RFC 1867 규격을 준수하는 ASP 업로드 컴포넌트 설계

Implementation of an ASP Upload Component to Comply with RFC 1867

강구홍*, 황현주**

서원대학교 컴퓨터정보통신공학부*, 서원대학교 정보통신대학원**

Koo-Hong Kang(khkang@seowon.ac.kr)*, Hyun-Ju Hwang(hunju@wizui.com)**

요약

오늘날 RFC 1867 표준문서를 따르는 HTML POST 폼을 사용해 웹 브라우저를 통해 업로드된 파일을 저장하고 관리하는 ASP 응용들이 다양하게 출시되고 있다. 특히 인터넷의 대중화와 함께 보안이 큰 이슈로 대두되면서 HTTP 포트를 통한 파일 송수신의 중요성이 한층 대두되고 있다. 본 논문에서는 ASP 환경에서 사용 할 수 있는 'Form based ASP 업로드 컴포넌트'를 직접 제작하고 대부분의 주요 코드들을 공개함으로서 향후 업로드 기능을 포함하는 다양한 새로운 ASP 응용들을 개발하는데 활용하도록 하였다. 한편 제작된 업로드 컴포넌트의 업로드 시간 및 CUP 사용시간을 잘 알려진 기존 상용 제품과 비교 분석함으로서 타당성을 검증하였다.

■ 중심어 : | RFC 1867 | 업로드 컴포넌트 | ASP(Active Server Pages) | IIS(Internet Information Server) |

Abstract

Recently many ASP applications have been released which enable them to accept, save and manipulate files uploaded with a web browser. The files are uploaded via an HTML POST form using RFC 1867. In particular, the file transfer via the HTTP port is getting more important because of the current Internet security issues. In this paper, we implement a form-based ASP upload component and disclose explicitly most of the main codes. That is, the open source might be helpful to develop the new ASP applications including file upload function in the future. We also show the upload time and CPU usage time of the proposed upload component and compare with the well-known commercial ones, showing the performance metrics of the proposed component are comparable to those of commercial ones.

■ keyword : | RFC 1867 | Upload Component | ASP(Active Server Pages) | IIS(Internet Information Server) |

I. 서 론

나날이 발전하는 인터넷 환경에서 파일 송수신은 중요하다는 말조차 무색할 정도로 일상화 되었다. 그러나 인터넷의 대중화와 함께 보안이 큰 이슈로 대두되면서 사용자들은 자료 접근에 많은 제약을 받게 되었다.

서버 측의 기본적인 보안 강화의 한 방법은 불필요한 모든 포트를 막아버리는 것이다. 이렇게 하면 네트워크

접수번호 : #051220-001
접수일자 : 2005년 12월 20일
심사완료일 : 2006년 02월 20일
교신저자 : 강구홍 e-mail : khkang@seowon.ac.kr

를 통한 외부의 접근을 원천적으로 차단 할 수 있기 때문에 가장 효과적이다[1]. 그러나 이렇게 포트에 제약을 두는 것은 반대로 사용자가 활용 할 수 있는 소켓을 사용하는 애플리케이션에도 영향을 준다는 것을 뜻한다. 이미 보안을 중요시하는 많은 서버에서는 HTTP(Hypertext Transfer Protocol) 포트를 제외하고는 모두 차단하고 있으며, 심지어는 FTP(File Transfer Protocol) 조차도 허용하지 않고 있는 실정이다. 따라서 몇 년 전부터 파일 송수신을 위해 HTTP 포트를 활용하는 다양한 방법들[2][3]이 제시되고 있고, 그 중에서 바이너리 데이터의 송수신을 위한 방법으로 'RFC 1867-Form-based File Upload in HTML' 문서[4]가 표준화되어 있다.

클라이언트에서 서버로 자료를 전송하기 위해서는 각각 RFC1867을 준수하는 소프트웨어가 필요하다. 다행스럽게도 근래에 사용되는 대부분의 웹 브라우저는 RFC 1867을 지원하기 때문에 사용자는 별다른 조치 없이 바이너리 데이터를 전송할 수 있다. 서버 쪽 처리의 경우에는 웹 서버의 종류에 따라 처리가 세세하게 구분되는 편인데, 보통 '아파치'에서 구동되는 PHP(Hypertext Preprocessor)의 경우에는 스크립트로 대부분의 작업을 처리하고 있다[4]. 그리고 마이크로소프트사의 'IIS (Internet Information Services)'[7]에서 구동되는 ASP(Active Server Pages)[9], ASP.Net[8] 등은 스크립트로 처리 할 수 있지만 대부분 ASP 컴포넌트라 총칭되는 바이너리 레벨의 애플리케이션을 사용하여 처리한다.

본 논문에서는 ASP 환경에서 사용할 수 있는 'Form based ASP 업로드 컴포넌트'를 직접 제작하고 대부분의 주요 코드들을 공개함으로서 향후 업로드 기능을 포함하는 다양한 새로운 ASP 응용들을 개발하는데 활용하도록 하였다. 현재 ASP 환경에서 동작하는 업로드 컴포넌트(이하 '업로드 컴포넌트')는 상용 제품만 하더라도 수십 개에 이른다[2][3]. 그렇지만 각각의 제품 마다 인터페이스가 모두 다르고, 어떤 제품은 함량 미달의 품질을 보인다. 성능 문제는 차치하더라도 업로드 기능을 포함하는 새로운 컴포넌트를 만드는 데 있어서 구조적 문제점으로 인해서 기존 제품을 활용하는 데 어려움이 크다는 것이 중요하다. 한편 제작된 업로드 컴포넌트의 업로드 시간 및 CPU 사용시간을 잘 알려진 기존 상용 제품과 비교 분석함으로서 타당성을 검증하였다.

본 논문은 서론에 이어 제2장에서는 파일 업로드 및 RFC 1867에 대해 간략히 언급하고, 제3장에서는 본 논문에서 제안하는 파일 업로드의 구조를 자세히 설명한다. 제4장에서는 기존 업로드 프로그램과의 성능비교 결과를 제시함으로서 제안된 파일 업로드의 타당성을 보이고, 마지막으로 제5장에서 결론을 맺는다.

II. 파일 업로드 및 RFC 1867

오늘날 파일 업로드 패키지(이하 "파일 업로드"라 부른다)는 RFC 1867을 따르는 HTTP 요구를 파싱한다. 즉 content type이 "multipart/form-data"인 POST 메소드를 사용하는 HTTP 요구가 수신되면, 파일 업로드는 요구를 파싱하고 클라이언트가 쉽게 사용할 수 있는 형태의 결과를 만들어 낸다. 따라서 파일 업로드는 서버 페이지 혹은 웹 애플리케이션에 강력한 고성능 파일 업로드 기능을 제공하게 된다.

파일 업로드는 애플리케이션 요구에 따라 다양한 방법으로 사용될 수 있다. 가장 간단한 경우는 서블렛 요구를 파싱하는 간단한 메소드를 호출하고 애플리케이션에 적용될 아이템 리스트를 처리하는 것이다. 보다 복잡한 예로는, 데이터베이스에 각각의 아이템들을 정렬해 저장하기 위해 파일 업로드를 수정(customize) 할 수도 있다.

파일 업로드 요구는 RFC 1867에 따라 코딩된 아이템 리스트로 구성된다. 파일 업로드는 이 요구를 파싱하고 각 업로드 아이템의 리스트를 애플리케이션에 제공하게 된다. 이때 각 아이템은 인터페이스를 구현하게 된다. 한편 이 아이템들은 애플리케이션에서 필요로 하는 다양한 프로퍼티를 가진다. 예를 들어, 모든 아이템들은 네임(name), content type, 그리고 이 데이터를 엑세스할 수 있는 메소드를 제공한다. 만약 이들 아이템들을 다르게 처리하기 원한다면 인터페이스는 적절한 메소드를 제공하여야 한다.

III. ASP 업로드 컴포넌트 설계

1. 설계 환경 및 클래스

본 논문에서 새로이 제안하는 ASP 업로드 컴포넌트(이하 AronUpload라 부른다) 설계는 Microsoft Windows NT 계열과 IIS5.0 이상에서 동작하도록 설계하였다. 개발 툴로는 Microsoft Visual Studio.NET 2003을 사용하였고 Visual C++로 프로그래밍 하였다. 한편 라이브러리[5]는 ATL7.1과 STL을 사용하였다. AronUpload는 [그림 1]과 같이 크게 6개의 클래스로 구성된다.

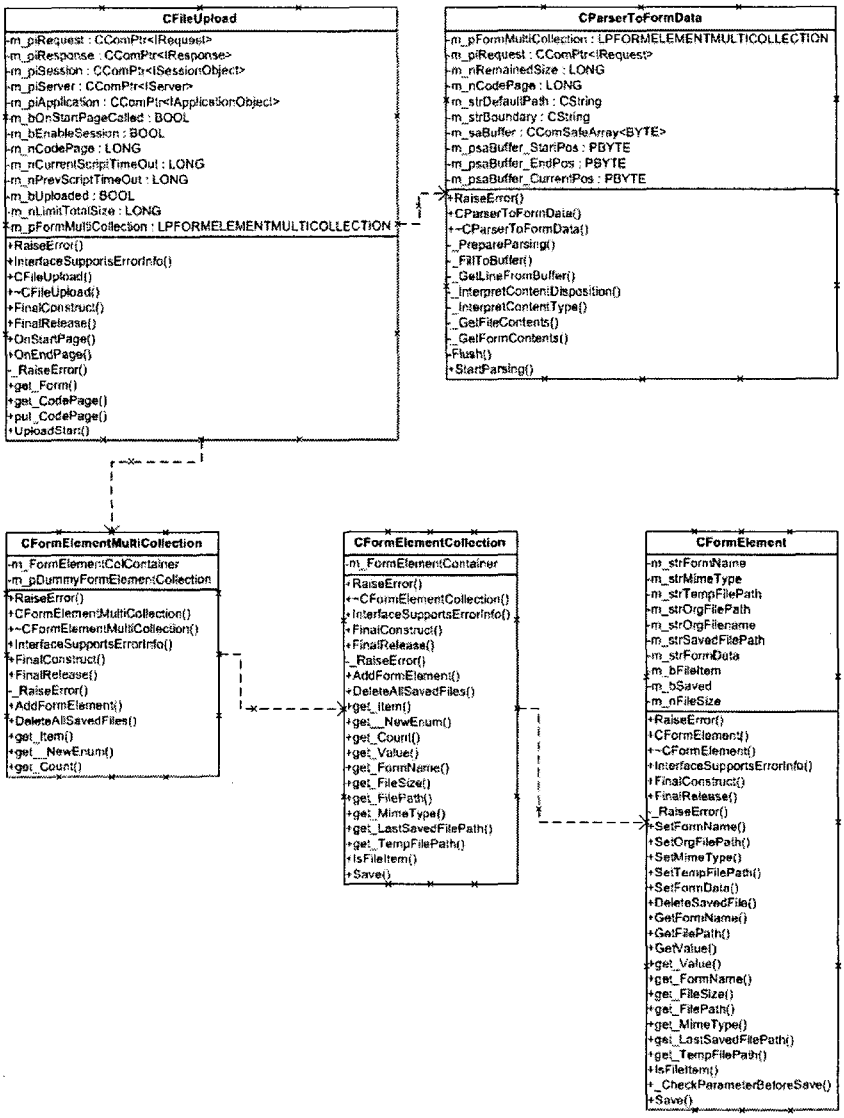


그림 1. 클래스 구조도

표 1. AronUpload를 구성하는 기본 클래스

클래스	설명
CErrorProcedure	*AronUpload*에 존재하는 모든 클래스의 오류처리 기본 클래스
CFile	사용자 인터페이스를 제공하는 클래스. 여기에 사용자가 호출하게 되는 메소드와 프로퍼티를 구현
CParserToFormData	사용자가 전송하는 데이터를 파싱하고 CFormElement 인스턴스를 생성
CFormElementMultiCollection	CParserToFormData가 생성한 CFormElement를 관리한다. 사용자가 CFileUpload를 통해서 요청하는 작업의 대부분은 이 클래스가 관리하는 CFormElement에 의해 처리
CFormElementCollection	CFormElementMultiCollection과 CFormElement의 중간에 위치한다. 같은 폼 네임을 가지는 경우를 위함이다.
CFormElement	바운더리로 구분되는 하나의 폼 요소를 표현한다.

각 클래스별 프로퍼티와 메소드는 [그림 1]의 클래스 구조도를 참조하고 이들 기본 클래스에 대한 설명은 [표 1]과 같다.

2. 오류 처리 구조

본 절에서는 구체적인 구현 설명에 앞서 AronUpload의 전반에 걸쳐 사용되는 오류 처리 메커니즘에 대해 설명한다. 모든 클래스는 CErrorProcedure 클래스를 상속받는다. CErrorProcedure 클래스를 상속한 클래스에서 오류가 발생할 경우 그에 대한 처리를 CErrorProcedure가 도와준다. 다음은 CErrorProcedure의 선언 부분이다.

```
class CErrorProcedure
{
public:
        // 예외 발생 에러 처리
void ExceptionErrorProcedure(LPCTSTR _szErrorPosition);
        // 일반적인 에러 처리
void GeneralErrorProcedure(DWORD _dwErrorCode, ...);
        // 마지막으로 발생한 시스템API의 에러 처리
void GetLastErrorProcedure(LPCTSTR _szModuleName =
NULL);
virtual void RaiseError(CErrorInfo &_clsErrorInfo) = 0;
}
```

상기 3개 메소드는 거의 유사하므로 GeneralErrorProcesure에 대해서만 살펴보겠다. GeneralErrorProcedure의 정의는 다음과 같다.

```
// 일반적인 에러 처리
void    CErrorProcedure::GeneralErrorProcedure(DWORD
_dwErrorCode, ...)
{
        // 로딩한 에러 메시지의 인자들을 채운다.
        va_list argList;

        va_start(argList, _dwErrorCode);

        // 에러 메시지를 로딩한다.
        CString strMessage;
        strMessage                                  =
Utilities::GetMessageFromResourceV(_dwErrorCode,
argList);

        va_end(argList);

        RaiseError(CErrorInfo(ERROR_TYPE_GENERAL +
_dwErrorCode, strMessage, _T("")));
}
```

GeneralErrorProcedure는 사용자가 예측 가능한 오류가 발생했을 때 호출하게 된다. 사용자가 GeneralErrorProcedure를 호출 할 때 해당되는 문자열 테이블의 ID 및 인자들을 넘겨주면 문자열을 생성하게 된다. 한편 RaiseError()는 오류 정보를 설정하고 난 후 해당 오류를 어떻게 처리할 것인지 알려주는 메소드다. 전달 인자로 CErrorInfo 클래스를 생성하여 전달한다. CErrorInfo는 발생한 오류 정보를 관리하는 클래스다. CErrorInfo의 선언은 다음과 같다.

```
class CErrorInfo
{
private:
        LONG m_nCode;
        CString m_strMessage;
        CString m_strDetailMessage;

public:
        CErrorInfo(LONG _nCode, LPCTSTR _szMessage,
LPCTSTR _szDetailMessage = NULL);

        LONG GetCode();
        LPCTSTR GetMessage();
        LPCTSTR GetDetailMessage();
}
```

RaiseError()의 선언을 보면 순수 가상함수다. 즉 RaiseError()의 정의는 CErrorProcedure를 상속하는 클래스에서 하게 된다. 실제로 CErrorProcedure를 상속하는 CFileUpload 클래스를 살펴보면 다음과 같다.

```
class ATL_NO_VTABLE CFileUpload :
        public CComObjectRootEx<CComSingleThreadModel>,
        public CComCoClass<CFileUpload, &CLSID_FileUpload>,
        public ISupportErrorInfo,
        public    IDispatchImpl<IFileUpload,    &IID_IFileUpload,
&LIBID_AronUploadLib, /*wMajor =*/ 1, /*wMinor =*/ 0>,
        public CErrorProcedure
{
public:
...
private:
        void RaiseError(CErrorInfo &_clsErrorInfo);
        void _RaiseError(CErrorInfo &_clsErrorInfo);
        STDMETHOD(UploadStart)(BSTR bstrDefaultPath);
}
```

CErrorProcedure 클래스를 상속받고 순수 가상 함수인 RaiseError() 메소드를 정의하는 것을 볼 수 있다. RaiseError()의 구현은 다음과 같다.

```
void CFileUpload::RaiseError(CErrorInfo &_clsErrorInfo)
{
        throw(_clsErrorInfo);
}
```

RaiseError()의 정의는 프로젝트 및 클래스마다 달라질 수 있다. 중요한 것은 오류 정보를 생성한 후 그것에 대한 처리를 이 메소드에서 할 수 있다는 것이다. 여기에서는 단지 CErrorInfo 클래스를 throw하고 있다. 이제 여기서 던져진 예외정보를 처리하는 메소드를 정의한다.

```
STDMETHODIMP CFileUpload::UploadStart(BSTR bstrDefaultPath)
{
    HRESULT hr = S_OK;

    try
    {
        LONG nUploadedBytes;

        m_piRequest->get_TotalBytes(&nUploadedBytes);

            //지정된 크기 보다 크다면 연결을 끊는다.
        if(nUploadedBytes > m_nLimitTotalSize)
        {
            //연결을 끊는다.
            m_piResponse->End();
            //에러 메시지를 설정하지만 사용자에게 보여지지는 않을 것
이다.

GeneralErrorProcedure(IDS_ERROR_LIMIT_TOTAL_SIZE,
nUploadedBytes, m_nLimitTotalSize);
        }
    }
    catch(CErrorInfo &err)
    {
        _RaiseError(err);
        hr = E_FAIL;
    }
    return hr;
}
```

상기 코드를 보면 if(nUploadBytes > m_nLimitTotalSize)의 조건을 만족하지 못하면 GeneralErrorProcedure()를 호출한다. GeneralErrorProcedure()의 전달 인자로 해당하는 문자열 ID 및 그에 필요한 인자들을 전달하고 있다. GeneralErrorProcedure()의 수행이 완료되면 RaiserError()를 호출하는데, 그것은 앞에서 설명한 바와 같이 단지 throw를 한다. 이렇게 throw된 CErrorInfo는 catch(CErrorInfo&err)에서 처리하게 된다. Catch()에서는 _RaiseError()를 호출하게 된다. _RaiseError()는 COM 객체의 오류가 발생했음을 설정하고 IIS에 알려주는 역할을 한다.

3. 클라이언트가 업로드하는 폼 데이터 처리

클라이언트가 업로드하는 파일 및 텍스트 폼 데이터를 처리하기 위해서는 ASP에서 CFileUpload 객체를 만들고 UploadStart()를 호출한다. CFileUpload는 실제로 사용자가 접근할 수 있는 인터페이스 역할을 하는 클래스다. 다음은 UploadStart()의 정의다.

```
STDMETHODIMP CFileUpload::UploadStart(BSTR bstrDefaultPath)
{
    HRESULT hr = S_OK;

    try
    {
        // 기존에 업로드 되었는지 확인
        {
            if(FALSE != m_bUploaded)
            {
                GeneralErrorProcedure(IDS_ERROR_UPLOADED);
            }
            m_bUploaded = TRUE;
        }

        LONG nUploadedBytes;

m_piRequest->get_TotalBytes(&nUploadedBytes);
```

```
        //지정된 크기 보다 크다면 연결을 끊는다.
        if(nUploadedBytes > m_nLimitTotalSize)
        {
            //연결을 끊는다.
            m_piResponse->End();
            //에러 메시지를 설정하지만 사용자에게 보여지지는 않을 것
이다.

GeneralErrorProcedure(IDS_ERROR_LIMIT_TOTAL_SIZE,
nUploadedBytes, m_nLimitTotalSize);
        }

        CParserToFormData   clsParserToFormData(m_nCodePage,
m_piRequest, OLE2W(bstrDefaultPath), m_pFormMultiCollection);
            clsParserToFormData.StartParsing();
    }
    catch(CErrorInfo &err)
    {
        _RaiseError(err);
        hr = E_FAIL;
    }
        return hr;
}
```

사용자는 이 메소드를 호출하여 업로드 처리를 하게 된다. 이 메소드의 코드 중 상당 부분은 간단한 것이며 실제적인 업로드 처리는 CParserToForData에서 이루어지고 있다. 다음은 CParserToFormData의 선언 일부이다.

```
class CParserToFormData : public CErrorProcedure
{
private:
        CComSafeArray(BYTE) m_saBuffer;
        PBYTE m_psaBuffer_StartPos;
        PBYTE m_psaBuffer_EndPos;
        PBYTE m_psaBuffer_CurrentPos;

void _FillToBuffer(void);
        CString _GetLineFromBuffer(void);
        LONG            _InterpretContentDisposition(CString
_strContentDisposition, LPFORMELEMENT &_pFormElement);
        CString         _InterpretContentType(CString
_strContentType); //ContentType 라인을 해석하는 함수 이 함수는
input이 file일 때만 호출된다.
        CString _GetFileContents(void); //file타입의 데이타를 파일
에 저장하는 함수
        CString _GetFormContents(void);
public:
        void StartParsing(void);
}
```

다음은 StartParsing()의 정의 일부이다.

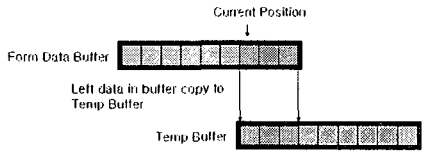
```
void CParserToFormData::StartParsing(void)
{
    // 폼 데이타 객체 생성 및 초기화
    LPFORMELEMENT pFormElement = NULL;

    do
    {
FORMELEMENT::CreateInstance(&pFormElement);
        pFormElement->AddRef();
        if(1                                                    ==
_InterpretContentDisposition(_GetLineFromBuffer(),
pFormElement)) // 리턴 값이 1이면 Input타입이 Text이다..
        {
            // 공백이다 그냥 한 라인 흘리자.
            _GetLineFromBuffer();
        pFormElement->SetFormData(_GetFormContents());
        }
        else // 그 외에 즉, 2라면 Input 타입이 File이다.
        {
pFormElement->SetMimeType(_InterpretContentType(_GetLine
FromBuffer()));
            // 공백이다 그냥 한 라인 흘리자.
            _GetLineFromBuffer();
        pFormElement->SetTempFilePath(_GetFileContents());
        }

        ////하나의 폼 데이타의 처리가 끝났으면 컬렉션에 저장하자.

m_pFormMultiCollection->AddFormElement(pFormElement);
            pFormElement->Release();
            pFormElement = NULL;
    }

while(strFinishBoundary.Compare(_GetLineFromBuffer()) != 0
&& // 이것은 일반적인 조건이다.
        //아래의 조건은 오류 방지를 위한 조건이다.
        (m_nRemainedSize > 0 || m_psaBuffer_CurrentPos <
m_psaBuffer_EndPos));
}
```

상기 코드에서 클라이언트가 언제나 정확하게 폼 데이터 보낸다는 보장이 없으므로 위의 while()의 조건처럼 그에 대한 처리를 해야 한다. 이런 기능이 미비하면 IIS가 멈춰버리는 일이 발생할 수 있다.

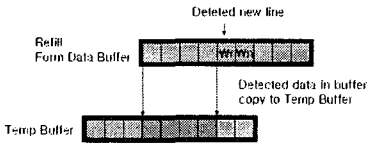
폼 데이터는 크게 ContentDisposition, ContentType (파일일 경우), 데이터, 바운더리로 구성된다. StartParsing은 루프를 돌면서 각각의 부분을 파싱하는 함수를 호출하고 있다. 파싱 방법은 여러 가지가 있을 수 있으므로 본 논문에서는 자세히 다루지 않겠다. 다만 위의 함수들은 버퍼에서 한 라인씩 가져와서 파싱을 하게 되는데, 한 라인씩 가져오는 부분은 버퍼의 경계가 분제될 수 있기 때문에 필요할 경우 임시 버퍼를 생성해서 결과 값을 돌려줘야 한다.

4. 버퍼링

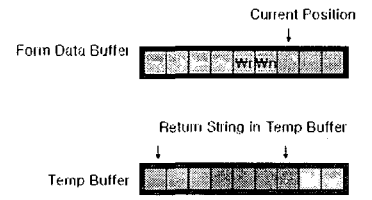
다음 [그림 2]는 메인 버퍼에서 한 줄을 읽어올 수 없을 경우 임시 버퍼를 사용해서 라인을 완성하는 과정을 보여준다. 버퍼링에 관련된 기술들은 소스 코드부분에서 자세한 코멘트 처리를 통해 설명한다.



(a) 버퍼에 남아있는 부분을 임시버퍼에 복사



(b) 메인버퍼를 다시 채우고 개행문자 전까지 복사



(c) 메인버퍼의 현재 위치를 수정하고 임시 버퍼의 값을 넘김

그림 2. 버퍼링 기술

다음은 상기 [그림 2]를 코드로 구현한 _GetLineFromBuffer()의 전문이다.

```
CString CParserToFormData::_GetLineFromBuffer(void)
{
    PBYTE
      pBufferStartPos = NULL,
      pBufferEndPos = NULL;

    vector<BYTE> vecBufferForHeader;
      {
      const BYTE strNULL[2] = {'\r','\n'};

      // 일단 버퍼에서 검색 함 해보자
      const PBYTE psaBuffer_ResultPos =
search(m_psaBuffer_CurrentPos, m_psaBuffer_EndPos,
&strNULL[0], &strNULL[2]);

      //값이 참이면 기존 버퍼에서 한 문장을 찾았다는 얘기 바로 리턴 하자
      if(psaBuffer_ResultPos != m_psaBuffer_EndPos)
      {
        pBufferStartPos = m_psaBuffer_CurrentPos;
        pBufferEndPos = psaBuffer_ResultPos;

        // 버퍼의현재 위치 설정하기
        m_psaBuffer_CurrentPos = psaBuffer_ResultPos;
      }
      else //검색을 했지만 발견되지 않았다면 남아있는 부분을 버퍼에
저장한다.
      {
        vecBufferForHeader.resize((m_psaBuffer_EndPos) -
m_psaBuffer_CurrentPos);

        // 일단 임시 벡터에 복사한다.
        const PBYTE pBufferForHeader_ResultPos =
Utilities::SafePointerCopy(m_psaBuffer_CurrentPos,
m_psaBuffer_EndPos, &vecBufferForHeader[0]);

        //버퍼를 새롭게 채운다.
        _FillToBuffer();

        // \r\n이 걸쳐있을 경우에는 이 코드가 처리한다.
        if('\r' == *(pBufferForHeader_ResultPos) && '\n' ==
*m_psaBuffer_StartPos)
          {

pBufferStartPos = &vecBufferForHeader[0];
        pBufferEndPos = pBufferForHeader_ResultPos;
        m_psaBuffer_CurrentPos = m_psaBuffer_StartPos;
        }
        else
        {
        const PBYTE pBufferForHeader_SearchResultPos =
search(m_psaBuffer_CurrentPos, m_psaBuffer_EndPos ,
&strNULL[0], &strNULL[2]);
        pBufferStartPos = &vecBufferForHeader[0];

        // 크기를 늘린다.

vecBufferForHeader.resize(vecBufferForHeader.size() +
(pBufferForHeader_SearchResultPos- m_psaBuffer_StartPos));
                                    pBufferEndPos =
Utilities::SafePointerCopy(m_psaBuffer_StartPos,
pBufferForHeader_SearchResultPos,
pBufferForHeader_ResultPos);

m_psaBuffer_CurrentPos = pBufferForHeader_SearchResultPos
;
        }
      }
    }

    return Utilities::MultiToWideChar(m_nCodePage,
(PCHAR)pBufferStartPos, (PCHAR)pBufferEndPos);
}
```

5. 인터페이스 정의

다음은 AronUpload의 실제 사용자 페이지의 사용 예이다.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<form name="write_form" enctype="multipart/form-data"
method="post" action="Process.asp">
    <input type="text" name="txtvalue" ID="Text1"><br>
    <input type="file" name="file"><br>
    <input type="submit" name="Upload" value="Upload"><br>
</form>
</BODY>
</HTML>

서버 페이지
<%@ Language=VBScript %>
<%
set AronUpload =server.CreateObject("AronStaff.FileUpload")
AronUpload.UploadStart "C:\TEMP"

AronUpload("file").Save "C:\Download Folder", True
Response.Write AronUpload("txtvalue")


Set AronUpload=nothing
%>
```

다음은 AronUpload의 인터페이스 정의이다.

```
// AronUpload.idl : AronUpload의 IDL 소스입니다.
//

// 이 파일은 MIDL 도구에 의해 처리되어
// 형식 라이브러리(AronUpload.tlb) 및 마샬링 코드가 생성됩니다.

import "oaidl.idl";
import "ocidl.idl";

[
    object,
    uuid(a817e7a2-43fa-11d0-9e44-00aa00b6770a),
    dual,
    helpstring("IComponentRegistrar 인터페이스"),
    pointer_default(unique)
]
interface IComponentRegistrar : IDispatch
{
    [id(1)]     HRESULT Attach([in] BSTR bstrPath);
    [id(2)]     HRESULT RegisterAll();
    [id(3)]     HRESULT UnregisterAll();
    [id(4)]     HRESULT GetComponents([out]
SAFEARRAY(BSTR)* pbstrCLSIDs, [out] SAFEARRAY(BSTR)*
pbstrDescriptions);
    [id(5)]     HRESULT RegisterComponent([in]
BSTR bstrCLSID);
    [id(6)] HRESULT UnregisterComponent([in] BSTR
bstrCLSID);
}

[
    object,
    uuid(FADEBD86-FA13-4E93-B605-D7C68AC8C665),
    dual,
    nonextensible,
    helpstring("IFileUpload 인터페이스"),
    pointer_default(unique)
]
interface IFileUpload : IDispatch
{
    // 표준 서버측 구성 요소 메서드입니다.
    HRESULT OnStartPage([in] IUnknown* piUnk);
    HRESULT OnEndPage();
    [propget, id(DISPID_VALUE), helpstring("속성 Form")]
HRESULT Form([out, retval] VARIANT* pVal);
    [propget, id(1), helpstring("속성 CodePage")] HRESULT
CodePage([out, retval] long* pVal);
    [propput, id(1), helpstring("속성 CodePage")] HRESULT
CodePage([in] long newVal);
    [id(101), helpstring("메서드 UploadStart")] HRESULT
UploadStart([in] BSTR bstrDefaultPath);
}
[
    object,
    uuid(2E9C5D7D-2022-4078-A327-ADDA7E451A4A),
    dual,
    nonextensible,
    helpstring("IFormElement 인터페이스"),
    pointer_default(unique)
]
interface IFormElement : IDispatch{
    [propget, id(DISPID_VALUE), helpstring("속성 Value")]
HRESULT Value([out,retval] BSTR* pVal);
    [propget, id(1), helpstring("속성 FormName")] HRESULT
FormName([out, retval] BSTR* pVal);
    [propget, id(3), helpstring("속성 FileSize")] HRESULT
FileSize([out, retval] LONG* pVal);
    [propget, id(4), helpstring("속성 FilePath")] HRESULT
FilePath([out, retval] BSTR* pVal);
    [propget, id(5), helpstring("속성 MimeType")] HRESULT
MimeType([out, retval] BSTR* pVal);
    [propget, id(7), helpstring("속성 LastSavedFilePath")]
HRESULT LastSavedFilePath([out, retval] BSTR* pVal);
    [propget, id(8), helpstring("속성 TempFilePath")]
HRESULT TempFilePath([out, retval] BSTR* pVal);
    [id(101), helpstring("메서드 IsFileItem")] HRESULT
IsFileItem([out,retval] VARIANT_BOOL* pVal);
    [id(102), helpstring("메서드 Save")] HRESULT Save([in,
defaultvalue("")] BSTR bstrPath, [in,defaultvalue(FALSE)]
VARIANT_BOOL bOverwrite, [out,retval] BSTR* bstrRet);
}
[
    object,
    uuid(C623C3FC-3B03-4B9F-9B71-B935E7E78513),
    dual,
    nonextensible,
    helpstring("IFormElementCollection 인터페이스"),
    pointer_default(unique)
]
interface IFormElementCollection : IDispatch{
    [propget, id(DISPID_VALUE), helpstring("속성 Item")]
HRESULT Item([in,defaultvalue("0")] VARIANT varIndex, [out,
retval] VARIANT* pVal);
    [propget, id(DISPID_NEWENUM), helpstring("속성
_NewEnum")] HRESULT _NewEnum([out, retval] LPUNKNOWN*
pVal);
    [propget, id(1), helpstring("속성 Count")] HRESULT
Count([out, retval] LONG* pVal);
    [propget, id(2), helpstring("속성 Value")] HRESULT
Value([out,retval] BSTR* pVal);
    [propget, id(3), helpstring("속성 FormName")] HRESULT
FormName([out, retval] BSTR* pVal);
```

```
        [propget, id(4), helpstring("속성 FileSize")] HRESULT
FileSize([out, retval] LONG* pVal);
        [propget, id(5), helpstring("속성 FilePath")] HRESULT
FilePath([out, retval] BSTR* pVal);
        [propget, id(6), helpstring("속성 MimeType")] HRESULT
MimeType([out, retval] BSTR* pVal);
        [propget, id(7), helpstring("속성 LastSavedFilePath")]
HRESULT LastSavedFilePath([out, retval] BSTR* pVal);
        [propget, id(8), helpstring("속성 TempFilePath")]
HRESULT TempFilePath([out, retval] BSTR* pVal);
        [id(101), helpstring("메서드 IsFileItem")] HRESULT
IsFileItem([out,retval] VARIANT_BOOL* pVal);
        [id(102), helpstring("메서드 Save")] HRESULT Save([in,
defaultvalue("")] BSTR bstrPath, [in,defaultvalue(FALSE)]
VARIANT_BOOL bOverwrite, [out,retval] BSTR* bstrRet);
}
[
        object,
        uuid(DB95977C-F963-4B4B-8000-C02D6E775AA1),
        dual,
        nonextensible,
        helpstring("IFormElementMultiCollection 인터페이스"),
        pointer_default(unique)
]
interface IFormElementMultiCollection : IDispatch{
        [propget, id(DISPID_VALUE), helpstring("속성 Item")]
HRESULT Item([in] VARIANT varIndex, [out, retval]
IFormElementCollection** pVal);
        [propget, id(DISPID_NEWENUM), helpstring("속성
_NewEnum")] HRESULT _NewEnum([out, retval] LPUNKNOWN*
pVal);
        [propget, id(1), helpstring("속성 Count")] HRESULT
Count([out, retval] LONG* pVal);
}
[
        uuid(FDE775A9-4873-46C2-B1A4-12E9FCB30176),
        version(1.0),
        helpstring("AronUpload 1.0 형식 라이브러리"),

custom(a817e7a1-43fa-11d0-9e44-00aa00b6770a,"{7822B
E4F-0F29-49F1-9E0D-4BB61D24AB85}")
]
library AronUploadLib
{
        importlib("stdole2.tlb");

        interface IFormElement;
        interface IFormElementCollection;
        interface IFormElementMultiCollection;

        [

uuid(7822BE4F-0F29-49F1-9E0D-4BB61D24AB85),
            helpstring("ComponentRegistrar 클래스")
        ]
        coclass CompReg
        {
            [default] interface IComponentRegistrar;
        }
        [

uuid(F70BD38D-DAC6-4867-89DD-52B13E978F42),
            helpstring("FileUpload Class")
        ]
        coclass FileUpload
        {
            [default] interface IFileUpload;
        }
}
```

IV. 성능평가

1. 시험 환경

본 논문에서 설계한 AronUpload의 성능 평가를 위해 비교 대상으로 두개의 상용 컴포넌트 DEXTUpload Professional 버전 3.1[2]과 TABSUpload 버전 2.0[3]을 선택했다. 한편 서버로 사용된 컴퓨터 사양은 다음과 같다.

- CPU: AMD Athlon64 3000+
- RAM: Samsung DDR3200 512MB*2
- HDD: Seagage 7200.7 S-ATA
- LAN: 100Mbps

먼저 시험을 위해 업로드 서버 와 클라이언트 컴퓨터 1대를 100 Mbps 스위칭 허브로 연결하고 동일한 조건에서 이들 세 가지 업로드 컴포넌트를 시험하기 위해 1대의 클라이언트 컴퓨터만 사용하였다. 클라이언트 컴퓨터에서 서버 쪽으로 업로드 이벤트를 발생시키기 위해 업로드 시뮬레이션 프로그램인 uptester.exe[6]을 사용하였으며 실제 상황을 반영하기 위해 동시에 여러 HTTP 연결을 맺은 후 업로드를 진행하였다. 성능 평가를 위해 업로드 완료시점까지 걸린 전체 시간과 각 HTTP 연결을 통해 업로드된 평균시간을 구하였다. 다음은 본 논문에서 사용한 uptester.exe의 명령어다.

uptester -s 192.168.1.2 -u /sample/upload.asp -f sample.zip -t 4 -i 10

여기서, -s 옵션은 서버 IP 주소, -u 옵션은 업로드 상대 URL, -f 옵션은 업로드할 파일 이름, -t 옵션은 동시에 업로드할 쓰레드 개수, 그리고 -i 옵션은 쓰레드별 업로드 반복 횟수가 된다. 이제 uptester.exe는 다음과 같은 폼을 사용해 서버로 데이터를 전송하게 된다.

```
<HTML>
<BODY>
<form method="POST" enctype="multipart/form-data"
action="process.asp">
<input type="file" name="uploadFile">
<input type="submit">
</form>
</BODY>
</HTML>
```

이때 action 부분은 -u 옵션에서 지정한 URL이 사용되며 -f 옵션에 지정된 파일이 첨부된다. 한편 서버 측 ASP 프로그램은 동시에 업로드 되는 파일을 각각 저장할 수 있도록 overwrite 옵션을 사용하지 않았으며 업로드를 처리하는 ASP 페이지는 다음과 같다.

```
<%
Dim AronUpload
Set AronUpload = Server.CreateObject("AronStaff.FileUpload")
AronUpload.UploadStart "C:\TEMP"
%>
```

2. 기존 업로드 컴포넌트와 성능 비교

2.1 업로드 시간

클라이언트 컴퓨터에서 uptester.exe를 사용해 하나의 쓰레드를 10회 반복해 업로드를 시도하였다. 이때 업로드 파일 크기는 100KB, 1MB, 10MB, 그리고 100MB로 설정하여 파일 크기에 따른 평균 업로드 시간을 측정하였다[표 2].

표 2. 업로드 파일 크기 변화에 따른 업로드 평균 소요시간 (초) (쓰레드 1개)

전송크기	100KB	1MB	10MB	100MB
Aron	14.5	89.56	889	8985.1
DEXT	17.2	92.2	889.56	9176.9
TABS	15	90.6	884.9	9229.7

[표 2]는 동일한 조건에서 세 번 반복 시험을 통해 결과 값을 얻었다 (이하, 본 논문을 통해 제시되는 모든 결과 값들은 세 번의 반복 시험을 통한 평균값을 나타내었음을 밝힌다).

실질적으로 서버는 다수의 클라이언트를 관리하기 때문에 업로드 또한 동시 다발적으로 이루어진다. 이러한 환경을 반영하기 위해 클라이언트 컴퓨터에서 uptester.exe를 사용해 열개의 쓰레드를 10회 반복해 업로드를 실시하였다. [표 3]은 업로드 파일 크기 변화에 따른 업로드 평균 소요시간을 보여준다.

표 3. 업로드 파일 크기 변화에 따른 업로드 평균 소요시간 (초) (쓰레드 10개)

전송크기	100KB	1MB	10MB	100MB
Aron	857.9	8035	84320	864232
DEXT	896.5	7768	84238	850826
TABS	871.3	8305	84006	856707

[표 2]와 [표 3]을 통해 본 논문에서 개발한 AronUpload 컴포넌트가 기존 상용 제품과 비교해 성능상 문제가 없음을 확인할 수 있다.

2.2 CPU 점유 시간

본 논문에서 개발된 컴포넌트가 얼마나 효율적으로 파싱할 수 있는지 알아보기 위해 기존 컴포넌트들과 함께 CPU 점유율을 조사하였다. 시험을 위해 2MB 크기의 파일을 업로드하였으며 윈도우즈의 작업관리자를 이용해 CPU 사용 시간을 조사하였다. CPU 사용 시간은 해당 프로세스가 CPU를 1초 동안 점유율 100%를 유지하는 경우 1초가 된다. [그림 3]은 1개의 쓰레드가 1000번 반복해 업로드할 경우 평균 CPU 점유 시간을 알기위해 윈도우즈 작업관리자를 캡쳐한 화면이다.

동일 작업을 처리해도 CPU 점유율이 더 낮다는 것은 파싱 알고리즘이 그만큼 효율적이라고 볼 수 있다. [그림 4]는 1개의 쓰레드와 10개의 쓰레드를 사용하여 1000번 반복해 업로드할 경우 AronUpload와 IIS 프로세스가 점유한 CPU 시간의 합을 보여준다. [그림 4]로부터 AronUpload 컴포넌트가 기존 상용 제품과 비교해 성능상 문제가 없음을 확인할 수 있다.

그림 3. 1개의 쓰레드가 1000번 반복해 업로드한 경우 CPU 점유 시간(dllhost.exe: 업로드 프로세스, inetinfo.exe: IIS 프로세스)

2.3 버퍼 크기에 따른 CPU 점유 시간

컴포넌트의 성능에 영향을 주는 또 다른 요인으로는 버퍼의 크기가 있다. 컴포넌트는 클라이언트에서 전송되

는 데이터를 버퍼에 일시 저장한 후 파싱을 한다. 이 때 사용하는 버퍼의 크기에 따라서 성능이 어떻게 변하는지 확인해 본다. DEXT나 TABS는 버퍼의 사이즈를 변경할 수 없기 때문에 AronUpload만으로 진행했다.

보통은 버퍼가 클수록 CPU 사용 시간이 줄어들 것이라고 예상 할 수 있다. 하지만 [그림 5]를 보면 8KB일 때 가장 높은 효율을 보이고 있으며 버퍼의 크기가 증가할수록 CPU 사용 시간이 조금씩 증가하는 것을 볼 수 있다.

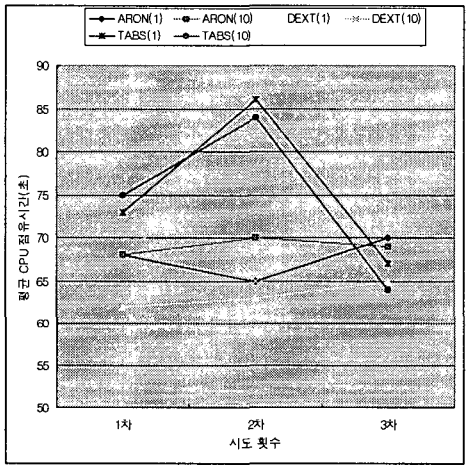


그림 4. 1000번 반복해 업로드한 경우 CPU 점유 시간 (초)
(범례의 괄호안의 숫자는 쓰레드 개수를 나타냄)

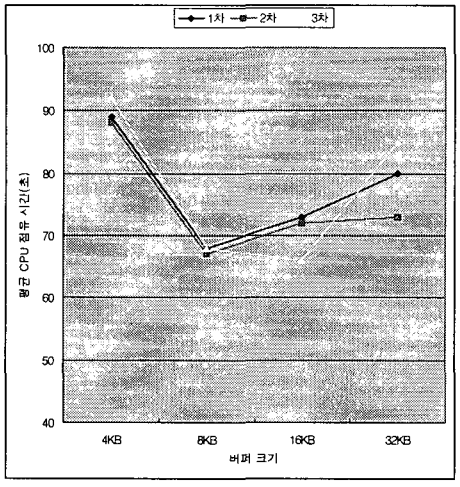


그림 5. 버퍼 크기에 따른 CPU 점유 시간 추이

버퍼의 크기를 키우면 효율이 증가하는 것은 매우 일반적인 것이다. 두 번에 걸쳐서 해야 할 작업을 한 번으로 줄이면 그 만큼의 작업이 줄어들기 때문이다. 따라서 4KB에서 8KB로 버퍼의 크기가 증가할 때 효율이 증가하는 것은 충분히 이해할 수 있다. 문제가 되는 것은 8KB 이후 버퍼의 크기가 증가하면서 역으로 효율이 떨어지는 것인데 이것은 메모리 단편화에 의한 버퍼의 할당과 해제에 들어가는 비용이 문제가 되는 것이다.

AronUpload 내부에서 사용하는 버퍼를 스택 메모리에 확보하고 파싱이 끝날 때까지 재활용 한다고 가정하더라도 IIS에서 데이터를 가져오는 과정에서 SafeArray를 반드시 사용해야 하며 이 과정에서 힙 메모리를 재할당하는 일은 피할 수 없다. 따라서 무조건 버퍼의 크기를 늘린다고 성능이 좋아지는 것이 아니며 사용되는 CPU와 OS 및 현재 프로세스의 메모리 단편화 정도에 따라서 최적의 버퍼 크기는 달라질 수 있다.

V. 결 론

현재 ASP 환경에서 동작하는 업로드 컴포넌트는 상용 제품만 하더라도 수 십 개에 이른다. 그렇지만 각각의 제품마다 인터페이스가 모두 다르고 어떤 제품은 함량미달의 품질을 보인다. 더욱이 아직 믿을만한 소스가 공개되지 않은 상태 있다. 본 논문에서는 ASP 환경에서 사용 할 수 있는 'Form based ASP 업로드 컴포넌트'를 직접 제작하고 대부분의 주요 코드들을 공개함으로서 '웹 에디터'와 같은 향후 업로드 기능을 포함하는 다양한 새로운 ASP 응용들을 개발하는데 활용하도록 하였다. 한편 제작된 업로드 컴포넌트의 성능분석을 위해 기존 상용제품과 함께 업로드 시간 및 CUP 사용시간을 조사하였다. 시험 결과 본 논문에서 제작한 컴포넌트가 기존 상용제품과 성능 면에서 거의 유사함을 보였다. 또한 버퍼 크기에 따른 업로드 시 CPU 사용 시간을 조사함으로서 메모리 단편화의 영향을 직접 확인할 수 있었으며 사용 컴퓨팅 환경에 따라 최적의 버퍼 크기를 적절히 설정할 필요성이 있음을 보였다.

참 고 문 헌

[1] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2003.

[2] http://www.dextupload.com/product/product_dext01.asp

[3] http://www.tabslab.co.kr/kr/product/upload10

[4] E. Nebel and L. Masinter, *Form-based File Upload in HTM*, IETF RFC 1867, Nov. 1995.

[5] 전병선, *Component Development Visual C++ & With ATL*, 영진닷컴, 2004.

[6] http://www.tabslab.com/kr/product/upload10/compare.asp

[7] http://www.microsoft.com/windowsserver2003/community/centers/iis/default.mspx

[8] http://www.asp.net/QuickStart/aspnet/Default.aspx

[9] G. Buczek, *ASP Developer's Guide*, McGraw-Hill, 1999.

저 자 소 개

강 구 홍(Koo-Hong Kang) 정회원

- 1985년 8월 : 경북대학교 전자공학과(공학사)
- 1990년 2월 : 충남대학교 전자공학과(공학석사)
- 1998년 2월 : 포항공과대학교 전자계산학과(공학박사)
- 2000년 9월~현재 : 서원대학교 컴퓨터정보통신공학부 조교수

<관심분야> : 컴퓨터 네트워크, 네트워크 프로그래밍

황 헌 주(Hyun-Ju Hwang) 준회원

- 2003년 2월 : 서원대학교 컴퓨터공학과(공학사)
- 2006년 2월 : 서원대학교 정보통신대학원(공학석사)
- 2006년 1월~현재 : (주)위즈도메인 연구원

<관심분야> : GUI, 네트워크 프로그래밍