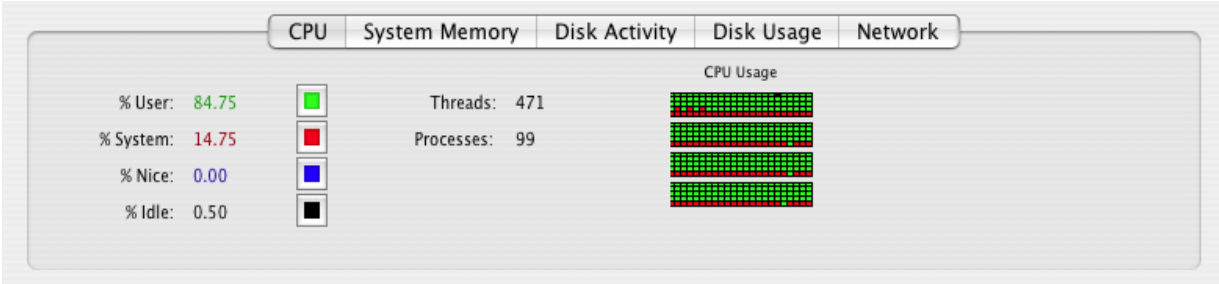
  In OS X 10.4, the Activity Monitor (usually located at Applications/Utilities/Activity Monitor) has a small chart at the bottom that can show details of both CPU and disk usage. There is a series of buttons just above the chart, two of which are CPU and Disk Activity.

CPU bound

Run your test with one instance of InDesign Server, and watch the Activity Monitor while the test runs. In the Activity Monitor, click the CPU button, and watch the peaks of each CPU's usage on the chart while the test runs. Also watch the %Idle number on the left-hand side. Think about seeing N times the CPU usage as you run N instances of InDesign Server. Is there enough idle CPU to handle that? For example, if you have a machine with four single-core processors and you plan to run five instances of InDesign Server, are there any times during your test where the %Idle was under 80%? If so, those are times when the scaling of InDesign Server would be limited by the CPU—when you would be CPU bound.

Run your test again using multiple instances of InDesign Server, and watch the Activity Monitor while you recreate your performance problem. If you see CPU usage consistently at 100%, you are CPU-bound.

The following screen shot shows the Activity Monitor during a CPU-bound test.



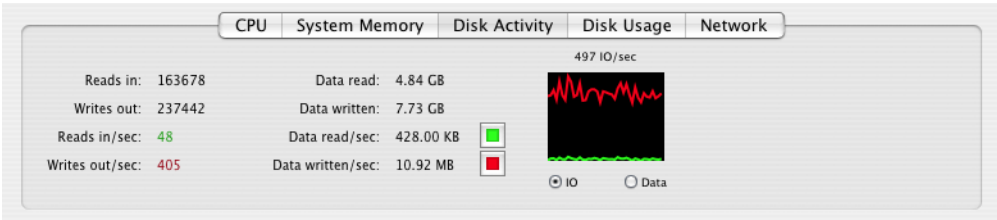
If it appears you are CPU bound, see "What to do if you are CPU bound" on page 39.

I/O bound

Run your test again with one instance of InDesign Server, and watch the Disk Activity. Note the red line in the chart, which is the number of writes per second that the disk is executing. Also watch the values just to the left of the chart at the bottom, which show the data-read and data-write rates. You can chart the read/write rates by clicking the Data radio button below the chart.

Run your test again, using multiple instances of InDesign Server, and watch the Activity Monitor while you recreate your performance problem. If you see consistently high disk activity, you may be I/O bound.

The following screen shot shows the Activity Monitor during an I/O bound test.



To be certain that you are I/O bound, we need to compare this activity rate with your maximum read/write rates. For each disk, there is a maximum read/write rate (which may be much less than the disk's advertised transfer rate). While it is easy to watch current read/write rates using Activity Monitor, it is harder to determine your maximum read or write rate. Typically, the I/O bottleneck is writing to disk.

The SDK includes a tool called *maxwrite* that continuously writes to disk, enabling you to determine your max write rate. While running maxwrite, watch the write rate on the Activity Monitor; this shows your maximum write rate. Stop maxwrite, run your InDesign Server test, and continue watching the write rate. If you are hitting the maximum write rate while running InDesign Server, you are I/O bound.

If it appears you are I/O bound, see "What to do if you are I/O bound" on page 37.

What to do if you are I/O bound

Being I/O bound means I/O performance is the limiting factor in your system's scalability. If you are experiencing scalability problems, and the preceding tests show that you are I/O bound, your attempt to scale has been thwarted by the performance limits of your disk. This is not uncommon when scaling up any system on a single machine, and often it is due to disk writes. This happens because the disk can write only one piece of data at a given time. When two simultaneous tasks request that the disk write data at the same time, one must wait until the other completes. That wait time means one instance will not perform the task as fast as the other, so two simultaneous instances do not exhibit throughput equal to twice a single instance. This is a scalability problem.

Each topic in this section examines a way of solving scalability problems involving disk writes. You may find that you need to combine more than one method to get a complete solution.

Multiple disks

Using multiple, independent disks eliminates waits on a single disk, which can eliminate an I/O scalability problem entirely.

When two instances of InDesign Server are having performance problems because they are competing for the same disk or other resource, one solution is to give them separate resources; that is, give each instance of InDesign Server its own disk. Make sure the inputs and outputs for instance #1 are located on disk #1, the inputs and outputs for instance #2 are located on disk #2, and so on for each of your N instances. You can do this by either adding N disks to a single machine or horizontally scaling your system across N machines, each with its own disk.

It may sound like either approach will solve the problem of being I/O bound entirely; unfortunately, that is not the case.

The first approach—adding N disks to a single machine—uses N disks but only one operating system, which resides on one disk. Certain locations in the operating system's footprint are used as locations for intermediate files that InDesign Server uses while processing a task. Even if you configured your single-machine system so N instances take their inputs and outputs from N different disks, all N instances still use the same disk for their intermediate files. So, while this approach will help alleviate the problem, it may not solve it entirely. The resolution depends on what your task asks InDesign Server to do, and how many intermediate files are involved in that task.

The second approach—horizontal scaling—involves N separate operating systems, each on its own disk. The InDesign Server instances will use completely independent I/O channels, which should eliminate the problem entirely. You will no longer have multiple, simultaneous requests for the same disk, so there should be no additional wait time, regardless of how many instances you run.

Horizontal scaling is the only solution discussed here that eliminates the problem entirely. All other solutions presented below minimize (but do not eliminate) the impact of multiple simultaneous requests.

NCQ/TCQ (native command queueing / tagged command queuing)

Using a disk with NCQ/TCQ decreases the performance degradation of incremental instances by minimizing the seek time between requests.

As mentioned above, the disk can write to only one location on disk at a time, so if there are two simultaneous requests, one must wait. If there are N simultaneous requests, N-1 must wait, forming a request queue. When a task has waited through the queue and is ready to be serviced, the first thing the disk must do is move the write head to the appropriate location on disk. This may be close to or far from the current location of the read/write head, depending on the previous task that was serviced. The amount of time it takes to move the read/write head is called the *seek time*.

In some disks, the queue is first-in, first-out, like a bank-teller line. Other disks are more like an elevator: everyone in the queue gets on, they all tell the elevator where they want to go, and the elevator services them in the most efficient way possible, usually making only one trip up the building to service all N people in the queue. SATA disks that service requests more like an elevator have an NCQ feature, while parallel ATA disks have a TCQ feature. Using a disk with an NCQ/TCQ feature will minimize the seek time, thus enabling multiple, simultaneous requests to be handled faster. This should decrease the performance degradation you see when running multiple instances.

RAID (redundant array of inexpensive disks)

Using a RAID can dramatically increase the performance of each task, minimizing the wait time of incremental, simultaneous requests.

There are many ways to configure a RAID array. The configurations involve *striping* data across the array, *mirroring* data across the array, or a combination of both striping and mirroring.

With striping, each chunk of data is split into N parts (where N is the number of disks in the array), and one-Nth of the data is written to each disk. The advantage of a striping configuration is performance: because the disks work simultaneously, writing data to a striped array is faster than writing the same data to one disk because each disk in a striped array has less data to write. In theory, a two-disk striped RAID should write twice as fast as the equivalent single disk (although there is some overhead for the RAID controller to split the data).

With mirroring, all data is written to all disks. The advantage of mirroring is redundancy: if any single disk in the array goes bad, the RAID controller can automatically switch to a mirrored disk.

If you have more than two disks in your RAID, you can have the advantages of both striping and mirroring. For example, in a three-disk RAID, you can have the data striped against two disks, and have the third disk record parity bits from a calculation based on the other two disks. That way, your data will get written twice as fast as with a single disk, and you can still tolerate losing either of the striped disks in the array. If you lose a disk, the RAID controller can automatically calculate the data that was on that disk, using the parity disk.

If you already use a striped RAID and still see I/O-bound scalability behavior, it may be worthwhile to adjust your RAID *stripe size*. The stripe size is the size of the data chunks that the RAID controller sends to each disk. How that stripe size affects performance depends on the size of each request coming into the controller (via the operating system). On Windows, larger stripe sizes generally work better for InDesign Server.

If you do not yet use a RAID configuration and you switch to something like a four-disk RAID configuration involving three striped disks and one parity disk, you may improve each task's performance so much that I/O is no longer your limiting factor, thus solving your I/O-bound scalability problem.

Caching

Using a cache can increase the performance of each task, minimizing the wait time of incremental, simultaneous requests.

RAID controllers (and sometimes disks) can include a write cache. When there is space available in the cache, a task's write request can be serviced almost immediately because the RAID controller can respond to the request by saying the data was written successfully as soon as the data is stored in the cache. The task does not have to wait for the data to be written to disk.

This is a risky approach, however, because the response happens before the disk actually does the write. For example, if a power outage occurs after the response but before the disk write, the task proceeds as if the data were written, but because the data is only in the cache, it may have been lost when the power went out. Fortunately, most vendors include a battery backup when they include a write cache, allowing any pending write requests to complete while using the battery power.

A write cache is an important tool to improve I/O performance; in some cases, it can improve I/O performance to the point where it is no longer a bottleneck.

Disk-rotation speed

Using a disk with a higher rotation speed can increase the performance of each task, minimizing the wait time of incremental, simultaneous requests.

The rotation speed of your disk affects the performance of that disk for each task. Currently, better enterprise drives spin at 15,000 RPM. If you run your test on a drive with a spin rate of 7,200 RPM or lower, you could see substantial performance improvements by switching to a faster drive.

Bus architecture

Using a disk with a fast bus architecture can increase the performance of each task, minimizing the wait time of incremental, simultaneous tasks.

There are three primary bus architectures for transferring data to disk: SATA (Serial ATA, or Serial Advanced Technology Attachment), SCSI (Small Computer System Interface), and SAS (Serial-Attached SCSI). In general, SCSI and SAS perform better than SATA, as they have lower seek times and higher transfer rates. If you are not using a SCSI or SAS drive, you may see increases in performance by switching.

Summary

The source of I/O-bound scalability problems is multiple simultaneous requests to the disk. Alleviating those simultaneous requests is possible only by having multiple separate disks, which is not an option for many hardware configurations. The other solutions presented in this section increase I/O performance. By implementing one or more of these other solutions, you may be able to increase your I/O performance enough so that it no longer limits your system's scalability.

What to do if you are CPU bound

If it appears your test is CPU-bound, there are several things you need to do:

- Understand the scalability expectations of InDesign Server and CPU-bound tasks.

- Dive deeper into exactly what your test is doing. Think about ways you could make it more efficient.
- Consider your hardware options.

These are discussed in this section.

Scalability expectations of InDesign Server

Usually, scalability of CPU-bound tasks is very predictable. For a single-threaded process like InDesign Server, scalability should be predictable up to the number of CPUs in your system. Each instance of InDesign Server can execute one task on one CPU at a time. For a CPU-bound task, maximum throughput is at N instances of InDesign Server, where N is the number of CPUs in the system.

Other investigations have found that, in general, increasing the number of InDesign Server instances to N+1 (where N is the number of cores on the machine) provides maximum throughput. This is true for most tasks, because most tasks rely on a mix of I/O and CPU to do their work. While one task is accomplishing I/O, another simultaneous task can use the CPU. For CPU-bound tasks, however, anything beyond N instances will not increase throughput, as each instance consumes an entire CPU.

Understanding your test

If you are not seeing the predictable scalability noted in the previous section, and if you are seeing CPU-bound symptoms in your tests, it is important to understand exactly what you are asking InDesign Server to do. Take a more detailed look at the script and/or Java®/CORBA code that you are sending to InDesign Server, and understand each individual request. As you do that, keep in mind the following topics.

General software design

Is your software as efficient as possible? As with any software, the quality of your code can have a big impact on performance. Here are a few things to consider.

If you detect that your code is CPU heavy, examine your algorithms for efficiency. If memory is a problem, consider caching options or processing smaller chunks of memory at a time. If I/O is a problem, look for places where you can limit the number of read/writes from disk. Are you using high-resolution images? If so, consider using low-resolution placeholders until content is final.

Efficiencies across multiple requests

Making your requests to InDesign Server more efficient can decrease the total CPU usage of each task, decreasing the chance that CPU is the limiting factor in your system.

As you read through the series of requests that you are sending to InDesign Server, think about whether they could be made more efficient. For example, are you searching through hundreds of page items so you can alter a given rectangle, when it would be simpler to alter that rectangle earlier, when there are many fewer page items? The point is to make your task as efficient as possible to minimize the total amount of CPU you use at any point in time.

Adobe Photoshop files and multithreading

While InDesign Server is written in a single-threaded model, there are certain libraries within InDesign Server that are multithreaded. One such library is Fargo, which handles compatibility with Adobe Photoshop® files. Do your tests use Photoshop files? If so, run your test again, and watch CPU usage closely using the tools and methods discussed earlier. At any time during your test, are you seeing InDesign Server use more than 100/N% of the CPU (where N is the number of cores on your system)? For example, if you have 4 CPUs on your machine, do you ever see InDesign Server use more than 25% of the CPU? If so, you are seeing multithreaded behavior.

Multithreaded environments do not scale like single-threaded environments. The expected behavior of a multithreaded environment depends on your exact test. To generate expectations for the scalability of your test, you must conduct multiple tests on your machine using different numbers of instances, and record their averages over time.

Hardware options

In some cases, your code is as efficient as it can possibly be, you're seeing that InDesign Server is showing no signs of multithreaded behavior, and you're still not seeing the predictable scalability you expect. Since you're seeing no evidence of I/O bound behavior, it's likely that this is because some other shared component of the system is your limiting factor (for example, memory bandwidth on a multiple-core machine). Using multiple independent machines will eliminate any dependence on shared resources, which should eliminate your scalability problem entirely. You could also try using a machine with faster processors, as that may reduce the CPU usage enough so that it no longer limits your scalability.

InDesign Performance Metrics API

InDesign and InDesign Server contain a scriptable performance counter architecture. This API allows you to monitor not only system metrics such as CPU and I/O, but also allows you to add custom performance counters in your components. This gives you the ability to monitor aspects of your own code or of InDesign's. The InDesign performance metrics are exposed to PerfMon on Windows, and DTrace and Instruments on Macintosh.

For detailed information about the performance metrics API, and how to view the metrics in PerfMon and DTrace, read "Performance Metrics" in the *Adobe InDesign Plug-In Programming Guide*.

Performance-testing tools

In addition to the performance-metrics API, the InDesign Server SDK includes performance-testing and benchmarking tools. These tools comprise automated tests that simulate the behavior of InDesign Server during general workflow tasks, high-CPU tasks, and high-I/O tasks. The tools include the following:

- maxwrite — Used to find your disk's maximum write rate.
- maxread — Used to find your disk's maximum read rate.
- Benchmark toolkit — Used to benchmark particular InDesign Server operations or workflows over time.

Maximum disk-write tool

The SDK includes a command-line tool called maxwrite that can be used to determine your disk's maximum write rate. The tool is located at <*IDS SDK*>/tools/disktools. On Windows, there are both 32-bit and 64-bit executables (maxwrite.exe and maxwritex64.exe). On Mac OS, there is one Unix executable (maxwrite).

There also is source code and a project file for the tool, in case you need to modify it. Look in <*IDS SDK*>/tools/disktools/build/<win|mac>.

To use maxwrite:

1. Copy the maxwrite application to a temporary location on the disk that you want to test.
2. When you run maxwrite, make sure the current working directory is on the disk you want to test. maxwrite creates a file in the current working directory and continually writes to the file.
3. Run maxwrite from a console window (Windows) or Terminal window (Mac OS). maxwrite takes no arguments.
4. Watch the Disk Write Bytes/sec using the Performance Monitor (Windows) or Activity Monitor (Mac OS). The write rate displayed for maxwrite is a maximum disk-write rate. Compare this rate to the write rates of tests using InDesign Server, to see whether InDesign Server is I/O bound.
5. To stop maxwrite, use Ctrl+C.

NOTE: When you finish using maxwrite and maxread, delete the temporary directory where you copied it, as maxwrite creates a large file in that directory.

Maximum disk-read tool

The SDK includes a command-line tool called maxread that can be used to determine your disk's maximum read rate. The tool is located at <*IDS SDK*>/tools/disktools. On Windows, there are both 32-bit and 64-bit executables (maxread.exe and maxreadx64.exe). On Mac OS, there is one Unix executable (maxread).

There also is source code and a project file for the tool, in case you need to modify it. Look in <*IDS SDK*>/tools/disktools/build/<win|mac>.

To use maxread:

1. Run maxwrite. Since maxread reads from the large file created by maxwrite, you must first run maxwrite to create this file.
2. Copy the maxread application to the directory containing the maxwrite output file. When you run maxwrite, make sure the current working directory is on the disk you want to test.
3. Run maxread from a console window (Windows) or Terminal window (Mac OS). maxread takes no arguments.
4. Watch the Disk Read Bytes/sec using the Performance Monitor (Windows) or Activity Monitor (Mac OS). The read rate displayed for maxread is a maximum disk-read rate. Compare this rate to the read rates of tests using InDesign Server, to see whether InDesign Server is I/O bound.
5. To stop maxread, use Ctrl+C.

Benchmarking tools

The benchmarking tools consist of a set of JavaScripts that perform operations that are representative of InDesign Server workflows, additional JavaScripts to drive test cases, and examples of how to run the benchmark on multiple instances. Because assets and operations vary greatly from system to system, these benchmarks may or may not be like your workflow. To make the benchmarks completely relevant for your system, you can write and exercise test cases that more closely resemble your assets and operations.

All the tests are executed in a similar way. RunTest.jsx (in the scripts folder) is the driving code or entry point. It relies on several files in the Includes directory to execute tests, record metrics, and write logs. RunTest.jsx is called with test, duration, and sample-length parameters. It executes a test continually for the designated duration, pausing to record statistics at the specified sample periods. The output includes the number of times the test is executed, the average time it took to complete the test, and how the CPU was utilized during the test.

Benchmarking files in the SDK

Content for the benchmark tests is located in the InDesign Server SDK in the <*IDS SDK*>/tools/benchmarks folder. This folder contains subfolders, described below.

BenchmarkChartsConsole

This folder contains the Benchmark Charts Console, an Adobe AIR application that can be used to display benchmark data.

IDServerBenchmarks

After following the setup instructions, you should have an IDServerBenchmarks folder at the root of the drive containing InDesign Server. The folder should contain a subfolder called TestFiles, containing a data folder (named using the port number) for each InDesign Server instance. Each folder contains a copy of the InDesign image and data files used by the benchmark tests.

IDServerFolder

This folder contains sample scripts that demonstrate how to execute the benchmarks concurrently on multiple instances with sampleclient.

ScriptsFolder

After following the setup steps, your InDesign Server scripts folder contains the contents of the ScriptsFolder, comprising the following files and folders:

- RunTest.jsx — The driver script used to execute a test case.
- Includes — Contains Java Script include files used to run tests, record metrics, and log results.
- BenchmarkTests — Contains the actual test scripts.

Setting up the benchmark tests

Running the benchmarks requires modest setup: copying data and script folders to expected locations, and duplicating some files for each instance of InDesign Server you will be testing.

You will find necessary files in the <*IDS SDK*>/tools/benchmark folder. To set up the benchmarks:

1. Decide how many instances of InDesign Server you will test.
2. The benchmarks are written to communicate with the server using SOAP. Determine the port number for each InDesign Server instance.
3. Copy the IDServerBenchmarks folder to the root of each drive containing an InDesign Server installation. For example, if your copy of InDesign Server is installed in C:\Program Files\Adobe, copy the folder to C:\. You would then have a C:\IDServerBenchmarks folder. If your installation is on another drive, copy your files to that drive.
4. Each instance of InDesign needs a set of documents and data files. Navigate to the IDServerBenchmark/TestFiles folder. (This is in the folder in the IDServerBenchmark that now exists at the root of your drive.) Duplicate the PortNumber folder once for each instance of InDesign Server you will test. Name the new folders using the port numbers for each instance of InDesign Server that will be executed on that drive.
5. Copy the contents of the ScriptsFolder to the Scripts folder in each InDesign Server installation. For example, if your installation is installed on C:\Program Files\Adobe\Adobe InDesign CS6 Server, copy the contents of the ScriptsFolder to C:\Program Files\Adobe\Adobe InDesign CS6 Server\Scripts.
6. Copy the contents of the IDServerFolder to your InDesign Server installation. For example, if your installation is installed in C:\Program Files\Adobe\Adobe InDesign CS6 Server, copy the contents of IDServerFolder to C:\Program Files\Adobe\Adobe InDesign CS6 Server. That should leave you with a benchmarks folder (C:\Program Files\Adobe\Adobe InDesign CS6 Server\benchmarks).

NOTE: If you forget to create instance-specific data files or make a mistake naming the folder, the benchmark tests will report file errors in the InDesign Server console, and your throughput likely will be 0 for that instance of InDesign Server.

Running a benchmark on a single instance

Running a benchmark on a single instance is fairly straightforward. You need to launch the instance of InDesign Server using a port number that matches the data file you set up (as described above).

Use InDesign Server's sampleclient program (a command-line utility that executes a script on the server via SOAP) to execute RunTest. The script expects three parameters:

- A string describing the test case.
- The test-execution duration.
- The sample length.

All times are expressed in milliseconds. For example, if the server is running on the local host and listening to port number 12345, and you want to run the PeriodicTable test for 10 minutes, with 1-minute sample intervals, you would run sample client as follows:

```
sampleclient -host localhost:12345 scripts/RunTest arg1=PeriodicTable arg2=600000
arg3=60000
```

Running a benchmark on multiple instances

Benchmarking multiple instances is slightly more involved. You must launch each instance of InDesign Server that you will test. Each test case needs to be executed concurrently across all InDesign Server

instances being tested. The SDK includes scripts that demonstrate two ways to call sampleclient concurrently.

The first approach is very simple, but it applies only to Windows. It uses the start command to launch a new window and process before executing sampleclient. For example, the following commands run the benchmarks on 4 instances (running on port 12345, 23456, 34567, and 45678) concurrently for 3 minutes:

```
start "12345" cmd /C sampleclient.exe -host localhost:12345 scripts\RunTest.jsx
   arg1=PeriodicTable arg2=180000 arg3=30000
start "23456" cmd /C sampleclient.exe -host localhost:23456 scripts\RunTest.jsx
   arg1=PeriodicTable arg2=180000 arg3=30000
start "34567" cmd /C sampleclient.exe -host localhost:34567 scripts\RunTest.jsx
   arg1=PeriodicTable arg2=180000 arg3=30000
start "45678" cmd /C sampleclient.exe -host localhost:45678 scripts\RunTest.jsx
   arg1=PeriodicTable arg2=180000 arg3=30000
```

Four windows, titled by their respective port numbers, are launched. Each window calls InDesign Server via sampleclient. All four windows close when the benchmark test is completed, in approximately 3 minutes (30000 milliseconds).

For an example of how to use this approach from within a bat file, see <*IDS SDK*>/tools/benchmark/idserverfolder/benchmarks/runPeriodicTable_4_Instances.bat. The primary difference is that the quotation marks in the window-title argument for the start command need to be escaped. The paths also expect the bat file to be run from a child folder in the InDesign Server installation directory. To run this on your machine, you need to ensure that the port and instance numbers match your configuration. The script can be executed via the command prompt or Windows explorer.

The second approach demonstrated in the SDK is slightly more sophisticated and cross platform. It uses a Perl script to fork a new process for each InDesign Server Instance. See <*IDS SDK*>/tools/benchmark/idserverfolder/benchmarks/runPeriodicTable_4_Instances.pl. The test name, port numbers, duration, and sample period are variables at the top of the file. To run this on your machine, you need Perl (standard on Mac OS). Also, you need to ensure that the variables match your configuration.

There are other ways to execute sampleclient concurrently; these are included, for your convenience. You can use one of these approaches to run other tests, including your own custom benchmarks.

Locating and understanding result files

Upon execution, results are logged to the IDServerBenchmarks/Logs folder. If this folder does not exist, it is created when a benchmark test is executed. Each instance on which a test is run produces three data files. For example, if you run the PeriodicTable test on two instances, using ports 12345 and 23456, the following data files are written to IDServerBenchmarks/Logs:

- PeriodicTable_12345.csv
- PeriodicTable_12345.txt
- PeriodicTable_12345.xml
- PeriodicTable_23456.csv
- PeriodicTable_23456.txt
- PeriodicTable_23456.xml

The port number for the instance on which the test executed is included in the filename. The three file types (CSV, TXT, and XML) are alternate formats for the same data.

The data contains one entry for each sample executed. For example, if the test is run for 30 minutes with 5-minute sample intervals, there should be 6 entries. Each entry contains the following statistics:

- CPU usage.
- Percent used for system operations.
- Percent used for this process.
- Minimum execution time for a test case.
- Maximum execution time for a test case.
- Average time for all test cases.
- Actual sample time. This may be slightly greater than the specified sample period, because test cases are allowed to finish before the sample is taken.
- Number of actions (or throughput). This is the number of times the test executes during the sample period.
- Number of errors.

Throughput is the number of times a test runs on an instance. This is recorded in the <num_actions> element in the XML output file, and as a column in the CSV output file. For details, see the file header.

Total throughput is the total number of times a test runs across all instances. This is calculated by adding the timings from the instance-specific data files. This allows you to compare the effectiveness of adding additional instances of InDesign Server.

Adobe-provided tests

The benchmarks include several built-in tests. Most of the benchmarks tests record metrics for a narrow set of operations. We call these *workflow tests* because they allow you to test specific parts of a workflow. A few tests record metrics for an entire script; in this case, the script is considered an atomic operation, so we call these *atomic tests*.

The built-in test cases are described in the following table. The names in the Test Name column are the parameters that can be passed as *arg1* to RunTest.jsx. For information on executing tests, see "Running a benchmark on a single instance" on page 44 and "Running a benchmark on multiple instances" on page 44.

Test Name	Description
All	Runs all the tests in this table. The list of tests is hard coded in ServerBenchmarkTests.jsxinc.
ApplyEffect	Opens Document.indd; cycles through all the links; applies rotation angle, vertical scale, drop shadow, and emboss to each one; then saves to EffectDocument.indd and closes the indd file. Metrics are recorded for the entire duration of the script.
AutoflowText	Creates a document, places AutoflowText.txt with autoflow turned on, then closes the document without saving. Timing metrics are recorded only for the place segment.

Test Name	Description
BuildPage	Creates a document, sets some document preferences, places a high-resolution Photoshop file (/Links/Adobe-006468-cover.psd), adds a text frame, fills the text frame with placeholder text, saves as BuildPage.indd, and closes the document. Metrics are recorded for the entire duration of the script.
ComposeTextMulti	Creates a document and inserts placeholder text into a full-page text frame with the Adobe Paragraph Composer text composer applied. Timing metrics are recorded only for the composer segment.
ComposeTextSingle	Creates a document and inserts placeholder text into a full-page text frame with the Adobe Single-line Composer text composer applied. Timing metrics are recorded only for the composer segment.
ExportDocumentIDML	Opens Document.indd and exports as IDML to DocumentIDML.idml. Timing metrics are recorded only for the export segment.
ExportDocumentJPG	Opens Document.indd and exports as JPG to DocumentJPG.jpg. Timing metrics are recorded only for the export segment.
ExportDocumentPressPDF	Opens Document.indd and exports it to PDF using the [Press Quality] option. Timing metrics are recorded only for the export segment.
ExportDocumentSmallPDF	Opens Document.indd and exports it to PDF using the [Smallest File Size] option. Timing metrics are recorded only for the export segment.
ExportDocumentXML	Opens Document.indd and exports it as XML to DocumentXML.xml. Timing metrics are recorded only for the export segment.
ExportSpreadIDML	Opens Spread.indd and exports as IDML to SpreadIDML.idml. Timing metrics are recorded only for the export segment.
ExportSpreadJPG	Opens Spread.indd and exports as JPG to SpreadJPG.jpg. Timing metrics are recorded only for the export segment.
ExportSpreadPressPDF	Opens Spread.indd and exports as PDF using the [Press Quality] option to SpreadPDF.pdf. Timing metrics are recorded only for the export segment.
ExportSpreadSmallPDF	Opens Spread.indd and exports as PDF using the [Smallest File Size] option to SpreadPDF.pdf. Timing metrics are recorded only for the export segment.
ExportSpreadXML	Opens Spread.indd and exports as XML to SpreadXML.jpg. Timing metrics are recorded only for the export segment.
Import	Imports and places the PhotoShop and TIFF files located in the Import folder. Timing metrics are recorded only for the import and place segment.
ImportDocumentXML	Opens XML_Document.indd and imports Document.xml. Timing metrics are recorded for the XML import segment only.
ImportSpreadXML	Opens XML_Spread.indd and imports Spread.xml. Timing metrics are recorded for the XML import segment only.
OpenCloseINDD	Open and closes Document.indd. Timing metrics are recorded for both the open and close operations.

Test Name	Description
OpenSaveIDML	Opens Document.idml and saves the temp file to IDML_Document.indd. Timing metrics are recorded for both the open and save (close) operations.
PeriodicTable	Runs the PeriodicTable script producing an InDesign document containing the periodic table of elements. Timing metrics are recorded for the entire operation.
script file path	If *arg1* is a path to a file on the file system, it runs the script as a test.

Running a custom atomic test

You can use the benchmark system to run a custom atomic test. To implement a test, you simply write a JavaScript file and execute it by passing the path to the file in *arg1* of RunTest. Such a test is performed as an atomic operation. Metric information is taken by RunTest.jsx. For examples of workflow tests, see the source for the PeriodicTable, BuildPage, and ApplyEffect tests.

```
sampleclient -host localhost:12345 scripts/RunTest
arg1="/MyTests/PerformMyOperations.jsx" arg2=600000 arg3=60000
```

Results are written to IDServerBenchmarks/Logs. The three output files are saved with filenames CustomScript_*PortNumber*. For example, the previous call would result in the following three output files:

- CustomScript_12345.csv
- CustomScript_12345.jsx
- CustomScript_12345.txt

Running a custom workflow test

Running custom workflow tests is not directly supported. The scripts themselves are not extensible in this way. To add a custom workflow test, you must edit the benchmark source code. This is reasonable if you need more precise measurements.

To begin, implement your test. Use an existing workflow test as a guide. The Import test is a good example to consider. As you approach the test, understand that the main difference between a workflow and atomic test is that timing metrics and logs are handled within the workflow tests.

ServerBenchmarkTests.jsxinc hard codes the set of available tests. You also must add your new test to this file.

Benchmark Charts Console

The BenchmarkChartsConsole is an Adobe AIR application that can be used to display benchmark data. You must build and export the application before running it. For an example of what the application looks like, see ScreenShot.jpg.

This section contains directions for building the application with Flex Builder and the free Flex SDK, and basic directions for using the application.

Building with Flex Builder

To import the application into Flex Builder:

1. Start Flex Builder.
2. Choose File > Import > Flex Project.
3. Select Project Folder and Browse to the BenchmarkChartsConsole folder.
4. Uncheck "Use Default Location," to import in place.
5. Click Next-> and follow the dialog to completion.

You should be able to run from the FlexBuilder debugger. If you encounter errors, you may need to change the runtime version in BenchmarkChartsConsole-app.xml to match your environment. We built with Flex Builder 3.0.2, targeting version 1.5 of the AIR Runtime.

To export an installable AIR bundle from Flex Builder:

1. Choose File > Export > Release Build...
2. Make any changes to output filenames and click Next>.
3. Choose "Export and sign an AIR file with a digital certificate."
4. Create a certificate using the Create button, or reuse an existing certificate.
5. Click Next>.
6. Click Finish.

Building with the Flex SDK

If you do not have Flex Builder, you can build this application with the free Flex SDK. The Flex SDK contains several tools that can be run from the command line to build and produce the AIR installable bundle. To build with the free Flex SDK:

1. Download the Flex SDK (we built with version 3.2), and add the <*IDS SDK*>/bin directory to your system search path.
2. From the terminal or command prompt, run this command:

   ```
   cd <IDS SDK>/tools/benchmark/BenchmarkChartsConsole/src
   ```

3. Run amxmlc to compile the mxml file:

   ```
   amxmlc BenchmarkChartsConsole.mxml
   ```

4. Run adt to produce a self-signed certificate. (Use your own password.)

   ```
   adt -certificate -cn TestCert 1024-RSA ..\sampleCert.p12 password
   ```

NOTE: The certificate should be created outside the src directory.

5. Run adt to produce an installable bundle. (Enter the password from step 4.)

   ```
   adt -package -storetype pkcs12 -keystore ..\sampleCert.p12
   ```

```
BenchmarkChartsConsole.air BenchmarkChartsConsole-app.xml .
```

Using BenchmarkChartsConsole

You can run the installer on any machine. After the program is installed, it should be available from the Start menu.

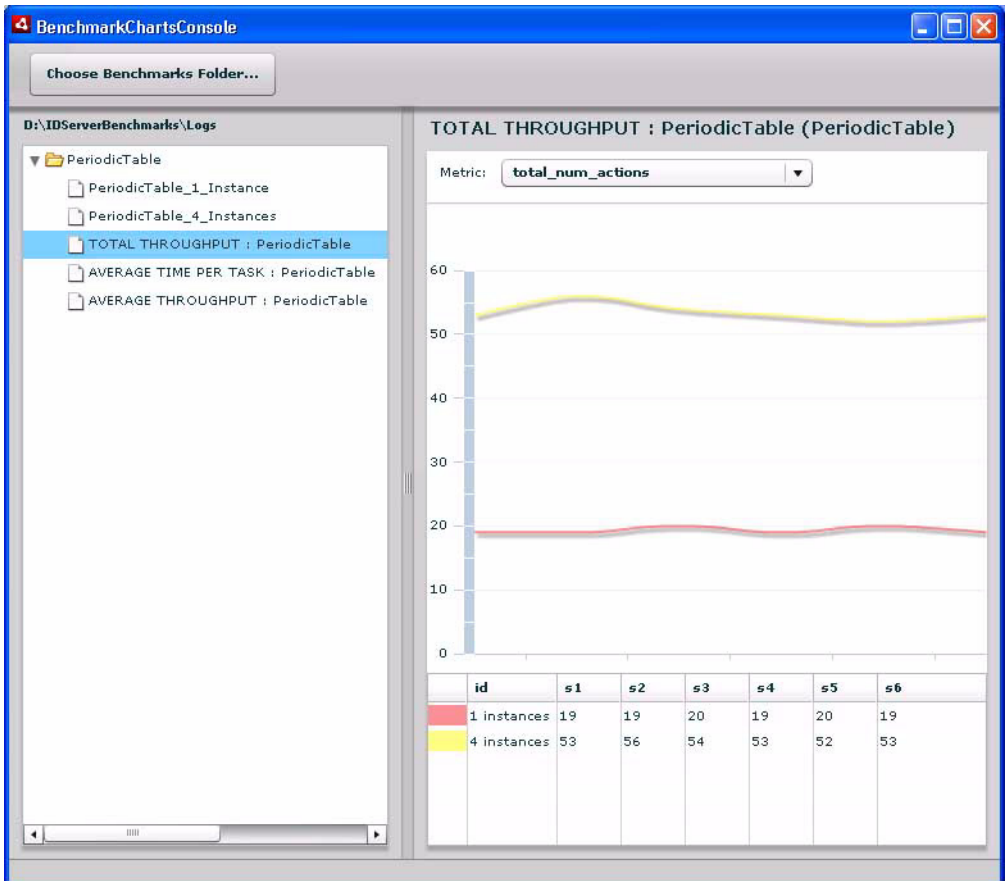
The application allows you to specify a data folder containing test executions from various runs of various benchmark tests. The tool expects the data folder to be organized as follows:

```
Data Folder
    Configuration_X
        Data Files
    Configuration_Y
        Data Files
```

For example, to compare execution on one- and four-instance configurations, the PeriodicTable test data files could be organized as follows:

```
Data Folder
    PeriodicTable
        PeriodicTable_1_Instance
            PeriodicTable_12345.csv
            PeriodicTable_12345.txt
            PeriodicTable_12345.xml
        PeriodicTable_2_Instances
            PeriodicTable_12345.csv
            PeriodicTable_12345.txt
            PeriodicTable_12345.xml
            PeriodicTable_23456.csv
            PeriodicTable_23456.txt
            PeriodicTable_23456.xml
            PeriodicTable_34567.csv
            PeriodicTable_34567.txt
            PeriodicTable_34567.xml
            PeriodicTable_45678.csv
            PeriodicTable_45678.txt
            PeriodicTable_45678.xml
```

The application allows you to examine metrics for each configuration that is present. Perhaps most meaningful, TOTAL THROUGHPUT allows you to compare the number of times the benchmark ran across all instances in each type of configuration. The following screen shot shows the tool comparing total throughput of the PeriodicTable test using one and four instances.


BenchmarkChartsConsole
Choose Benchmarks Folder...
D:\IDServerBenchmarks\Logs
PeriodicTable
PeriodicTable_1_Instance
PeriodicTable_4_Instances
TOTAL THROUGHPUT : PeriodicTable
AVERAGE TIME PER TASK : PeriodicTable
AVERAGE THROUGHPUT : PeriodicTable
TOTAL THROUGHPUT : PeriodicTable (PeriodicTable)
Metric:
total_num_actions
60
50
40
30
20
10
0
id s1 s2 s3 s4 s5 s6
1 instances 19 19 20 19 20 19
4 instances 53 56 54 53 52 53