CUSTOM SURROGATE HOST FOR ACTIVEX IN-PROCESS SERVERS

Nikolay Pavlov

Abstract: *This paper describes the design, implementation and evaluation of AX Host, a custom surrogate host for ActiveX in-process servers. AX Host aims to give ActiveX client applications improved stability by using software fault isolation.*

Keywords: ActiveX, surrogate host, software fault isolation
2010 Mathematics Subject Classification: 68U35, 68M15, 68N01

1. Introduction

Modern operating systems with Graphical User Interface have interoperability of applications as their primary goal. Multi-tasking and the ability to quickly switch between applications are coupled with various means to exchange. This interoperability is an important feature of desktop applications, which distinguishes them from web applications even in the era of Web 2.0. Therefore, for a desktop application to maximize the user experience, it should be able to integrate with other software packages, which play important role in the operations of users.

The most common means to exchange data, which are not dependent on the operating system, is to use files in known formats. These, however, require manual operations from users. Microsoft Windows provides several ways for applications to exchange data – clipboard, window messages, memory-mapped files, pipes, sockets, local and remote procedure calls, and OLE compound documents [2].

The Framework for business applications [7] provides in-built support for document management. It integrates with the de fact standard packages Microsoft Office and Adobe PDF Reader for that purpose. The Framework provides tight coupling of documents with business data, including inserting data from the application into Word documents. It enables users to create documents from within their application, store it in their application database, and access and manage their documents in their application, and under the relevant data entities. The selected technology for integrating the Framework with Microsoft Office and Adobe PDF reader is OLE compound documents [2].

This paper describes the software fault isolation subsystem of the Framework for using OLE compound documents, which protects Framework applications from errors in the compound document servers.

2. Overview of the problem

OLE compound documents enable users working within a single application to manipulate data written in various formats and derived from multiple sources. For example, a user might insert into a word processing document a graph created in a second application and a sound object created in a third application. Activating the graph causes the second application to load its user interface, or at least that part containing tools necessary to edit the object. Activating the sound object causes the third application to play it. In both cases, a user is able to manipulate data from external sources from within the context of a single document.

OLE compound document technology rests on a foundation consisting of COM, structured storage, and uniform data transfer.

OLE's compound document technology benefits both software developers and users alike. Instead of feeling obligated to cram every conceivable feature into a single application, software developers are now free, if they like to develop smaller, more focused applications that rely on other applications to supply additional features. In cases where a software developer decides to provide an application with capabilities beyond its core features, the developer can implement these additional services as separate DLLs, which are loaded into memory only when their services are required. Users benefit from smaller, faster, more capable software that they can mix and match as needed, manipulating all required components from within a single master document [2].

Compound document applications are of two basic types: container applications and server applications. OLE container applications provide users with the ability to create, edit, save, and retrieve compound documents. OLE server applications provide users with the means to create documents and other data representations that can be contained as either links or embedded in container applications. An OLE application can be a container application, a server application, or both [3].

As mentioned earlier, the Framework utilized OLE compound documents to provide users with seamless integration with Microsoft Office and Abode Reader. The technique allows users to author and access Office and PDF documents within the Framework application without having to switch to and from the appropriate external application.

Real-life practice, however, demonstrated that the Framework is experiencing undesired instability issues due to faults in compound document server

applications. Extensive studies of error logs, stack traces and memory dumps revealed corrupted memory pointers, outside the code of the Framework application. The culprit was found – a faulty module in Adobe Reader. While the system experienced problems with Microsoft Office, too, the Framework application is always able to recover and continue its operation. On the other hand, errors in Adobe Reader caused the Framework application to crash. The explanation of the different behavior lies in understanding how OLE server applications work.

OLE server applications differ in whether they are implemented as in-process servers or local servers. An in-process server is a dynamic link library (DLL) that runs in the container application's process space. You can run an in-process server only from within the container application [3].

The type of server – in-process or local, has a direct impact on the stability of container applications. Local servers run as separate isolated processes. Unhandled or catastrophic errors in in-process servers happen within the process of the container application, and translate as direct errors in their code. These errors are difficult, if not impossible to trap and handle, depending on the nature of the error. Even if trapped, they are very likely to leave the process in unstable state, and therefore the process cannot reliably recover and continue working. As a result, such errors normally cause container applications to crash, thus and increase the risk of data-loss for users. On the other hand, local servers run as separate, standalone processes. Unhandled errors remain within the scope of the local server, and are only communicated as messages (OLE exceptions) to container applications. A crash in a local server does not force container applications to quit, and container applications are able to reinitialize the affected local server, and retry the operation, thus minimizing the negative impact on users, and reduce chances of data loss. Data loss can be zero in case a compound document local server is used only to represent data visually.

3. Solution design

This is a well-known issue with compound document infrastructure, called surrogates [4, 5, 6, 8]. It allows an in-process server to be executed in a special process, outside the client process which requested it.

The roles in surrogate case are: 1) requesting application; 2) surrogate host; 3) in-process server.

When the in-process server crashes, the surrogate host will quit, but the client process will have a chance to recover and continue its operation. However, existing solution in COM system infrastructure is global for the operating system.

Therefore, it should be used only with components, which are developed and tested with it, or it can cause instability in other compound document containers. Risk of incompatibility with other applications discarded this solution as a feasible option.

Windows Vista introduces a new technique for read-only presentation of external content, named Preview Handlers [5]. The problem is this technique is not supported on Windows XP, which makes it inappropriate for the Framework.

The suggested solution involves design and development of a custom surrogate host. It is not system-global, i.e. it is applied only by the Framework for consuming services from Adobe PDF, and therefore does not break other software applications. The structure of the solution is the same as the COM Surrogate. The Framework application is a requesting application, and AX Host performs the role of a surrogate host.

The key implementations are three:

1) AX Host runs the Adobe PDF ActiveX control

2) It receives a valid handle to a visible windowed control from the Framework application, and assigns it as a parent to the Adobe PDF compound document (ActiveX) control.

3) A vector structured exception handler is used to mask the error from the operating system, and perform a gracious exit of the application.

This has the effect of the compound document to be visualized within the Framework application, as it if is hosting the in-process server. In reality, the in-process server runs completely in the context of AX Host.

In case of faulty behavior of the in-process server, AX Host will crash. The Framework application will only lose the visual presentation, but its state will not be corrupted, and it will be able to continue. Actually, when the Framework application detects that AX Host has ended prematurely, it will try to re-initialize it.

The workflow involves these steps:

1. The Framework application executes AX Host and passes via the command line its Windows process handle and a reading handle to a pre-created anonymous pipe.

2. AX Host performs internal initializations and starts listening to commands from the Framework applications coming from the anonymous pipe.

3. On request, AX Host loads Adobe Reader ActiveX control, initializes it and asks it to load the requested document. Then, it assigns its parent window to a window handle, provided by the Framework application.

4. AX Host pumps messages and waits for commands by the Framework application via the pipe. AX Host also checks regularly if the Framework application is still running.

5. On request from the Framework application, or on detecting a failure, AX Host cleans up and quits.

4. Implementation

AX Host is developed as a standard Win32 application with an invisible window. The window is used to pump messages for COM and for the visual control of the in-process server. All windows messages to the Adobe PDF ActiveX control are send to the message queue of AX Host, and processed by it. Therefore, the thread on AX Host which creates the Adobe PDF ActiveX control must not block until the ActiveX control is loaded.

Reading anonymous pipes is blocking until there is data to be read. At the same time AX host must constantly pump and process messages for COM and the visual control. Therefore pipe communications are executed on a separate thread. When a new command is received through the pipe, the reading thread sends a message to the main window of AX Host. It does not make direct calls, as the ActiveX control is created in the thread of the main window for the reason of pumping messages. Consequently, all calls to the ActiveX control must be made on this thread.

4.1. Communications

The Framework application and AX Host use several means of exchanging data and messages. The Framework application runs AX Host with two command line parameters: the reading handle of the anonymous pipe, and its handle identifier (Id). The read pipe handle enables AX Host to receive commands from the Framework application. The process id is required, because Windows NT does not provide a documented way for a child process to identify its parent process. This id is used for lifecycle management, as described later on.

AX Host listens to incoming commands through the pipe in a separate thread (pipe thread), because IO operations with anonymous pipes are blocking. The following commands are being sent over the pipe:

- Set control's parent window.
- Load document.
- Unload document.
- Quit.

All actions on commands are carried out by the thread, which pumps messages and controls the Adobe PDF ActiveX control. The pipe thread dispatches the commands by sending messages to its message queue.

```
while (1)
{
  ZeroMemory(&Msg, sizeof(Msg));
```

```
      ReadFile (ReadPipeHandle, Msg, sizeof (Msg), &BytesRead,
null);
      switch (Msg.Command)
      {
         case cmdPDFSetParentHandle:
           ParentControlContainer = PipeMessage.wParam;
           PostMessage (WindowHandle, WM_SETPARENT_AX_PDF, 0, 0);
           break;
        …
         case cmdPDFClose:
           ExitCode = EC_CLOSED_GRACEFULLY;
           PostMessage(WindowHandle, WM_UNLOAD_AX_PDF, 0, 0);
           PostMessage(WindowHandle, WM_QUIT, 0, 0);
           break;
      }
    }
```

4.2. Message Queue

Once all initialization is complete, and the pipe thread is running, the Host enters the message processing loop:

```
    while ((getMsgRetValuye = GetMessage(&Msg, 0, 0, 0 )) != 0))
    {
      if (WaitForSingleObject (ParentProcessHandle, 0) ==
WAIT_OBJECT_0) break;
      TranslateMessage(&Msg);
      DispatchMessage(&Msg);
      }
```

The loop performs two tasks. First it processes all messages, received by its window and by the window of the Abode PDF ActiveX control. Second, it checks the state of the Framework application.

4.3. Lifecycle

Normally, AX Host will quit when it receives a command from the Framework application that its services are no longer required. Alternatively, it will close in case it intercepts an error from the in-process server. In this case it will notify the Framework application.

It is also possible that the Framework application crashes due to other error before it has notified AX Host to close. The risk to have abandoned instances of the host application running and consuming resources is avoided by an extra check. AX Host uses the process handle of the Framework application which created it to create a synchronization object for that handle. On every new message AX Host performs a non-blocking wait on the synchronization object. If the object becomes

signaled then AX Host knows the Framework application has been closed prematurely. In such case it will cleanup and exit.

An implementation detail is that AX Host creates a timer, which sends a message to the window message queue every five seconds. AX Host does nothing on processing this message. The timer acts as insurance that the check for the state of the Framework application will be executed regularly, because it is possible that no messages are sent to AX Host window if the Framework application crashes. If there are no messages available, GetMessage will not return, and AX Host will not have a chance to check the Framework application.

4.4. Error Handling

Errors, unhandled by applications, are trapped by OS Windows and reported to users. Even though these error messages do not refer to the Framework application, they are confusing to users and reduce user experience. Apparently the Adobe PDF ActiveX control is using internally several threads, and errors, occurring in them, cannot be trapped using frame-based structured exception handling, i.e. using standard try … except blocks.

Vector structured exception handling enables developers to set a global exception handler for their application. A vector structured exception handler is used to trap the errors in the Adobe PDF ActiveX control. The handler is used to mask the error from the OS and thus suppress the system error message, and to initiate closing of AX Host. The error in the ActiveX control leaves AX Host process in a unknown state, and therefore recovery is impossible.

5. Results

The described solution is currently deployed in Dutch and German companies from ship insurance brokerage. These clients have an average size of 30 concurrently operating workstations. The results after six months demonstrate that the Framework application no longer exhibits faults, caused by the Adobe PDF ActiveX control. Errors are successfully masked by AX Host application. The Framework application is able to recover, and restart the host. The experience for users is only slight delays in viewing PDF documents.

6. Conclusion

The suggested solution achieves its goal to improve the user experience. It can easily be adapted for use with other in-process compound document servers.

References

[1] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, Google Inc, Native Client: A Sandbox for Portable, Untrusted x86 Native Code (2009), In Proceedings of the 2007 IEEE Symposium on Security and Privacy

[2] Microsoft. Compound Documents (COM). *MSDN*. [Online] 2010. http://msdn.microsoft.com/en-us/library/ms693383(v=VS.85).aspx.

[3] Microsoft. Containers and Servers (COM). *MSDN*. [Online] 2010. http://msdn.microsoft.com/en-us/library/ms682269(v=VS.85).aspx.

[4] Microsoft. DLL Surrogates (COM). *MSDN*. [Online] 2010. http://msdn.microsoft.com/en-us/library/ms695225(VS.85).aspx.

[5] Microsoft. Preview Handlers and Shell Preview Host. *MSDN*. [Online] 2010. http://msdn.microsoft.com/en-us/library/cc144143(VS.85).aspx.

[6] M.H. Knahl, A Componentware based Management Framework Utilising Adaptation, 2003, Proceeding (394) Computer Science and Technology – 2003, May 19 – 21, 2003, Cancun, Mexico

[7] Pavlov N., A. Rahnev, Architecture and Design of Customer Support System using Microsoft .NET technologies, .NET Technologies 4th International Conference, May 29 – June 1 2006, Plzen, Czech Republic, ISBN 80-86943-11-9, pp 21-26.

[8] Taeho Kwon Zhendong Su, Automatic Detection of Vulnerable Dynamic Component Loadings, CSE-2010, UC Davis Department of Computer Science

Nikolay Pavlov,
9N Kuklensko Shose Str,
4004 Plovdiv, Bulgaria
e-mail: nik.pavlov@kodar.net