Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Collaborative Web Surfing

Source Code Accompanies This Article. Download It Now.

- cosurfer.txt
- cosurfer.zip

Cosurfer is a peer-to-peer GUI application that lets two users chat and surf the Web together.

April 01, 2005
URL:http://www.drdobbs.com/web-development/collaborative-web-surfing/184406030

A peer-to-peer application for surfing the Web together

Gigi is a software developer specializing in object-oriented and component-oriented programming using C++. He can be contacted at gigi_sayfan@playstation.sony.com.

Cosurfer is a peer-to-peer (P2P) GUI application that lets two users chat and surf the Web together. Cosurfing means that whenever one user browses to a new URL, the other user's browser is automatically updated so they are "always on the same page"—literally. Users typically run Cosurfer on different machines, although during development I run two Cosurfers on the same machine for testing purposes. Cosurfer has a GUI, connection manager, browser component, and Cosurfing engine that orchestrates all the other components. The complete source code and related files for Cosurfer are available electronically; see "Resource Center," page 5.

Cosurfer involves a number of aspects of .NET programming, including:

- Windows Forms programming.
- Socket-level networking.
- Asynchronous method calls.
- Multithreaded programming.
- Events and Delegates.
- XML parsing and composition.
- COM interop.

In addition, Cosurfer explores the Internet Explorer (IE) programming model quite extensively.

When you launch Cosurfer, it automatically starts listening on port 8888. If you launch a second instance, it starts listening on port 8889. If you click on the Connect button on one of the forms, a TCP connection is established. In addition, each instance launches a new IE browser with a blank page. From this point on, the two Cosurfers function like Siamese twins. Whenever users navigate to a new URL or frame inside one of the browsers, the other browser follows. Moreover, you can exchange witty chat lines with your friend (or yourself) through the chat facility. The trick is to hook into IE and monitor it closely. Every change is encoded in a proprietary XML dialect and reported to the peer Cosurfer, which updates the attached browser. It is not enough to send only URLs because frame-based web pages might have arbitrarily nested structure.

IE Programming

When I originally wrote Cosurfer, Internet Explorer was a web browser that was starting to show its age. It hadn't been updated for years (except for security patches). Nevertheless, it exhibited (and still does) total configurability and flexibility through a variety of methods. You can control some aspects of its operation through command-line switches and other aspects through registry entries, you can customize its appearance, you can host it in your application (with or without UI), and you can hook into its events and intercept and modify almost anything it does. It provides several ways to attach your code to running instances and also an elaborate DHTML object model. In this article, I show how to launch a new instance of IE, hook into its events, control its navigation, and drill down the HTML DOM.

ShDocVw.dll, located in *<your windows dir>*\System32 and commonly known as the "the WebBrowser control," contains various shell COM objects, interfaces, and enumerations. One object is *InternetExplorer*, which represents an instance of a standalone IE application. You can use the OLE/COM Object Viewer tool (located in *<your visual studio dir>*\Common7\Tools\oleview.exe) to investigate it. The type library, called "Microsoft Internet Controls Version 1.1," contains 18 interfaces, nine objects (coclasses), and eight enumerations. Most interfaces are dual and contain around 10 methods each. Luckily, you don't have to tackle the raw complexity head on. As you can see in Figure 1, the *InternetExplorer* object implements two interfaces —*IWebBeowser2* and *IWebBrowserApp*—and has two outgoing *dispinterfaces*, *DWebBrowserEvents* and *DWebBrowserEvents2*. The *dispinterfaces* are event interfaces that the client code implements to receive events from *InternetExplorer*. *DWebBrowserEvents* exists only for compatibility reasons with code that automates Internet Explorer 3 (can you believe it?). You can safely concentrate on *DWebBrowserEvents2*.

Listing One (also available electronically) contains the *Puppeteer* class that controls IE, demonstrates how to create an instance of the *InternetExplorer* application, hooks into two events (*Quit* and *NavigateComplete*) and shows how to navigate to a URL (*google* in this case). The *AutoResetEvent m_quitEvent* is initialized to *false*, and the *Run()* method waits for it to be signaled. This happens when the *Quit* event is received from the browser.

MSHTML.dll (located in *<your windows dir>*\System32) is responsible for HTML parsing and rendering, as well as exposing the Dynamic HTML Document Object Model (DHTML DOM). It is usually hosted by ShDocVw.dll, although it can be hosted directly by any application. If you thought the ShDocVw.dll contained many interfaces and objects, think again. MSHTML.dll has an intimidating number of objects and interfaces—147 *enums* and *typedefs*, 121 objects, and more than 400 interfaces! The good news is that you will only need to work with a few interfaces most of the time.

Listing Two (available electronically) shows how to get the *Document* object (which is really an *IHTMLDocument2* interface) and use its *all* collection to iterate over all the elements in the document. Once you get the *DocumentComplete*, it means that the DHTML object model can be accessed safely.

(Before using ShDocVw.dll and MSHTML.dll, make sure they are referenced by the using assemblies. If they are not, you should import them using Add Reference... from the context menu of assembly references.)

HTML Documents and Frames

So far, this sounds simple. The browser finishes loading a document and sends the *DocumentComplete* event. In your code, you respond by getting the *Document* property and starting to use the object model. However, frames introduce some complications. For starters, there are two types of frames —regular frames embedded in a frameset, and iframes that are embedded anywhere inside another document. An HTML/XHTML document should have either a frameset or body element. If it has a frameset element, it means that the document is actually divided into multiple frames or more nested framesets; see Figure 2. Each frame contains a separate document with its own URL. Of course, such a framed document may have a frameset that contains more frames. If it has a *body* element, then it may contain (in addition to regular elements) one or more *iframe* elements, which are simply HTML documents embedded in the middle of the body. The frameset.html document represents such a complex document. Here is its structure:

```
frameset.html
frame_1.html
frame_2.html
frame_3.html
www.google.com
```

The *frameset.html* document has no content of its own—it just "hosts" *frame_1*, *frame_2*, and *frame_3*. *frame_3.html* contains an *iframe* element that points to Google's main page. If you search through Google and browse to another framed document, then the structure becomes even more complicated.

IHTMLDocument2 provides the frames collection that is supposed to contain all the frame elements in a document, with a frameset element or all the *iframe* elements in document with a body. Unfortunately, it doesn't work. The debugger claims that the *frames* collection is indeed of type *mshtml.FramesCollection*, but the value is *<error: an exception of type: {System.InvalidCastException} occurred>*. I recall that, even in the old days, when I tried to manipulate the DHTML object model from C++, the *frames* collection never worked properly. Fortunately, the *all* collection also contains all the frame elements. Checking the documentation, you find that there are numerous frame-related interfaces—*IHTMLFrameBase, IHTMLFrameBase2, IHTMLFrameBase3, IHTMLFrameElement, IHTMLFrameElement2, IHTMLFrameCollection2, IHTMLFramesetElement,* and *IHTMLFramesetElement2*—most of which deal with various visual properties of the frame elements or containment relationships. When drilling down the DHTML object model of a nested document, you would like to recursively acquire the document object inside each frame. Fortunately, all the various flavors of frame elements implement the *IWebBrowser2* interface from which you can get to the document object.

Another complication related to multiple frames embedded in a frameset is that the order of the *DocumentComplete* events is not deterministic, so when you get a *DocumentComplete* event, it is not clear from what frame it originated (especially if some frames have the same URL). A simple way to deal with it is to drill down recursively from the top-level document whenever any *DocumentComplete* event is received. Listing Three (available electronically) contains a different *OnDocumentComplete()* handler that employs this technique.

.NET Socket Programming

The Base Class Library (BCL) provides communication APIs spanning diverse protocols, technologies, and levels of abstraction. The Socket API (popularized by BSD sockets) is where the rubber meets the road. At this level, you send/receive raw byte buffers. The traditional socket programming model is blocking, which means that the application is blocked until some network event happens (data is received or a connection is accepted, for instance), or the application programmer has run the socket code in a separate thread. There is also a polling API, where the app can check if some network event has happened and handle it or carry on with its business if not. Finally, asynchronous I/O is introduced where the application received a notification when a network event happened. All these APIs and programming models made the sockets API complicated. There are numerous options you can set; some functions should be used only in certain situations and only with other functions.

Now, I'll demonstrate how to work in asynchronous mode with the *Socket* class using the TCP protocol. The *System.Net.Sockets* assembly contains all the interesting types. *System.Net* contains some helper types that are useful, too. Echo server is the canonical communications "Hello, World" program where the server simply sends back to the client whatever it gets. Listing Six (available electronically) contains the entry code to the app that simply creates *TcpEchoServer* and *TcpEchoClient* objects, tells the server to listen on port 6666, and tells the client to connect to the server. The client and server take it from there, exchanging messages. The server (Listing Seven; available electronically) binds itself to a port and starts listening for incoming connections. Immediately after calling *listen()*, the server should be ready to accept connections. The *Socket* class provides an asynchronous method pair *BeginAccept()*/*EndAccept()* and *TcpEchoServer* uses it. Once a connection has been accepted, a new socket is created, and through this socket, the server sends/receives data to/from the client. While most servers handle many clients, in this code (and in CoSurfer), there is always just one client (peer). The client (Listing Eight; available electronically) model is different. It connects to the server, which resides in some well-known *EndPoint* (IP + port). Once connected, the client starts sending messages to the server using *Send()* and waits for responses using *OnReceive()*, which is the asynchronous callback function. The server works in a similar fashion—it waits for messages using *OnReceive()* and responds by sending them back using *Send()*. Both the server and client may close the connection at any time.

The *Socket* class works with raw byte arrays while most applications exchange some combination of text and binary data. It is necessary to translate back and forth between byte arrays and your data structures. *System.Net.Sockets* contains the two helper classes *TcpListener* and *TcpClient*, which take much of the drudgery away. They provide a thin veneer on top of the *Socket* class and are inherently thread safe. The classes wrap the buffer and expose a stream abstraction to *Read()* and *Write()*, which takes care of the necessary transformations and buffer recycling. The *TcpClient* also provides a friendly *Connect()* method overload that accepts a string URL so you don't have to painstakingly create an *IPEndPoint* that the *Socket.Connect()* method requires. There is also a *UDPClient* for datagram protocol communications (no connections and no server here). I chose to use the raw *Socket* class because I wanted to investigate the nuts-and-bolts and full control and asynchronous operation. While the *TcpClient* class is probably appropriate for most purposes, the *TcpListener* is a little weak. It doesn't have *BeginAccept()/EndAccept()* methods—only *Accept()* and a *Pending* property to poll for incoming connections. Another major omission is that it lacks the *Select()* method, which is an absolute must for servers that handle a lot of traffic. I will not go into all the intricacies here, but to that, *Select()* is needed to handle multiple connections without creating a separate thread for each connection (which will bring a machine to its knees after several hundreds of connections).

.NET XML Programming

XML is a .NET technology pillar (borrowing a catchphrase from Longhorn nomenclature). Several other core .NET technologies—configuration files, ADO.NET, and web services—rely on XML. I leave it to you whether it is wise to tie your platform so tightly to a verbose textual format. In any event, XML is standard, ubiquitous, and hot, and the answer for platform/language neutral structured data exchange across any boundary. Of course, no one can keep track of all the XML languages, technologies, and metatechnologies that pop up everywhere. The .NET Framework seems to gather the most important XML-derived standards (XPath, XML Schema, XSLT, and so on) under its wings. The XML facilities in the BCL include parsing, composition, validation, navigation, and transformation. Here, I only explore XML parsing and composition because that's what I use in Cosurfer.

The two most common programming models for parsing XML are DOM (Document Object Model) and SAX (Simple API for XML). DOM parses an entire XML document and creates a tree structure in memory, which it returns for the calling code to manipulate. The *XmlDocument* from the *System.Xml* assembly provides conforming DOM level 1 and level 2 implementations. SAX streams through the document, raising events whenever it encounters an element, processing instruction, or attribute. It's the application's responsibility to handle all the appropriate events and there is no going back. It is also known as the "push model." The BCL provides an interesting alternative model. The *XmlReader*-derived classes (*XmlTextReader, XmlNodeReader,* and *XmlValidatingReader*) implement a pull model, which is a combination of DOM and SAX. The idea is to provide an efficient read-only, forward-only noncaching parser. Instead of firing events automatically as SAX does, the *XmlReader* model lets the application pull more content whenever it is ready. It is also possible to skip the children of the current node. This way, whole branches may be pruned and a lot of processing is avoided.

In Cosurfer, I only use the *XmlDocument*'s DOM interface. Listing Nine (available electronically) demonstrates how to load an XML buffer from a string into a new instance of *XmlDocument*, get the root element (*DocumentElement*), and then iterate over all the children recursively and print the value of an attribute. The code reads just like plain English, which is a sign of a good interface (credits go to W3C for designing the DOM interface).

XML Parsing and traversing the DOM (or using *XmlReader*) is what you need most of the time. However, composing XML documents dynamically is also often necessary. You can do it by creating a new *XmlDocument* and start creating nodes and appending them, but there is an easier way. The *XmlTextWriter* is the best tool for the job. *XmlTextWriter* has several constructors that take one of various outputs (a stream, text writer, or filename). It has numerous *WriteXXX* methods to write elements, attributes, processing instructions, and whatnot. Some of them come in pairs, as in *WriteStartDocument()* and *WriteEndDocument()*. It tries very hard to make you write well-formed XML. For example, if you write multiple XML processing instructions or if you don't have exactly one root element, it raises an exception (except if you write an XML fragment). I noticed one glitch when using the constructor that accepts a *TextWriter* and using the *WriteStartDocument()* method. *XmlTextWriter* automatically writes the line: *<?xml version="1.0" encoding="UTF-16">*. This is unfortunate because the encoding was not UTF-16. I couldn't find a way to control the encoding (it is possible with the constructor that accepts an *IO.Stream*); so eventually, I dropped the *WriteStartDocument()* method and created the XML processing instruction myself (Example 3).

Cosurfer Architecture & Design Principles

Cosurfer is made up of a couple of generic components that can be used as-is in other applications and a couple of components that are specific to the Cosurfer application. The components utilize abstract interactions through interfaces. Every component defines an events interface *I<component name>Events* (for example, *IConnectionManagerEvents*). An interested component (event sink) implements the events interface to receive events. The sender components needs a reference to the sink, of course, to call the event handlers it implemented. A simple solution is to pass the sink as one of the constructor arguments. However, in some cases, two components need to call each other. In this case, it is not possible to pass both references in the constructor. In this case, one of the components implements an *AttachSink()* method that can be called later. The *Factory* class usually hooks up event sources to event sinks by attaching the receiving component to the sending component by passing it in the constructor or via the *AttachSink()* method (Example 1).

User Interface. The Cosurfer UI is spartan, yet you can learn a lot about Windows.Forms programming by following the code. There is only one window (or "form" in Windows.Forms lingo). This form contains a connect button and two edit boxes for chat purposes (Figure 3). The form is semitransparent (*Opacity=70%*) and always stays on top (*TopMost=true*). This combination keeps it always visible, while not completely obscuring what transpires underneath. It is convenient during development when I have two Cosurfer instances and two IE browsers open. The design of Cosurfer separates the UI code from the functional code. The *MainForm* receives events from the various components and updates the UI accordingly. In response to user actions, such as pressing the Connect button or sending a new chat line, the *MainForm* simply delegates the action to another component. This kind of design allows for better maintainability and flexibility. For example, it should be simple to port Cosurfer to XML because the functional components are totally UI agnostic.

ConnectionManager. The communication layer of Cosurfer works at the TCP socket level and is mildly sophisticated (see Example 4). Every Cosurfer is both a server and a client. The reason is that, as a P2P application, every Cosurfer may initiate a connection or accept incoming connection from a peer. The .NET Framework provides *TcpClient* and *Socket* classes that the *ConnectionManager* class builds upon. Asynchronous I/O (as opposed to blocking or polling I/O) is usually preferred in an event-driven programming model. While the *TcpClient* supports Async I/O through its stream, there is no corresponding Async listening capability. The *TcpListener* class supports only blocking I/O or polling I/O. The *ConnectionManager* uses instead the *Socket* class to listen for incoming connections from the peer. However, instead of using the *BeginAccept()/EndAccept()* method, I chose to create a thread (or rather request a thread from the *ThreadPool*) and use the blocking *Accept()* method (see Listing Ten; available electronically). Once connected, it uses

asynchronous method calls to read incoming data and act accordingly. The *ConnectionManager* encapsulates the gory details and exposes a streamlined asynchronous interface to the world. Listing Five (available electronically) contains two interfaces: *IConnectionManager* and *IConnectionManagerEvents*. *IConnectionManager* is the active interface that *ConnectionManager* implements. It allows connecting and sending data to a connected peer. The *IConnectionManagerEvents* interface is implemented by the user of *ConnectionManager*, and *ConnectionManager* calls its methods when a corresponding event occurs. The *ConnectionManager* is a generic TCP P2P component. It knows nothing about the specifics of Cosurfer or even the type of the object that implements the *IConnectionManagerEvents* interface. The interaction is completely abstract through an interface. The only assumption the connection manager makes is that the stream of bytes it reads is ASCII encoded because it converts the bytes to an ASCII string before calling the *OnReceive()* method.

Browser Component. The Browser component is a wrapper around the *BrowserControl* COM object. I use the COM interoperability to utilize it in the .NET-managed environment. Making it work was the most difficult part of the project. I sort of anticipated it because integrating and bridging across separate technologies is almost never painless.

The *Browser* class public interface includes the *Navigate()* method and two properties—*Busy* and *Document*. It also has a constructor that accepts an event sink that implements the *IBrowserEvents* interface. The constructor creates a new instance of the *InternetExplorer* COM object (which effectively launches a new Explorer window) and stores the event sink reference. *Navigate()* navigates the associated web browser to the provided URL. The *Browser* listens for the two web-browser events *DocumentComplete* and *OnQuit* and immediately forwards them to its sink. When the sink receives the *OnDocumentComplete* event, it should check if the browser is busy (via the *Busy* property), and if it is not, it is safe to access the entire nested object model through the *Document* property.

CosurfEngine. The *CosurfEngine* is the brain of Cosurfer. *CosurfEngine* controls Cosurfer's behavior and the protocol used for communication. It also interacts with the generic components (*ConnectionManager* and *Browser*). The engine is wired to the *ConnectionManager* and the *Browser* components (thanks to the *Factory*). It reacts to browser and connection manager events. In the case of connection- or chat-related events, it forwards them to the *MainForm* through the *ICosurfEngineEvents* interface and handles all other events itself.

The interesting events are *OnReceive()* from the *ConnectionManager,* and *OnDocumentComplete()* from the *Browser*.

When *OnReceive()* is called, it means that the peer Cosurfer is sending a chat line or surf buffer. To distinguish between the two, a chat line is surrounded by the XML-like <*ChatLine*> and </*ChatLine*>, and a surf buffer is already an XML document. This is necessary to be able to parse the incoming data into meaningful messages (either chat lines or surf buffers). Without it, all kinds of TCP streaming issues raise their heads, such as long chat lines and surf buffers that are divided between multiple packets or multiple chat lines and/or surf buffers that arrive in a single TCP packet (*OnReceive()* event). These issues should be handled in any industrial-strength application. I accumulate partial data until I have a full message (chat line or surf buffer) and then chat lines are simply forwarded to the *MainForm* via the *OnIncomingChatLine()* event. Surf buffers get much more serious treatment. A surf buffer is an XML serialization of the frame structure of the HTML document in the peer's browser. Example 2 contains the XML you get when visiting http://www.w3schools.com/tags/planets.htm, which has a frameset that contains three frames. When the *CosurfEngine* receives a surf buffer, it can be in one of two states: *Idle* or *Updating*. *Idle* means idle (big surprise); *Updating* means that *CosurfEngine* is in the process of navigating the local browser to an incoming surf buffer. This is not an atomic operation due to the asynchronous nature of getting web pages combined with the infamous multiple nested frames. If the state is *Updating* and the browser is still busy, the incoming surf buffer is simply ignored. The motivation is not to interfere with an ongoing complex process that can easily get out of whack. So how do the browsers stay in sync if some surf buffers are ignored? When the updating *CosurfEngine* is done, it sends the complete surf buffer to its peer. This sounds like a vicious circle, but if the incoming surf buffer corresponds to the content of the receiving browser, there is no sending back. Thus, if both users stop fiddling with their browser for a second, both browsers will settle down. This mode of operation is intuitive (from the user point of view) and does not require explicit control passing as in "now, it's my turn to navigate us somewhere." It is very nonintuitive to figure out from the programmer's seat. I spent a lot of time getting this micro state machine (only two states) working.

The *OnDocumentComplete()* event is sent whenever the browser finished loading a document into one of the frames in the current page. In the simple case, there is only one frame and the page is fully loaded, which means it's probably the right time to send it to the peer. In the not so simple case, it is only one frame out of several. Unfortunately, there is no good way to distinguish between these cases or verify when the last frame has been loaded. So, when is the right time to send a surf buffer to the peer? The answer is every *OnDocumentComplete(),* as long as the browser is not busy anymore. If the browser is busy, it means that more frames are still loading. If it's not busy, it might be done or it might not be done. So, it means that one Cosurfer may send a partial surf buffer to its peer. It might seem counter productive initially, but it actually is a performance booster. The receiving peer starts loading the frames it received and soon gets more frames when the sending Cosurfer completes loading the other frames in the current page. The nasty part is when the receiving Cosurfer completes loading a partial surf buffer, it sends it back to the sender. The sender may be in a more advanced stage already and will treat the partial surf buffer it sent itself just a short while ago as a fresh buffer from the peer and revert back to it. Well, I didn't witness it in practice, so I don't protect against it. If it ever becomes a real problem I can always keep a history of sent buffers and ignore them.

I admit that it is a pretty crazy algorithm to synchronize two dynamic beasts—web browsers loading multiframe pages while users liberally click hyperlinks, hit the back button, and enter new URLs on the address bar. Still, I found this algorithm to be the best way to address these high-uncertainty conditions.

The code includes the Cosurfer solution (Visual Studio .NET 2003), the Cosurfer project itself, the *IE_Puppeteer* project that demonstrates how to control IE, the SocketSpike project that demonstrates some low-level socket programming, and the XmlSpike project that demonstrates parsing and composing XML. I also attach the original .bat files, .rsp, and cordbg.cfg files I used to build and debug Cosurfer using the command-line tools of .NET Framework Beta-1 of several years ago.

DDJ

```
 ...
ConnectionManager connectionManager = new ConnectionManager();
CosurfEngine engine = new CosurfEngine(connectionManager);

connectionManager.AttachSink(engine);
 ...
```

Example 1: AttachSink() method.

```
<?xml version="1.0"?>
<Document>
   <Frame Url="http://www.w3schools.com/tags/planets.htm">
      <Frame Url="http://www.w3schools.com/tags/venus.htm" />
      <Frame Url="http://www.w3schools.com/tags/sun.htm" />
      <Frame Url="http://www.w3schools.com/tags/mercur.htm" />
   </Frame>
</Document>
```

Example 2: XML you get when visiting http://www.w3schools.com/tags/planets.htm.

```
StringWriter sw = new StringWriter();

XmlTextWriter w = new XmlTextWriter(sw);
w.IndentChar = '\t';
w.Indentation = 1;
w.Formatting = Formatting.Indented;

w.WriteProcessingInstruction("xml", @"version=""1.0""");
w.WriteStartElement("Document");
    w.WriteStartElement("Frame");
    w.WriteAttributeString("Url",
                      "http://www.w3schools.com/tags/planets.htm");
        w.WriteStartElement("Frame");
        w.WriteAttributeString("Url",
                      "http://www.w3schools.com/tags/venus.htm");
        w.WriteEndElement();
        w.WriteStartElement("Frame");
        w.WriteAttributeString("Url",
                      "http://www.w3schools.com/tags/sun.htm");
        w.WriteEndElement();
        w.WriteStartElement("Frame");
        w.WriteAttributeString("Url",
                      "http://www.w3schools.com/tags/mercur.htm");
        w.WriteEndElement();

    w.WriteEndElement();
w.WriteEndElement();
```

Example 3: XML processing instruction.

```
public void Listen()
{
    try
    {
    CommCleanup();
        ThreadPool.QueueUserWorkItem(new WaitCallback(SockThreadFunc));
    }
    catch (Exception e)
    {
      Console.WriteLine(
            "*** Exception *** ConnectionManager::ConnectionManager(),
              Description: " + e.Message);
    }
}
void SockThreadFunc(Object state)
{
    Socket listener;
    listener = new Socket(AddressFamily.InterNetwork,
                                SocketType.Stream, ProtocolType.Tcp);
    listener.Blocking = true;
    IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, m_listenPort);
    try
    {
        listener.Bind(endPoint);
    }
    catch(SocketException e)
    {
        // If there is already a listener on this port, listen on another port
        if(e.ErrorCode == 10048)
        {
            endPoint = new IPEndPoint(IPAddress.Any, ++m_listenPort);
            —m_connectPort;

           listener.Bind(endPoint);
        }
        else
        {
            throw e;
        }
    }
    catch(Exception e)
    {
        Console.WriteLine("Exception - {0}", e.Message);
        Debug.Assert(false);
        return;
    }
    m_listening = true;
    m_sink.OnStartListening(m_listenPort);
    listener.Listen(m_listenPort);
    m_sock = listener.Accept();
```

```
    m_listening = false;

    // Notify the sink that a connection was accepted
    m_sink.OnConnectionAccepted();

    m_stream = new NetworkStream(m_sock);
    m_writer = new StreamWriter(m_stream);
    m_writer.AutoFlush = true;

    AsyncCallback readReadyCallback = new AsyncCallback(OnReadReady);
    m_stream.BeginRead(m_inBuff, 0, BUFF_SIZE, readReadyCallback, this);
}
```

Example 4: The communication layer of Cosurfer.

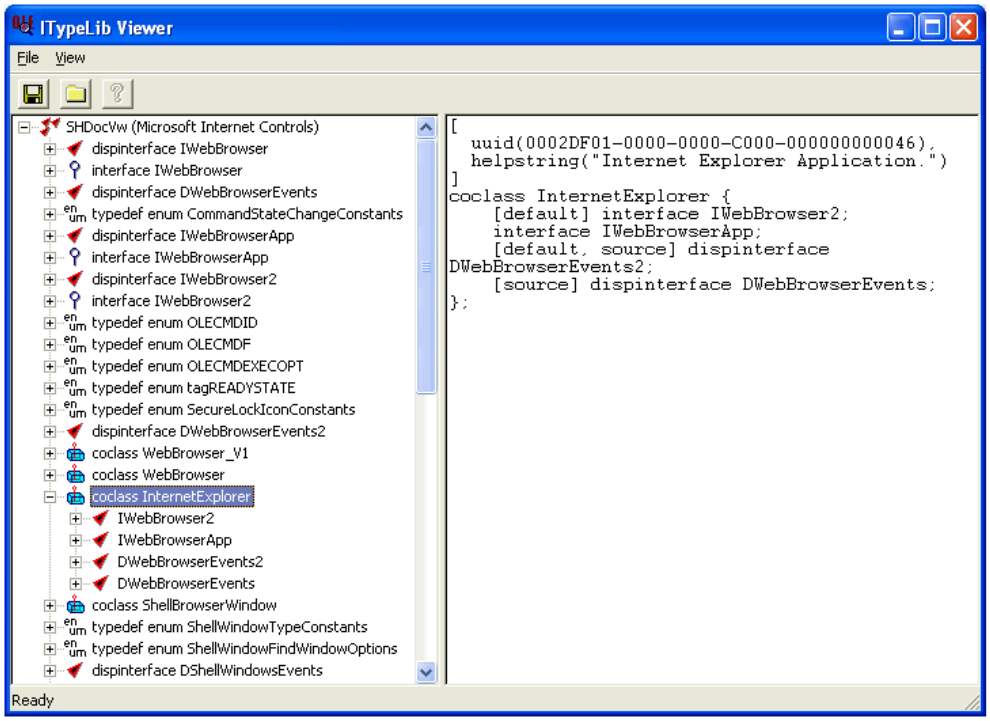

Figure 1: Cosurfer UI.

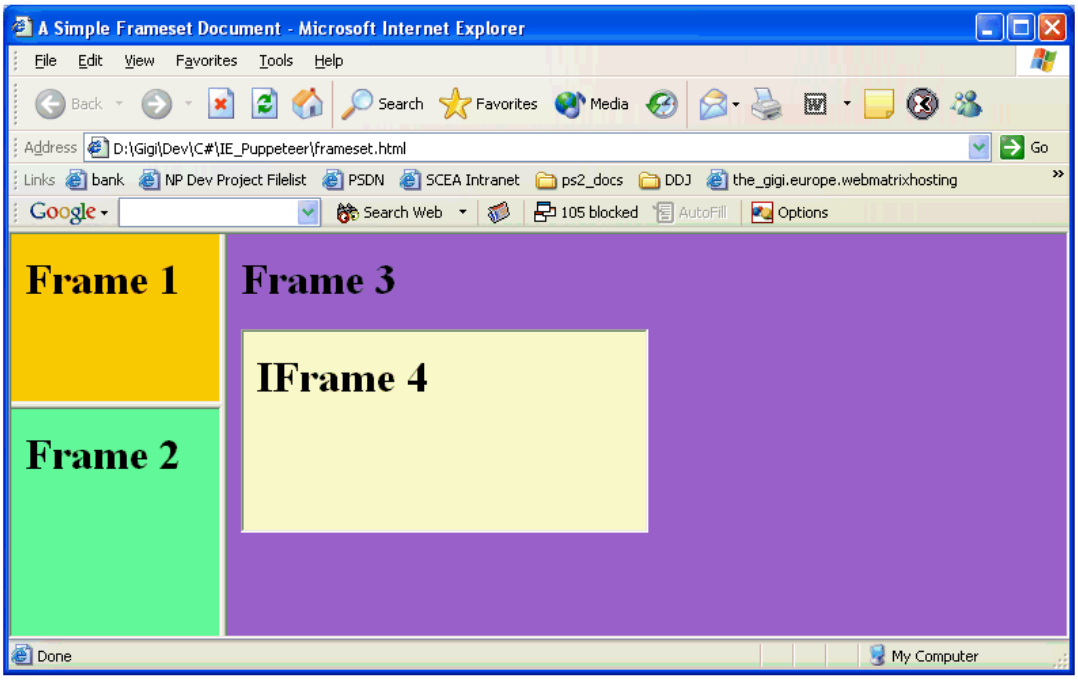

Figure 2: Frames.

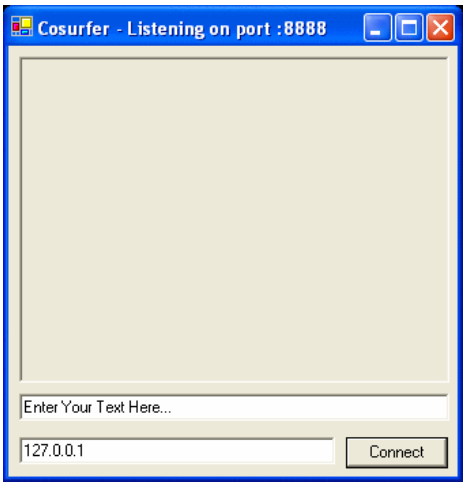

Figure 3: Cosurfer.