



Cognitier SimpleIPC

Exercise 4 - Throttling

Version 1.0.0.1

April 21, 2009

Copyright © 2009 Cognitier, Inc. All rights reserved.

Cognitier and the Cognitier logo are trademarks of Cognitier, Inc. All other company and product names referred to may be trademarks of their respective owners. This documentation is copyrighted material and is intended exclusively for use by licensed users of Cognitier software. The information in this document is subject to change without notice.

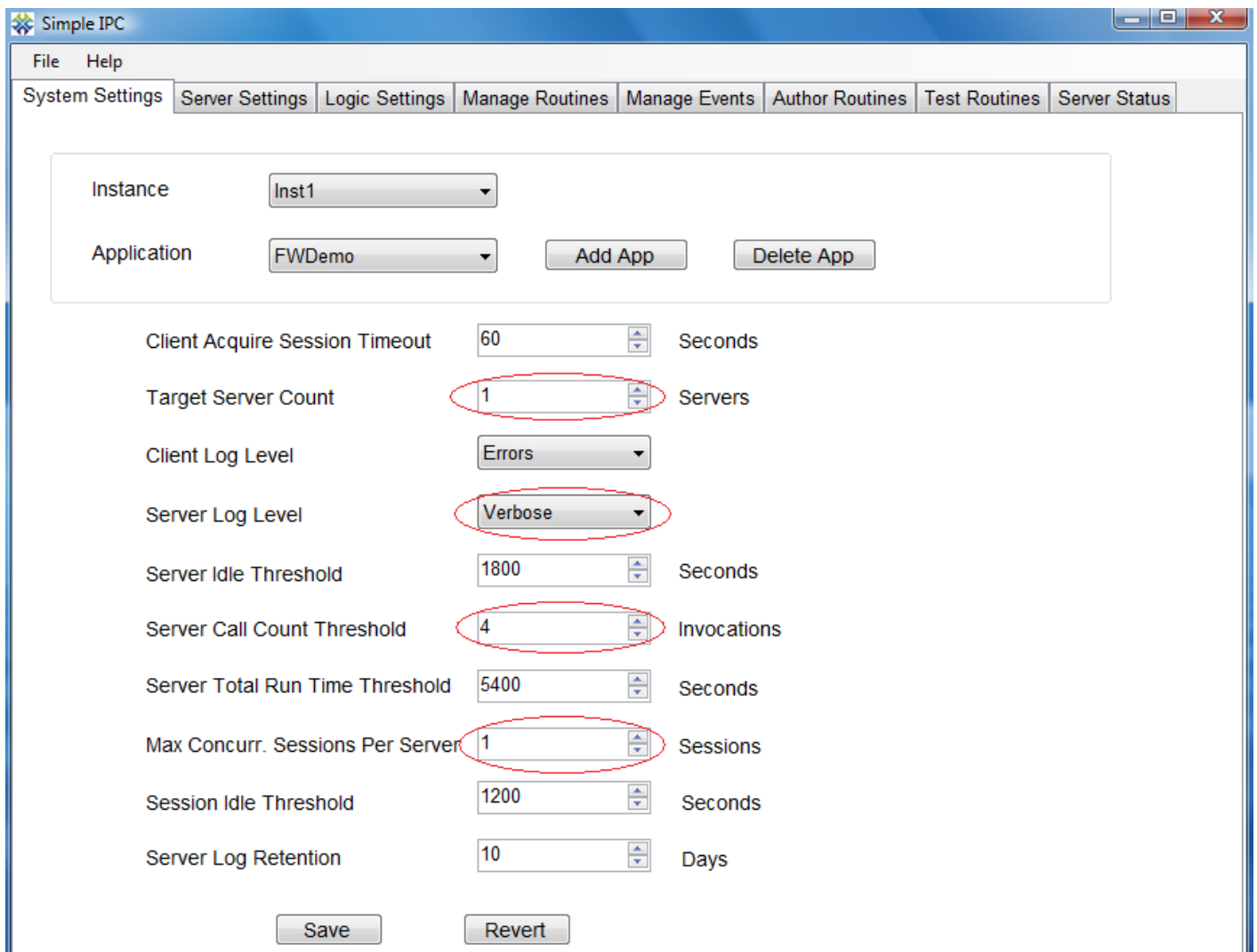
Exercise 4 - Throttling

The point of this exercise is to demonstrate how to use configuration to restrict concurrent access to sensitive back-end resources. Examples of such resources could be databases that only allow a limited number of connections or third-party systems that cannot handle a significant load. In this exercise, the resource we access will be a simple text file, and we will restrict access to one session at a time.

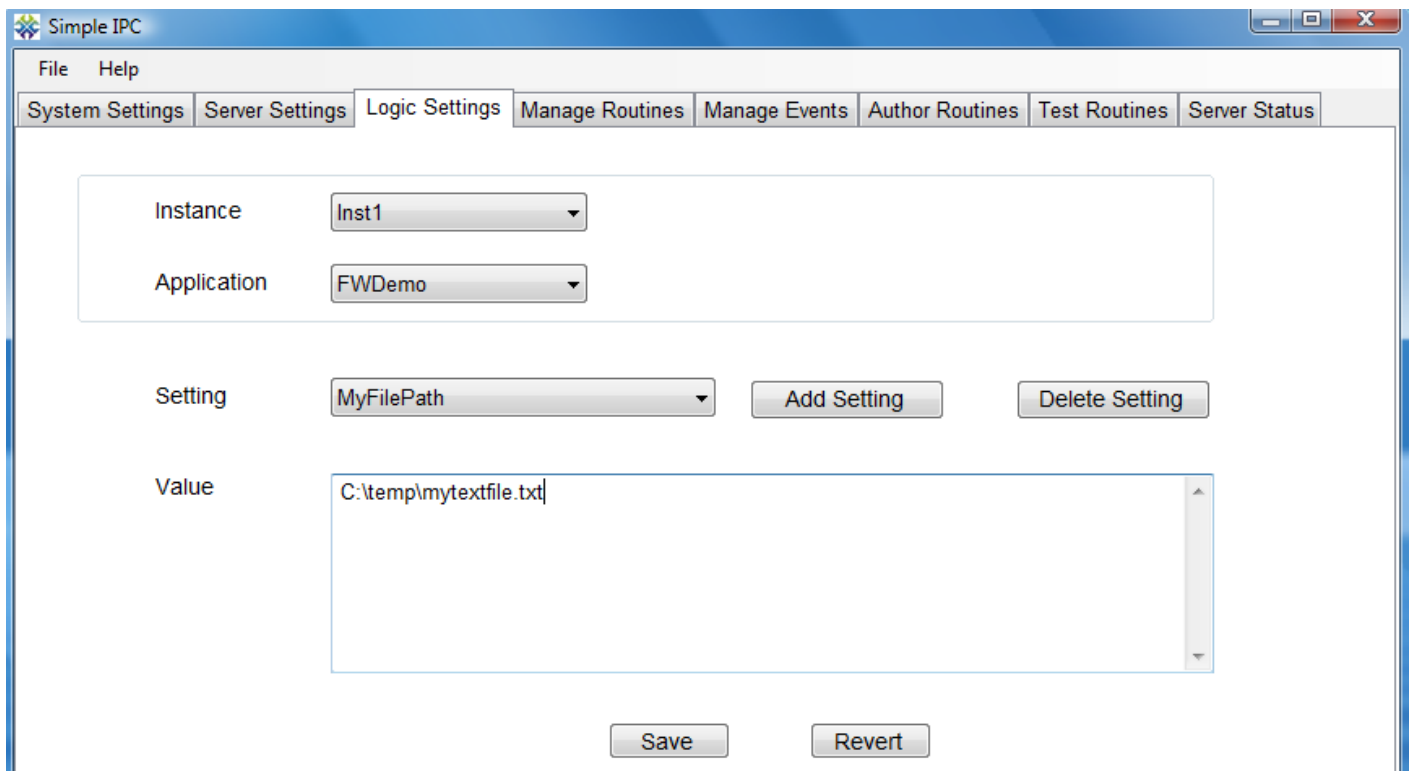
There is a complication in this exercise because we allow only one client to access the file at a time, but in accordance with the IPC server life cycle, a server might take a while to exit after it has met the criteria to transition to Due To Recycle status. When the server reaches Due To Recycle status, it no longer counts against the Target Server Count, and so another IPC server may be launched at that time. We do not want two IPC servers running at the same time for our Instance/Application combination, and so we will pro-actively terminate the first server process from within our session exit routine code. This is possible because we know that we only allow one session on the server and because we know what clean-up code must be run.

Within this example, we will also demonstrate calling into one IPC server from another IPC server. We will be placing a concurrent load on one IPC server, but only one session at a time will be able to call into the second IPC server.

Step 1: Go to the System Settings tab and create a new application and call it FWDemo. Set the server log level for this application to Verbose. Set the target number of servers to one and the maximum number of concurrent sessions to one. Set the maximum call count for the server to four (i.e. the server is expected to service four routine invocations before exiting - although we will see that in this exercise we will actually service just three invocations per server).



Step 2: Create a Logic Setting for the path and file name of the file to which we will be writing. Go to the Logic Settings tab and toggle the Instance selection from Inst1 to Inst2 and back to Inst1 in order to refresh the list of applications. Select the FWDemo application and create a new setting. For this exercise, we will call it "MyFilePath". We will provide a file path appropriate for our test environment and save the setting.



Step 3: Create the server start-up routine. Go to the Author Routines tab of the CTConsole. Create a new Source Code file. Call the file FWServStart, and select the BasicServerOp1 template. For this exercise, use the following code to create and open a StreamWriter object and place it in the server context.

```
import System;
import System.Collections;
import System.Reflection;
import System.IO;

[assembly: AssemblyVersion("1.0.0.0")]

public class FWServStart implements CTSHost.IBasicServerOp1
{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;
        var oSW: StreamWriter = null;

        oLogUtil = CTCommon.LogUtil.Instance;
        bRet = false;
    }
}
```

```

try
{
    //Get the file path Logic Setting
    var sFilePath: System.String = String.Empty;

    if (oServerContextAccessor.GetLogicSetting("MyFilePath") != null)
    {
        sFilePath = oServerContextAccessor.GetLogicSetting("MyFilePath");
    }
    if (sFilePath.length == 0)
    {
        throw new Exception("Unable to obtain file path and name");
    }
    oSW = new StreamWriter(sFilePath, true);
    oServerContextAccessor.SetServerContextItem("FileWriter", oSW);

    bRet = true;
}
catch (e)
{
    //Write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
}
return bRet;
} //end function
} //end class

```

Step 4: Register the routine we just created. Go to the Manage Routines tab and click the "Refresh List" button so that our new dll will appear in the dlls drop-down. Select FWServStart.dll, enter a routine name of FWServStart, and save it.

Step 5: Associate the routine we just created with the "On Server Start" event of the FWDemo application. Go to the Manage Events tab and again toggle the Instance selection in order to refresh the selections in the Applications drop-down. Select the FWDemo application. Select the FWServStart routine from the drop-down for the "On Server Start" event and click the Save button.

Step 6: Create the routine to be invoked by name by external clients. Go to the Author Routines tab of the CTConsole. Create a new Source Code file. Call the file FWBasicCall, and select the BasicRoutineRun1 template. For this exercise, use the following code to get the StreamWriter object out of the server context and write some output to the text file.

```

import System;
import System.Collections;
import System.Reflection;
import System.IO;

[assembly:AssemblyVersion("1.0.0.0")]

public class FWBasicCall implements CTSHost.IBasicRoutineRun1
{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor, oCallParamsAccessor:
CTSHost.CallParamsAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;
        var oSW: StreamWriter = null;

        oLogUtil = CTCommon.LogUtil.Instance;
        bRet = false;
        try
        {
            //Get the StreamWriter object out of the server context
            var htServerCtxtItems: Hashtable;
            htServerCtxtItems = oServerContextAccessor.GetServerContextItems();
            if (htServerCtxtItems["FileWriter"] != null)
            {
                //Use a StringBuilder to prepare a string to write to the text file.
                //Include the server process id and client session id so we can
identify who wrote
                //each string to the file.
                var sMsgBuilder: System.Text.StringBuilder = new
System.Text.StringBuilder();
                sMsgBuilder.Append(DateTime.Now.ToShortDateString());
                sMsgBuilder.Append(" ");
                sMsgBuilder.Append(DateTime.Now.ToLongTimeString());
                sMsgBuilder.Append(" Message written by server process ");

sMsgBuilder.Append(System.Diagnostics.Process.GetCurrentProcess().Id);
                sMsgBuilder.Append("; client session ");
                sMsgBuilder.Append(oSessionContextAccessor.SessionID);
            }
        }
    }
}

```

```

        oSW = System.Convert.ChangeType(htServerCtxtItems["FileWriter"],
StreamWriter);
        oSW.WriteLine(sMsgBuilder.ToString());
        oSW.Flush();
    }

    bRet = true;
}
catch (e)
{
    //Set errors into errors collection - must be strings
    oCallParamsAccessor.OutputErrors.Add(e.Message);

    //Also write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
}
return bRet;
} //end function
} //end class

```

Step 7: Register the routine we just created. Go to the Manage Routines tab and click the "Refresh List" button so that our new dll will appear in the dlls drop-down. Select FWBasicCall.dll, enter a routine name of FWBasicCall, and save it.

Step 8: Usually when we perform initialization in the server start-up routine, we define a server-exit routine to perform the un-initialization. In this case, we will be manually terminating the server process before the server-exit routine would fire. We will be defining a session exit routine. Go to the Author Routines tab of the CTConsole. Create a new Source Code file. Call the file FWSessionEnd, and select the BasicSessionOp1 template. For this exercise, use the following code to check if we are at our limit for the number of routine invocations for the server. If so, get the StreamWriter object out of the server context and close it, and then terminate the server process.

```

import System;
import System.Collections;
import System.Reflection;
import System.IO;

[assembly:AssemblyVersion("1.0.0.0")]

public class FWSessionEnd implements CTSHost.IBasicSessionOp1

```

```

{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;
        var oSW: StreamWriter = null;

        oLogUtil = CTCommon.LogUtil.Instance;
        bRet = false;
        try
        {
            //Check if the server is within one call of being at the call count limit.
            //oServerContextAccessor.ServerCallCount is the current cumulative number
of calls for the server.
            //oServerContextAccessor.ServerCallCountLimit is the System Setting for
the call count threshold
            if (oServerContextAccessor.ServerCallCount >=
(oServerContextAccessor.ServerCallCountLimit - 1))
            {
                //Get the StreamWriter out of the server context and close it.
                //Then terminate the host IPC server process.
                var htServerCtxtItems: Hashtable;
                htServerCtxtItems = oServerContextAccessor.GetServerContextItems();
                if (htServerCtxtItems["FileWriter"] != null)
                {
                    oSW = System.Convert.ChangeType(htServerCtxtItems["FileWriter"],
StreamWriter);
                    oSW.Close();
                    oSW.Dispose();
                    System.Diagnostics.Process.GetCurrentProcess().Kill();
                }
            }
            bRet = true;
        }
        catch (e)
        {
            //Write errors to the server log file
            oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
        }
    }
}

```

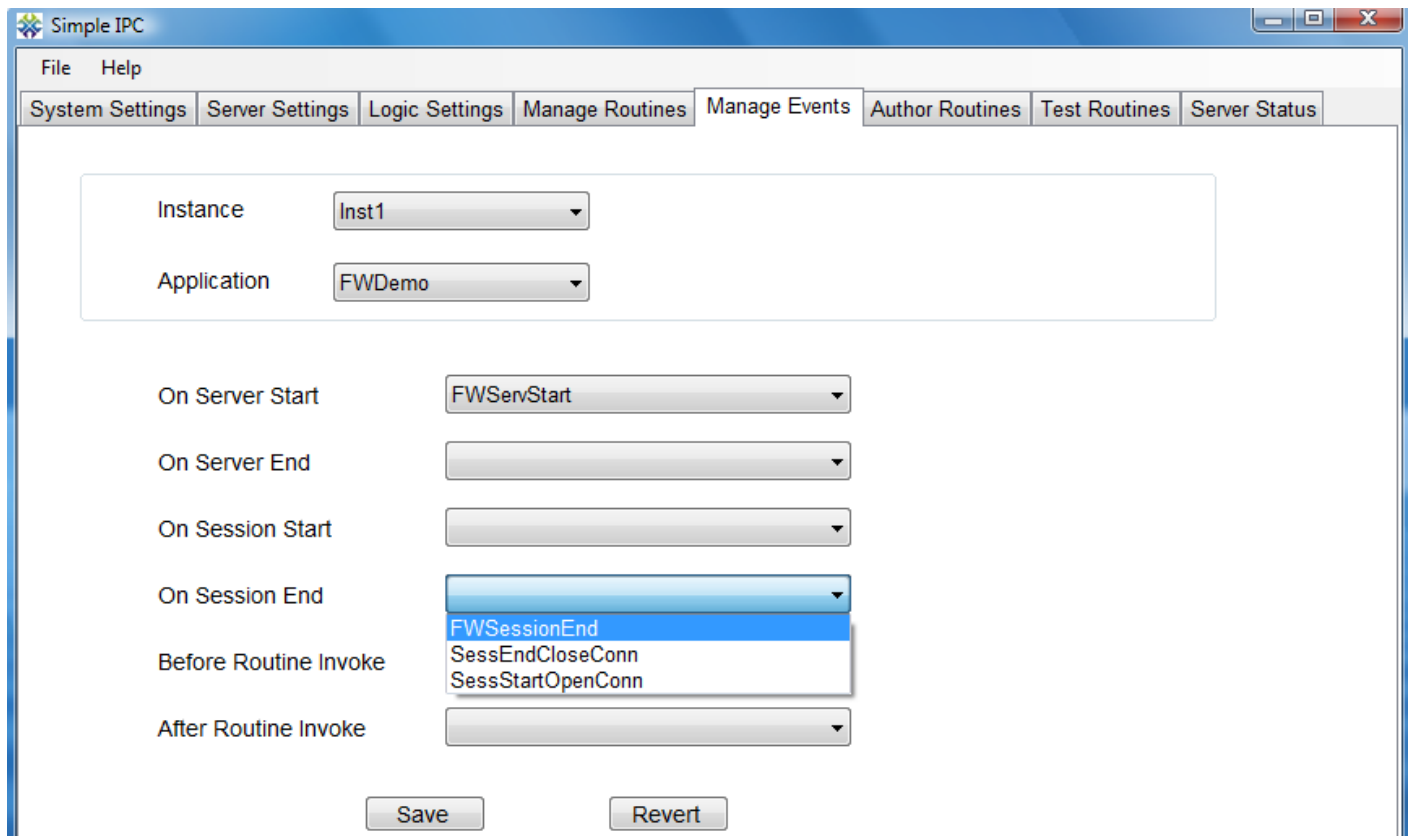
```

    }
    return bRet;
} //end function
} //end class

```

Step 9: Register the routine we just created. Go to the Manage Routines tab and click the "Refresh List" button so that our new dll will appear in the dlls drop-down. Select FWSessionEnd.dll, enter a routine name of FWSessionEnd, and save it.

Step 10: Associate the routine we just created with the "On Session End" event of the FWDemo application. Go to the Manage Events tab and toggle the selected Application to App1 and then to FWDemo (in order to refresh the lists of routines). Select the FWSessionEnd routine from the drop-down for the "On Session End" event and click the Save button.



Step 11: In this example, we only want to make one invocation of FWBasicCall per session because we put the code to terminate the server in the session exit routine. We do not want the client application to impose the restriction of making only one call per session, so we will define an intermediate routine. The intermediate routine will be named InvokeFW. The client application will call the InvokeFW routine (running in an IPC server hosting the Inst1/App1 combination). InvokeFW will make one call to FWBasicCall (running in an IPC server hosting the Inst1/FWDemo combination) per session. Go to the Author Routines tab of the CTConsole. Create a new Source Code file. Call the

file `InvokeFW`, and select the `BasicRoutineRun1` template. For this exercise, use the following code to make a call from one IPC server into another IPC server (be aware that word-wrapping can lead to syntax errors if you copy and paste directly into the script editor).

```
import System;
import System.Collections;
import System.Reflection;

[assembly:AssemblyVersion("1.0.0.0")]

public class InvokeFW implements CTSHost.IBasicRoutineRun1
{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor, oCallParamsAccessor:
CTSHost.CallParamsAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;

        //Define local string variables for the error codes upon which to try to
acquire a new session
        var SESSIONINVALIDORTIMEOUT: System.String =
System.Convert.ToString(CTCommon.Constants.RoutineExecutionErrors.SessionInvali
dOrTimedOut);
        var SERVERCOMMUNICATIONERR: System.String =
System.Convert.ToString(CTCommon.Constants.RoutineExecutionErrors.ServerCommuni
cationErr);
        var ERRRESUMINGSESSION: System.String =
System.Convert.ToString(CTCommon.Constants.RoutineExecutionErrors.ErrResumingSe
ssion);

        oLogUtil = CTCommon.LogUtil.Instance;
        bRet = false;
        try
        {
            //Instantiate the SessionEntry object - use the .NET Entry Point to get a
session on the other IPC server
            var oSessionEntry: CTRClient.SessionEntry = new CTRClient.SessionEntry();
            var oJSBasicCall1RetStruct: CTCommon.JSBasicCall1RetStruct;
            var oJSSessionReturnStruct: CTCommon.JSSessionReturnStruct;
```

```

        //We have to use the JScript.NET version of the function to obtain a
session
        oJSSessionReturnStruct =
System.Convert.ChangeType(oSessionEntry.JSObtainSession("Inst1", "FWDemo"),
CTCommon.JSSessionReturnStruct);
        if (oJSSessionReturnStruct.ReturnValue == true)
        {
            //We're basically forwarding the input arguments we received in this
routine invocation into the
            //call to the FWBasicCall routine
            oJSBasicCall1RetStruct =
System.Convert.ChangeType(oSessionEntry.JSBasicRoutineCall1(oJSSessionReturnStr
uct.SessionID, "Inst1", "FWDemo", "FWBasicCall",
oCallParamsAccessor.InputArgs), CTCommon.JSBasicCall1RetStruct);
            if (oJSBasicCall1RetStruct.ReturnValue == false)
            {
                //If the call returned false, then try to get a new session and
repeat the call if the error code is
                //one of the three that suggest a process or communications problem
with the other IPC server
                if (oJSBasicCall1RetStruct.ErrorAL.Count > 0)
                {
                    if (oJSBasicCall1RetStruct.ErrorAL(0) ==
SESSIONINVALIDORTIMEOUT || oJSBasicCall1RetStruct.ErrorAL(0) ==
SERVERCOMMUNICATIONERR ||
oJSBasicCall1RetStruct.ErrorAL(0) == ERRRESUMINGSESSION)
                    {
                        //Try to get a new session. If successful, repeat the routine
invocation.
                        oJSSessionReturnStruct =
System.Convert.ChangeType(oSessionEntry.JSObtainSession("Inst1", "FWDemo"),
CTCommon.JSSessionReturnStruct);
                        if (oJSSessionReturnStruct.ReturnValue == true)
                        {
                            oJSBasicCall1RetStruct =
System.Convert.ChangeType(oSessionEntry.JSBasicRoutineCall1(oJSSessionReturnStr
uct.SessionID, "Inst1", "FWDemo",
"FWBasicCall", oCallParamsAccessor.InputArgs), CTCommon.JSBasicCall1RetStruct);

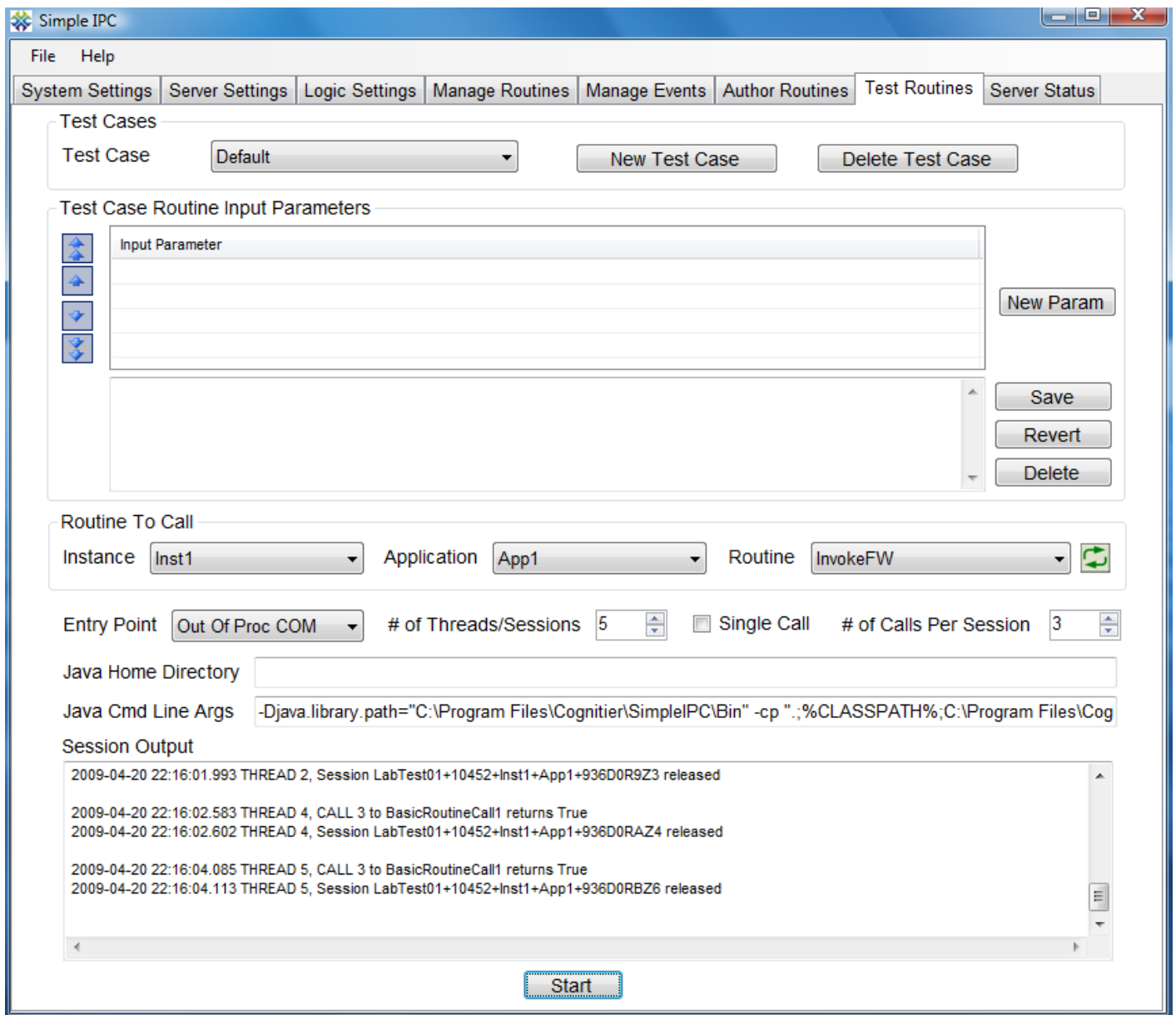
```



```
    //Also write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
    }
    return bRet;
} //end function
} //end class
```

Step 12: Register the routine we just created. Go to the Manage Routines tab and click the "Refresh List" button so that our new dll will appear in the dlls drop-down. Select InvokeFW.dll, enter a routine name of InvokeFW, and save it.

Step 13: Test your work. Go to the Test Routines tab of the CTConsole. Select the "Inst1" instance and the "App1" application. Click the Refresh button to the right of the Routines drop-down and select the InvokeFW routine. For this exercise, it does not matter which Test Case is selected or which Entry Point. To start, designate 5 sessions for your test, each making 3 calls per session.



When you run this test, you will be launching two IPC servers (one hosting the App1 application and responding to the invocation of the InvokeFW routine - the other hosting the FWDemo application and responding to the invocation of the FWBasicCall routine). If the IPC servers run within a visible console window, be aware that both servers may be writing their output to the same console window.

Open the file specified in your "MyFilePath" Logic Setting. There should be output similar to the following:

```
4/20/2009 10:14:53 PM Message written by server process 15112; client session
LabTest01+15112+Inst1+FWDemo+936D0PIZ1
4/20/2009 10:15:54 PM Message written by server process 15112; client session
LabTest01+15112+Inst1+FWDemo+936D0R9Z2
```

4/20/2009 10:15:54 PM Message written by server process 15112; client session LabTest01+15112+Inst1+FWDemo+936D0R9Z3
4/20/2009 10:15:55 PM Message written by server process 13212; client session LabTest01+13212+Inst1+FWDemo+936D0RAZ1
4/20/2009 10:15:56 PM Message written by server process 13212; client session LabTest01+13212+Inst1+FWDemo+936D0RBZ2
4/20/2009 10:15:56 PM Message written by server process 13212; client session LabTest01+13212+Inst1+FWDemo+936D0RBZ3
4/20/2009 10:15:57 PM Message written by server process 11772; client session LabTest01+11772+Inst1+FWDemo+936D0RCZ1
4/20/2009 10:15:57 PM Message written by server process 11772; client session LabTest01+11772+Inst1+FWDemo+936D0RCZ2
4/20/2009 10:15:58 PM Message written by server process 11772; client session LabTest01+11772+Inst1+FWDemo+936D0RDZ3
4/20/2009 10:15:59 PM Message written by server process 17352; client session LabTest01+17352+Inst1+FWDemo+936D0REZ1
4/20/2009 10:15:59 PM Message written by server process 17352; client session LabTest01+17352+Inst1+FWDemo+936D0REZ2
4/20/2009 10:15:59 PM Message written by server process 17352; client session LabTest01+17352+Inst1+FWDemo+936D0REZ3
4/20/2009 10:16:01 PM Message written by server process 22476; client session LabTest01+22476+Inst1+FWDemo+936D0RGZ1
4/20/2009 10:16:01 PM Message written by server process 22476; client session LabTest01+22476+Inst1+FWDemo+936D0RGZ2
4/20/2009 10:16:02 PM Message written by server process 22476; client session LabTest01+22476+Inst1+FWDemo+936D0RHZ3
4/20/2009 10:16:04 PM Message written by server process 21468; client session LabTest01+21468+Inst1+FWDemo+936D0RJZ1

We see that each line of text was written by a different client session, and the server process identifier indicates that the call to the FWBasicCall routine was serviced three times by each IPC server before that IPC server exited. Experiment with larger numbers of sessions and calls per session and observe a similar pattern in the target text file.

This exercise was made more complicated by the need to have no more than one IPC server for the Instance/Application combination running at any time. If you were simply trying to limit the load on a back-end resource to a handful of "active" clients, then the simpler approach would have been to have code within the FWBasicCall routine check the status of the server and put a warning in the warning array if the status had gone to "Due To Recycle". The calling client would need to look for the warning and release the current session and obtain a new one in this case. In this way, servers would remain running a short time while in Due To Recycle status, but the clients would stop making routine invocations at that point, which would constrain the concurrent load on the back end resource.