



# **Cognitier SimpleIPC**

## **Stabilizing Unstable COM Components**

### **White Paper**

Version 1.0.0.1

May 14, 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.

## Introduction

There are a vast number of very useful COM components deployed in Enterprise IT systems today. However, some of these components exhibit stability issues and memory leaks, and some are very sensitive to the threading model of the application consuming the components. A problem with a COM component can quickly turn into a problem for the consuming application. An expedient solution to retain the component functionality without incurring application instability is to use out-of-process hosting for the COM component itself, and inter-process communication between your application and the worker process hosting the component. Together with worker process lifecycle services, this approach can greatly enhance the performance, stability, and scalability of your existing applications.

## The SimpleIPC Approach

The approach to isolating unstable code and objects (of which COM components are just one example) taken by SimpleIPC is to dynamically spawn worker processes based on demand, and to host user-defined functions within those worker processes which are exposed to external applications over the Inter-Process Communication (IPC) channel. Within the custom functions, COM components might be instantiated and used, but it is really the function with its defined inputs and outputs being hosted, rather than a discrete COM component. The custom functions may be written in C# (or another .NET language) via Microsoft Visual Studio, or they may be "scripted" in JScript.NET using SimpleIPC's built-in script editor. The approach is conceptually similar to hosting Active Server Pages (ASP) in Microsoft Internet Information Services (IIS) and employing application pools with worker process isolation.

In the SimpleIPC architecture, the "worker processes" are IPC servers. These IPC servers are configured to run custom functions for defined events (server startup and shutdown, client session initiation and termination, etc.). The servers maintain server-wide and session-wide memory contexts for use in caching information between client invocations. The servers may also implement object pooling - enabling instantiated objects to be re-used by different clients throughout the life of the server. Also within the IPC server, new threads may be started, and the threading model may be explicitly manipulated. This makes it possible to host COM components which require a Single Threaded Apartment (STA) thread, for example. It also provides a straightforward means to ensure that invocations on a COM component occur on the same thread that instantiated the component.

The IPC servers have several available performance parameters which may be "tuned" for a specific application's needs. The most important parameters are the number of IPC servers which may be running concurrently, the number of client sessions each will accommodate, and the recycle criteria for periodically replacing running servers.

Clients establish sessions and make invocations against the IPC servers over the IPC channel. When a new client seeks to establish a session, it will find a vacancy on an existing IPC server. If all of the running IPC servers are fully occupied, a new IPC server will be started if the number of running servers is under the configured limit.

The IPC servers are designed to be expendable. Risky or otherwise unstable code and objects may be hosted by the IPC server. If an IPC server terminates expectedly or unexpectedly, then the consuming client application is unaffected. On the next attempted function invocation, the calling client can detect that the IPC server has exited. The client can simply establish a new session on a different IPC server and resume operations.

Although the custom code hosted by the IPC server must be written in a .NET language, the calling client may invoke functions within the IPC server via Java, COM, or .NET APIs. The multi-language APIs provide an expedient approach to interoping from Java to .NET, for example. Almost all of the server functionality can be achieved via a combination of Jscript.NET code and server configuration. The developer need not write low-level code for the cross-process communication, cross-language communication, establishing mutually-exclusive locks on resources, etc.

Permissions within the SimpleIPC architecture are based on Windows group membership. IPC servers may be configured to run under a fixed Windows identity, but the calling client may be running under a lesser-privileged account. Specific permissions are required for acting as an IPC server, accessing the IPC channel, and decrypting sensitive user-defined configuration settings (typically database connection strings, etc.). User and permissions administration typically consists of adding and/or removing user accounts from the three Windows groups established by SimpleIPC. Use of IPC as the communications channel has some inherent security advantages because access is restricted to processes running on the local machine (unless the SimpleIPC architecture is explicitly exposed to the network via the web service that ships with the product).

Application scalability can be increased by delegating code that leaks memory or runs the risk of throwing unhandled exceptions. Additionally, developers can typically limit concurrent access to shared resources via configuration, instead of implementing their own semaphores in code. This itself can improve scalability because sometimes a resource can process ten requests in sequence very quickly, but ten concurrent requests can cause a crash. SimpleIPC ships with a built-in test facility to simulate a configurable number of clients making a configurable number of function invocations against a set of IPC servers. This test facility gives the developer a straightforward way to get a preview of how their IPC server application will withstand a concurrent load. This gives the developer the opportunity to adjust the performance tuning parameters before the initial deployment into production.

## More Information, Samples, and Tutorials

Product tutorials with step-by-step instructions and sample code are available as follows:

### Practical Exercises:

#### *Use of the server-wide and session-wide IPC server contexts for caching objects and data.*

This exercise demonstrates how system resources can be conserved and performance improved through the use of caching.

<http://www.cognitier.com/downloads/Exercise1.pdf>

***Use of the "before-routine-invoke" and "after-routine-invoke" events to perform pre-processing and post-processing of routine invocations.***

This exercise demonstrates how to perform pre-processing and post-processing of routine invocations in SimpleIPC, similar to the way a SOAP Extension might be used to perform pre-processing and post-processing of web method invocations.

<http://www.cognitier.com/downloads/Exercise2.pdf>

***Implementing object pooling for the purpose of re-using object instances between client sessions.***

This exercise demonstrates how to instantiate a fixed number of objects of a particular type (such as database connections), and assign them to client sessions. A specific client has its own instance of the object and does not share it, but when the client ends its session and a new client obtains a session, the existing object instance is assigned to the new client session.

<http://www.cognitier.com/downloads/Exercise3.pdf>

***Using the configuration of performance parameters to limit concurrent access to specified system resources (throttling).***

This exercise demonstrates how to support a multi-user application which restricts access to a specified resource to a single client at a time. Multiple clients may be supported, but requests for the specified resource are serviced sequentially, rather than concurrently.

<http://www.cognitier.com/downloads/Exercise4.pdf>

***Deploying the SimpleIPC web service to allow access to IPC servers from across a network.***

This exercise demonstrates implementing the web service that ships with SimpleIPC for use with the Microsoft IIS web server. With the SimpleIPC web service, you can set up the web service one time, but then invoke any routine hosted by the IPC servers on the machine.

<http://www.cognitier.com/downloads/Exercise5.pdf>

**Product integration "HOW TO" articles:**

***Incorporating the Siebel COM Data Control in "thick client" mode in an Active Server Page (ASP).***

This exercise demonstrates in greater detail how to use SimpleIPC servers to provide the ideal environment for a COM component. Instructions are provided for restricting access to the COM object to one client per server, but deploying multiple servers in order to accommodate a significant concurrent load. Instructions for verifying the permissions and threading model for the server hosting the COM object are also provided.

<http://www.cognitier.com/downloads/SeblASPSample.pdf>

***Creating and returning a Microsoft Word document from a Java servlet hosted by a Tomcat web server.***

This exercise demonstrates interoping from Java to .NET. However, it also demonstrates how to emulate some of the behaviors of worker process isolation in a Tomcat-hosted web application. COM automation of Microsoft Word is used to create the document, and the throttling of access to the COM object and the document being created is achieved via configuration.

<http://www.cognitier.com/downloads/ServletWordDocSample.pdf>