AS-2 Implementation

Name: (Deon) Guo Tian
ID: 3309011
UPI: tguo002

Abstract

Applicability Statement 2 (AS2) is an Internet Engineering Task Force (IETF) standard for the secure exchange of structured business data using HTTP transfer protocol. This project was an implementation of AS2 using the iterative and prototyping methodologies. AS2 was implemented in C#, using a Microsoft Visual Studio 2005 development environment.

The first phase of the project was to implement a program that could send and receive messages, construct messages, and query and update an underlying database according to the AS2 protocol. A standard .net application was used to send messages using the HTTP post method, and an .ashx file was used to receive them. The underlying database and database layer were implemented with Pervasive RDBMS and then migrated to SQL 2000. IIS 5.3 server was used for the server-client mechanism.

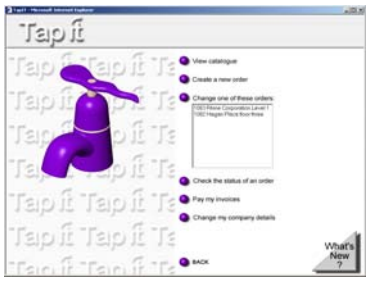
The second phase was to add security features in the AS2 protocol, the most important being digital signature and encryption. A simple, non-AS2 program with digital signature and encryption was first developed using a public key cryptography. The keys were stored as X509 certificates in Windows certificate stores. The program was then integrated into the basic AS2 program from the first phase.

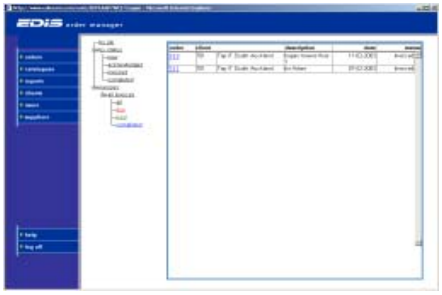
1. Background

1.1 Who is AARN

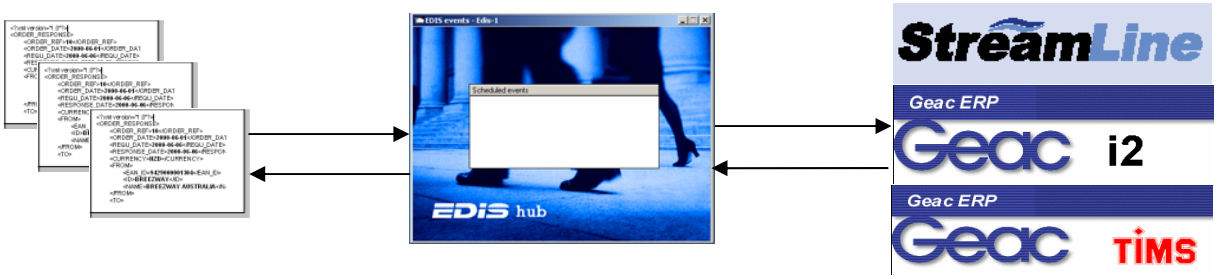
AARN innovation is a software research and development organization. It provides complete, end-to-end, manufacturing and distribution e-commerce solutions that include both:

- Web-enabled applications that only require the user to have access to a web-browser and a connection to the Internet. The applications below are few examples of Web-enabled applications.





- Integrated translators that move information into and out of manufacturing and financial systems. EDIS for Windows is one of them.



AARN is the sponsor company for this AS2 project and EDIS for windows is one of AARN's core e-commerce products.

1.2 Brief Introduction to EDIS E-commerce Trading System

Computers have been used in the business world since the 1950's. In the 1990's, the focus shifted from enterprise computing to the internet. Recently, the idea of a computer-supported business between businesses (B2B) has attracted fresh interest.

EDIS for Windows is a message exchange system. Figure 3.1 below shows how EDIS works in the business world.

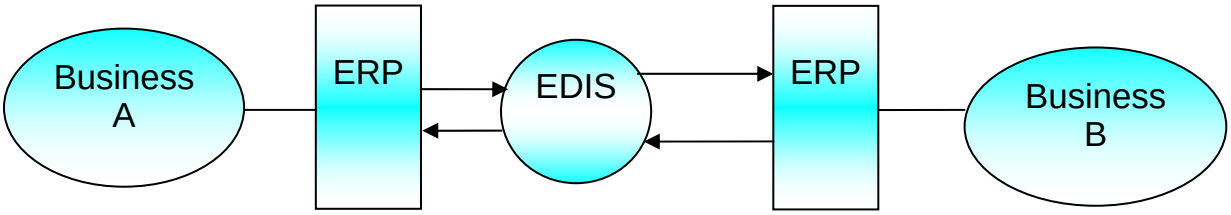


Figure EDIS in business world

From the figure above, Business A and Business B both have their own ERP (enterprise resource planning) systems. The two ERP systems are different in most cases. EDIS sits in between the two ERP systems and handles the data exchange.

The figure below shows the structure of the EDIS for Windows systems. It consists of different modules, some of which form groups to perform discrete aspects of the system's overall functionality.

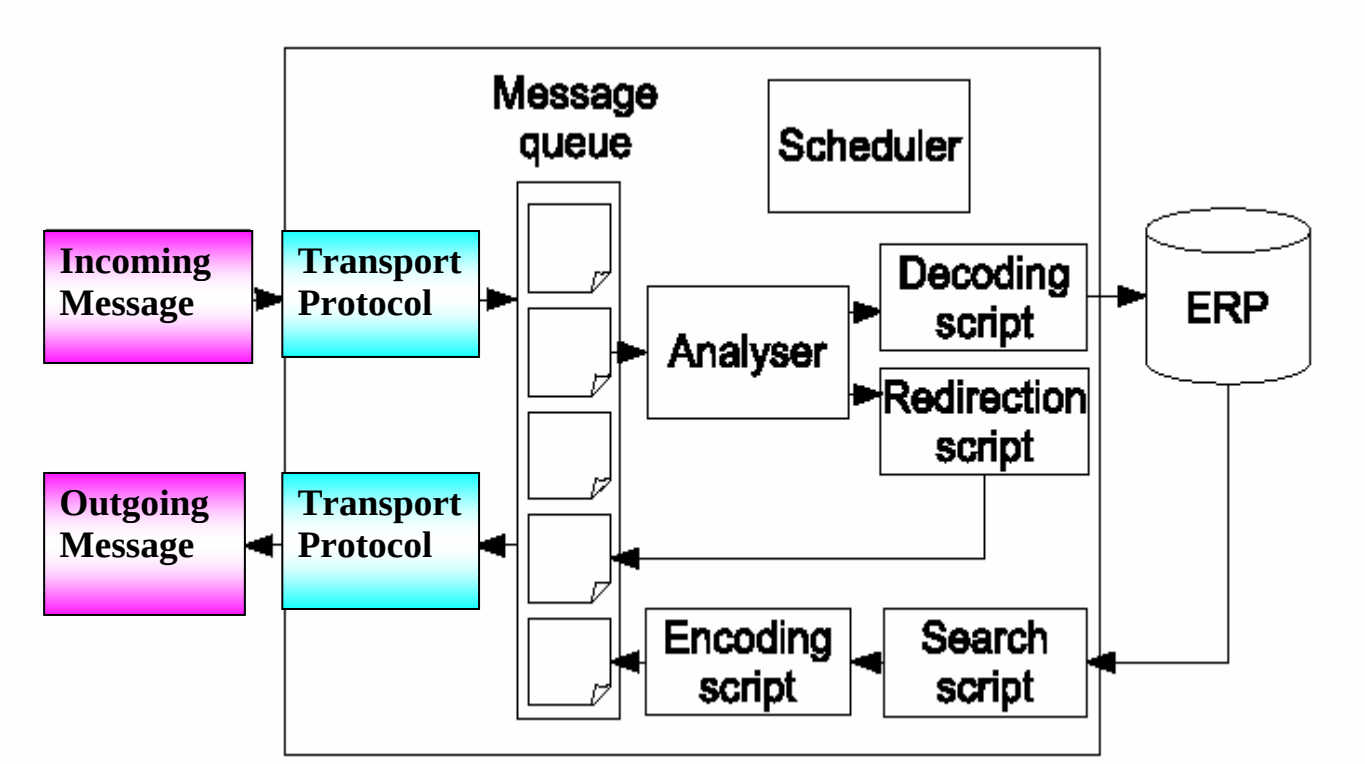


Figure Overview of the EDIS for Windows system [1]

The modules on the right hand side translate the messages and handle the data flow between the message queue and the ERP system. At the centre of this system is the Message Queue module. The modules on the left hand side (that are highlighted) handle incoming and outgoing messages and they are what this project is about: retrieve data from the database, format the message, securely connect to the server, send outgoing message, process incoming message, etc.

1.3 Initiative for AARN

AS2 presents both a major opportunity as well as a competitive risk to AARN Innovation. With the current global focus on e-commerce security and scalability, the traditional VAN model of e-business is being challenged. Worldwide, corporations are adopting communications protocols that utilize Digital Certificates to enhance security. They are also switching to a peer-to-peer network model that more closely emulates the Internet architecture than the traditional, proprietary, VAN communications platform.

Today, leading retailers and manufacturers are realizing the benefits of AS2. This list of companies includes: Wal-Mart, Shaw's, Target, Lowe's, Wegmans, Procter & Gamble, Hershey Foods, Campbell's and many others. Many of these organizations are actively requesting that all of their supply chain partners utilize this technology to communicate across the business community.

Figure - companies using AS2 [2]

From November 2004 through to February 2005, AARN was privileged to be able to facilitate a research and development project at the University of Auckland. The three students produced development reports that provided valuable input into the decision-making process. This has resulted in AARN being able to determine that developing the software in-house is practical and that gaining an international AS2-compliance is achievable. [3]

2. Introduction to AS2

AS2 is Applicability Statement 2 for short. It describes how to exchange structured business data securely using the HTTP transfer protocol. Structured business data may be XML or other structured data formats. The data is packaged using standard MIME structures. Authentication and data confidentiality are obtained by using Cryptographic Message Syntax with S/MIME security body parts. Authenticated acknowledgements make use of multipart/signed Message Disposition Notification (MDN) responses to the original HTTP message. This applicability statement is informally referred to as "AS2" because it is the second applicability statement, produced after "AS1".

In this chapter AS2 is discussed in detail. There are mainly 5 aspects, that is: The overall operation of AS2, the assumptions (conditions) to be made for AS2, the structure of an AS2 message and MDN.

2.1 Overall Operation

The image below shows how AS2 works in a typical data exchange between two trading partners.

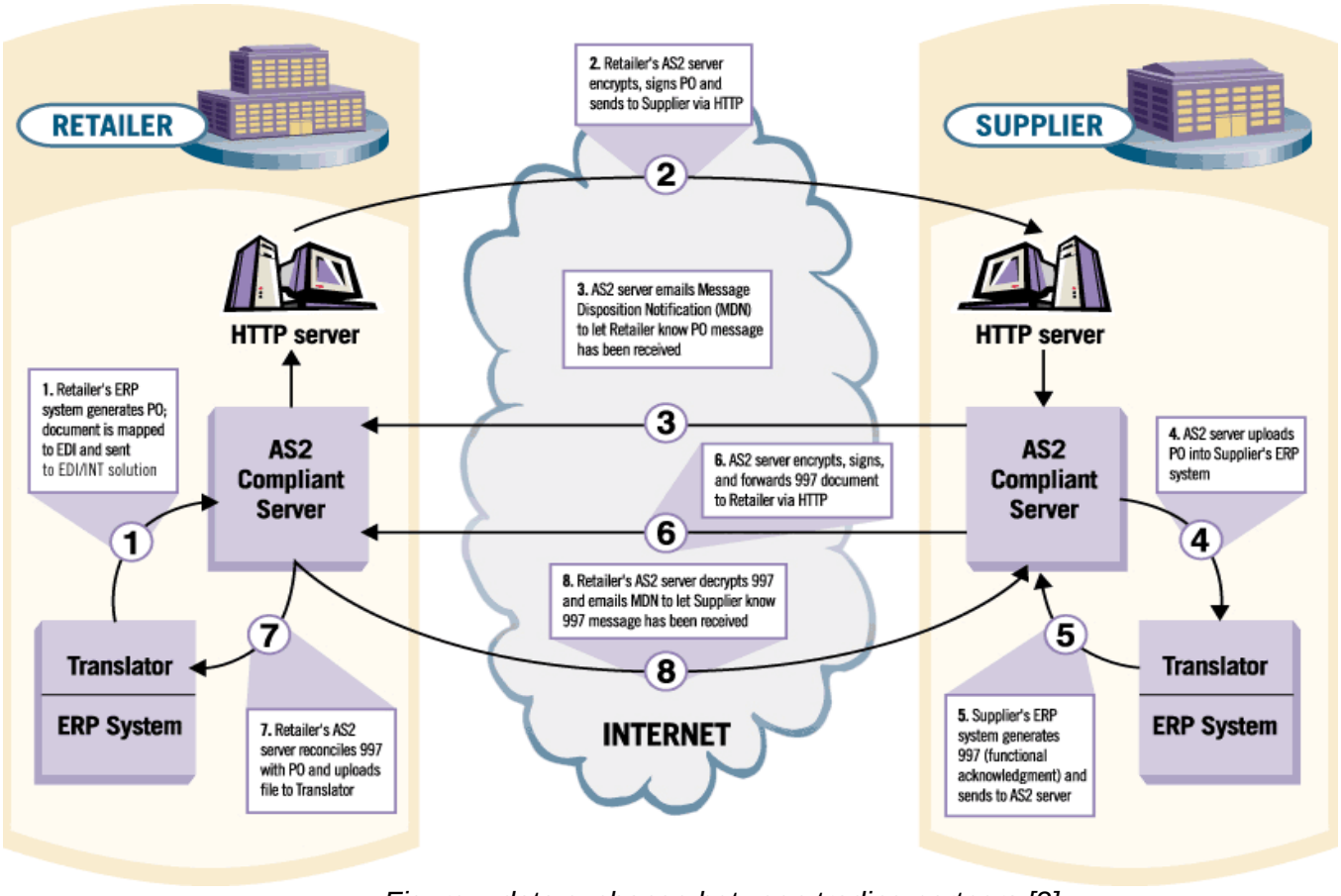


Figure – data exchange between trading partners [2]

As shown in the figure above, there are usually 8 steps for one data exchange

Step	Task	Location
1	Retailer's ERP system generates PO (purchase order). The document is then mapped to EDI and sent to AS2 server	Retailer side
2	Retailer's AS2 server encrypts, signs the PO and sends to the Supplier via HTTP	Retailer side – Internet
3	Supplier's AS2 server receives the AS2 message from the Retailer and sends Message Disposition Notification (MDN) back to let the Retailer know that the PO message has been received.	Supplier side -- Internet
4	Supplier's AS2 server uploads PO into Supplier's ERP (Enterprise resource planning) system.	Supplier side
5	Supplier's ERP system generates functional acknowledgement (997) and sends to the Supplier's AS2 server.	Supplier side
6	Supplier's AS2 server encrypts, signs and forwards the 997 to Retail via HTTP	Supplier side -- Internet
7	Retailer's AS2 server reconciles the 997 with PO and uploads file to ERP	Retailer side
8	Retailer's AS2 server decrypts 997 and sends over MDN to let Supplier know that 997 message has been received.	

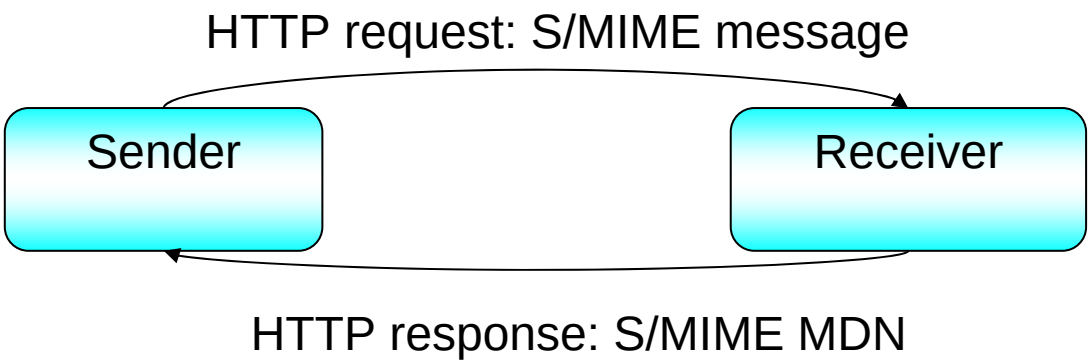
An HTTP POST operation is used to send appropriately packaged EDI, XML, or other business data. The Request-URI identifies a process for unpacking and handling the message data and for generating a reply for the client that contains a message disposition acknowledgement (MDN), either signed or unsigned. The MDN is either returned in the HTTP response message body or by a new HTTP POST operation to a URL for the original sender.

This request/reply transactional interchange can provide secure, reliable, and authenticated transport for EDI or other business data using HTTP as a transfer protocol.

The security protocols and structures used also support auditable records of these document data transmissions, acknowledgements, and authentication.

2.1.1 The Secure Transmission Loop

In the transmission process, one organization sends a signed and encrypted EDI/EC (Electronic Data Interchange/Electronic Commerce) interchange to another organization and requests a signed receipt, and later the receiving organization sends this signed receipt back to the sending organization. Such a process is called “Secure Transmission Loop” and its main steps are:



- The organization sending EDI/EC data signs and encrypts the data using S/MIME. In addition, the message will request that a signed receipt be returned to the sender. To support NRR, the original sender retains records of the message, message-ID, and digest (MIC) value.

- The receiving organization decrypts the message and verifies the signature, resulting in verified integrity of the data and authenticity of the sender.

- The receiving organization then returns a signed receipt using the HTTP reply body or a separate HTTP POST operation to the sending organization in the form of a signed message disposition notification. This signed receipt will contain the hash of the received message, allowing the original sender to have evidence that the received message was authenticated and/or decrypted properly by the receiver.

The above describes functionality that, if implemented, will satisfy all security requirements and implement non-repudiation of receipt for the exchange.

2.2 Assumptions

There were some assumptions made when AS2 was designed. These assumptions must be satisfied in implementation. The assumptions are mainly made in the aspect of EDI/EC Process and Flexibility.

2.2.1 EDI/EC Process Assumptions

- Encrypted object is an EDI/EC Interchange.

AS2 assumes that a typical EDI/EC interchange is the lowest-level object that will be subject to security services.

- EDI envelope headers are encrypted.

Congruent with the above statement, EDI envelope headers are NOT visible in the MIME package.

In many cases, some envelope information is set to be visible for Optimization purpose. In commercial EDI networks (Value Added Networks or VANs), the visible information of the envelope helps to optimize routing and makes it more efficient. However, AS2 does not support for this optimization. This means EDI envelope headers are not visible in the MIME package and no exception is allowwed. [4]

2.2.2 Flexibility Assumptions

- Encrypted or Unencrypted Data

AS2 allows for EDI/EC message exchange in which the EDI/EC data can be either encrypted or not.

- Signed or Unsigned Data

AS2 allows for EDI/EC message exchange with or without digital signature of the original EDI transmission.

- Optional Use of Receipt

AS2 allows for EDI/EC message transmission with or without a request for receipt notification. A signed receipt notification is requested; however, a MIC value is REQUIRED as part of the returned receipt, except when a severe error condition prevents computation of the digest value. In the exceptional case, a signed receipt should be returned with an error message that effectively explains why the MIC is absent.

- Use of Synchronous or Asynchronous Receipts

In addition to a receipt request, AS2 allows the specification of the type of receipt that should be returned. It supports synchronous or asynchronous receipts in the MDN format.

- Hash Function, Message Digest Choices

When a signature is used, it is RECOMMENDED that the SHA-1 hash algorithm be used for all outgoing messages, and that both MD5 and SHA-1 be supported for incoming messages. [4]

2.2.3 Permutation Summary

From the assumptions stated above, an AS2 transmission can be permuted with the 4 optional conditions: The message is signed or not, the message is encrypted or not, requests an MDN or not and the requested MDN is signed or not.

In summary, the following twelve security permutations are possible in any given trading relationship:

Case	**Signed message**	**Encrypted message**	**Request MDN**	**Signed MDN**
1	No	No	No	-
2	No	No	Yes	No
3	No	No	Yes	Yes
4	No	Yes	No	-
5	No	Yes	Yes	No
6	No	Yes	Yes	Yes
7	Yes	No	No	-
8	Yes	No	Yes	No
9	Yes	No	Yes	Yes
10	Yes	Yes	No	-
11	Yes	Yes	Yes	No
12	Yes	Yes	Yes	Yes

Cases:

1. Sender sends un-encrypted data and does NOT request a receipt.

2. Sender sends un-encrypted data and requests an unsigned receipt. Receiver sends back the unsigned receipt.

3. Sender sends un-encrypted data and requests a signed receipt. Receiver sends back the signed receipt.

4. Sender sends encrypted data and does NOT request a receipt.

5. Sender sends encrypted data and requests an unsigned receipt. Receiver sends back the unsigned receipt.

6. Sender sends encrypted data and requests a signed receipt. Receiver sends back the signed receipt.

7. Sender sends signed data and does NOT request a signed or unsigned receipt.

8. Sender sends signed data and requests an unsigned receipt. Receiver sends back the unsigned receipt.

9. Sender sends signed data and requests a signed receipt. Receiver sends back the signed receipt.

10. Sender sends encrypted and signed data and does NOT request a signed or unsigned receipt.

11. Sender sends encrypted and signed data and requests an unsigned receipt. Receiver sends back the unsigned receipt.

12. Sender sends encrypted and signed data and requests a signed receipt. Receiver sends back the signed receipt.

Users can choose any of the twelve possibilities, but only the last case (12), when a signed receipt is requested, offers the whole suite of security features described in Section "The Secure Transmission Loop".

Additionally, the receipts discussed above may be either synchronous or asynchronous depending on the type requested. The use of either the synchronous or asynchronous receipts does not change the nature of the secure transmission loop in support of NRR.

2.3 Structure of an AS2 Message

The basic structure of an AS2 message consists of MIME format inside an HTTP message with a few additional specific AS2 headers.

2.3.1 Headers

Internet EDI MIME Message types and Content-type header

The EDI MIME message can be any of the following types:

- No encryption, no signature
- No encryption, signature
- Encryption, no signature
- Encryption, signature
- MDN over HTTP, no signature
- MDN over HTTP, signature
- MDN over SMTP, no signature
- MDN over SMTP, signature

Although all MIME content types should be supported, the following MIME content types are essential and must be supported:

Case	Content-type
1	multipart/signed
2	multipart/report
3	message/disposition-notification
4	application/PKCS7-signature
5	application/PKCS7-mime
6	application/EDI-X12
7	application/EDIFACT
8	application/edi-consent
9	application/XML

Http headers

- Final Recipient and Original Recipient

The final and original recipient values SHOULD be the same value. These values MUST NOT be aliases or mailing lists.

- Message-Id and Original-Message-Id

Message-Id and Original-Message-Id is formatted as "<" id-left "@" id-right ">"

Message-Id length is a maximum of 998 characters. For maximum backward compatibility, Message-Id length SHOULD be 255 characters or less. Message-Id SHOULD be globally unique, and id-right SHOULD be something unique to the sending host environment (e.g., a host name).

When sending a message, always include the angle brackets. Angle brackets are not part of the Message-Id value. For maximum backward compatibility, when receiving a message, do not check for angle brackets. When creating the Original-Message-Id header in an MDN, always use the exact syntax as received on the original message; don't strip or add angle brackets.

- Host Header

The host request header field MUST be included in the POST request made when sending business data. This field is intended to allow one server IP address to service multiple hostnames, and potentially to conserve IP addresses.

- Content-Transfer-Encoding Not Used in HTTP Transport

HTTP can handle binary data and so there is no need to use the content transfer encodings of MIME. However, a content transfer encoding value of binary or 8-bit is permissible but not required. The absence of this header MUST NOT result in transaction failure. Content transfer encoding of MIME body parts within the AS2 message body is also allowed. [4]

Additional AS2-Specific HTTP Headers

The following headers are to be included in all AS2 messages and all AS2 MDNs, except for asynchronous MDNs that are sent using SMTP and that follow the AS1 semantics.

- AS2 Version Header

To promote backward compatibility, AS2 includes a version header:

AS2-Version: 1.0

Used in all implementations of this specification. 1.x will be interpreted as 1.0 by all implementations with the "AS2 Version: 1.0" header. That is, only the most significant digit is used as the version identifier for those not implementing additional non-AS2-specified functionality. "AS2-Version: 1.0 through 1.9" may be used. All implementations MUST interpret "1.0 through 1.9" as implementing this specification However, an implementation MAY extend this specification with additional functionality by specifying versions 1.1 through 1.9. If this mechanism is used, the additional functionality must be completely transparent to implementations with the "AS2-Version: 1.0" designation.

AS2-Version: 1.1
Designates those implementations that support compression.

Receiving systems MUST NOT fail due to the absence of the AS2-Version header. Its absence would indicate that the message is from an implementation based on a previous version of this specification.

- AS2 System Identifiers

To aid the receiving system in identifying the sending system,
AS2-From and AS2-To headers are used.

AS2-From: < AS2-name >
AS2-To: < AS2-name >

These AS2 headers contain textual values, identifying the sender/receiver of a data exchange. Their values may be company specific, such as Data Universal Numbering System (DUNS) numbers, or they may be simply identification strings agreed upon between the trading partners.

There is no required response to a client request containing invalid or unknown AS2-From or AS2-To header values. The receiving AS2 system MAY return an unsigned MDN with an explanation of the error, if the sending system requested an MDN.

2.3.2 Message body

The message body is just stored as a string at this stage.

2.4 HTTP considerations

AS2 operations are all based on HTTP protocol so there are many HTTP issues that we must consider. A few of the most important HTTP considerations are listed below:

2.4.1 HTTP Response Status Codes

The status codes return status concerning HTTP operations. For example, the status code 401, together with the WWW-Authenticate header, is used to challenge the client to repeat the request with an Authorization header.

For errors in the request-URI, 400 ("Bad Request"), 404 ("Not Found"), and similar codes are appropriate status codes. A careful examination of these codes and their semantics should be made before implementing any retry functionality. Retries SHOULD NOT be made if the error is not transient or if retries are explicitly discouraged. [4]

2.4.2 HTTP Error Recovery

If the HTTP client fails to read the HTTP server response data, the POST operation with identical content, including same Message-ID, SHOULD be repeated, if the condition is transient.

The Message-ID on a POST operation can be reused if and only if all of the content (including the original Date) is identical.

Details of the retry process (including time intervals to pause, number of retries to attempt, and timeouts for retrying) are implementation dependent. These settings are selected as part of the trading partner agreement.

Servers SHOULD be prepared to receive a POST with a repeated Message-ID. The MIME reply body previously sent SHOULD be resent, including the MDN and other MIME parts. [4]

2.5 Important Security Features of AS2

The most important Security features of AS2 are digital signature and encryption. Both of them are implemented using the Public key cryptography.

2.5.1 Public Key Cryptography and Encryption

The Public-Private key algorithm requires a pair of keys: Public Key and the Private Key. The private key is kept secret, while the public key may be widely distributed. In a sense, one key "locks" a lock; while the other is required to unlock it. It should not be feasible to deduce the private key of a pair given the public key, and in high quality algorithms no such technique is known.

The simple example [5] demonstrates how public-private key algorithm works:

Sam has a pair of keys: The Public Key and the Private Key.


(Sam's public key)


Sam


(Sam's private key)

Sam's Co-workers:




Pat Doug Susan


Anyone can get Sam's Public Key, but Sam keeps his Private Key to himself

Sam's Public key is available to anyone who needs it, but he keeps his Private Key to himself. Nobody else other than Sam himself has access to Sam's Private Key. Keys are used to encrypt information. Encrypting information means "scrambling it up", so that only

a person with the appropriate key can make it readable again. For a pair of keys, if one key is used to encrypt some data, the encrypted data then can only be decrypted by the other of this pair.

Susan (shown below) can encrypt a message using Sam's Public Key. Sam uses his Private Key to decrypt the message. Any of Sam's coworkers might have access to the message Susan encrypted, but without Sam's Private Key, nobody can decrypt it.

"Sam, let's meet secretly at 7pm at the Quad."

MIIBNwYJKoZIhvcN
AQcDoIIBKDCCASQ
CAQAxgckwgcYCAQ
AwLzAaMQswCQYD
VQQGEwJVUzELMA
kGA1UEAx……..

MIIBNwYJKoZIhvcN
AQcDoIIBKDCCASQ
CAQAxgckwgcYCAQ
AwLzAaMQswCQYD
VQQGEwJVUzELMA
kGA1UEAx……..

"Sam, let's meet secretly at 7pm at the Quad."

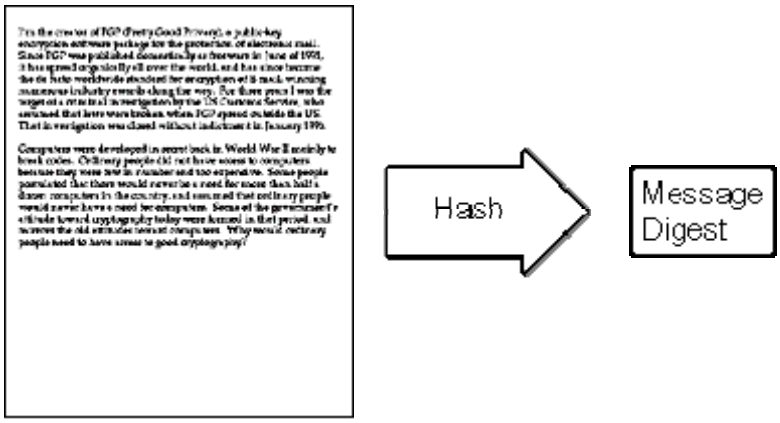
Other people may find a way to see the message Susan sent to Sam:

MIIBNwYJKoZIhvcN
AQcDoIIBKDCCASQ
CAQAxgckwgcYCAQ
AwLzAaMQswCQYD
VQQGEwJVUzELMA
kGA1UEAx……..

But they cannot decrypt it without Sam's Private Key.

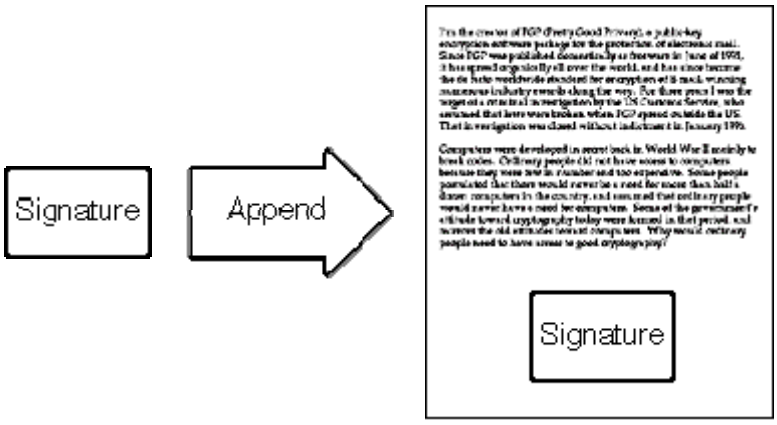
2.5.2 Public Key Cryptography and Digital Signature

With his private key, Sam can put digital signatures on documents and other data. A digital signature is a "stamp" Sam places on the data which is unique to Sam, and is very difficult to forge. In addition, the signature assures that any changes made to the data that has been signed can not go undetected.



To sign a document, Sam's software will crunch down the data into just a few lines by performing a hash. These few lines are called a message digest. (It is not possible to change a message digest back into the original data from which it was created.)

Sam then encrypts the message digest with his private key. The result is the digital signature.

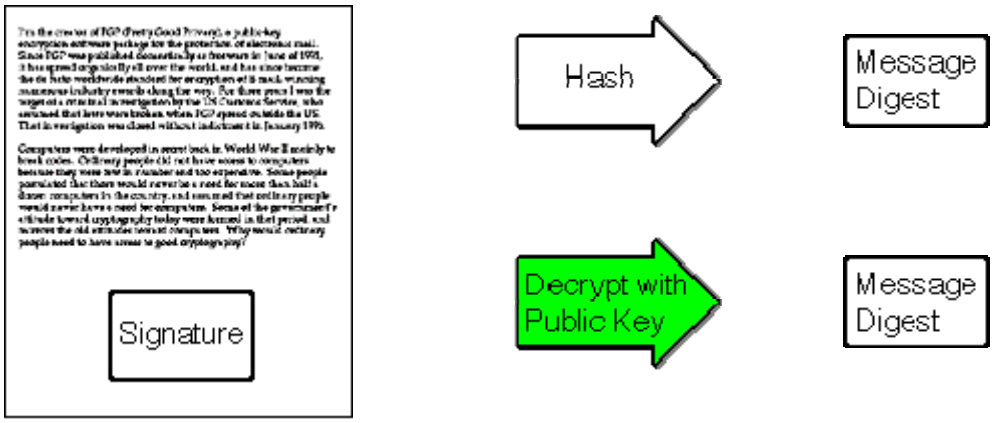


Finally, Sam appends the digital signature to document. All of the data that was hashed has been signed.

Sam now passes the document on to Pat.

First, Pat decrypts the signature (using Sam's public key) changing it back into a message digest. If this worked, then it proves that Sam signed the document,

because only Sam has his private key. Pat then hashes the document data into a message digest. If the message digest is the same as the message digest created when the signature was decrypted, then Pat knows that the signed data has not been changed.

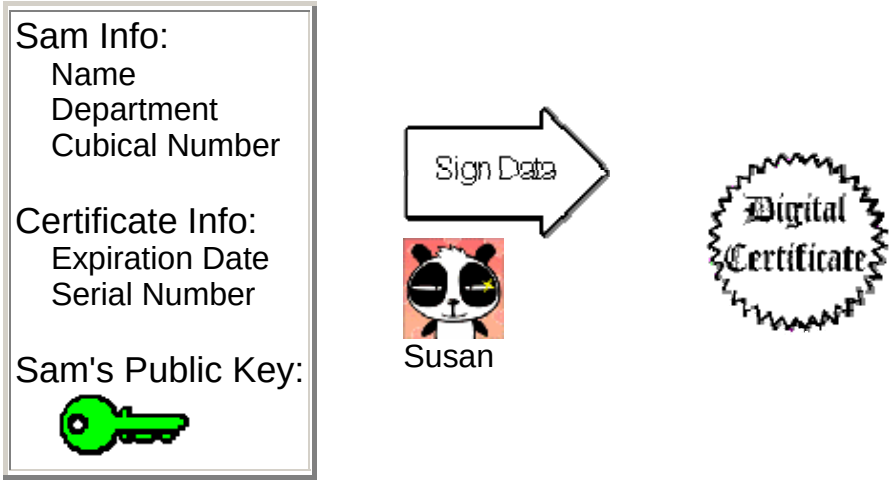


We get two Message Digests: one is from decrypting the digital signature and the other is from hashing the document data.

2.5.3 Digital Certificate

Doug wishes to deceive Pat. Doug makes sure that Pat receives a signed message and a public key that appears to belong to Sam. Unbeknownst to Pat, Doug deceitfully sent a key pair he created using Sam's name. Short of receiving Sam's public key from him in person, how can Pat be sure that Sam's public key is authentic?

It just so happens that Susan works at the company's certificate authority center. Susan can create a digital certificate for Sam simply by signing Sam's public key as well as some information about Sam.



Now Sam's co-workers can check Sam's trusted certificate to make sure that his public key truly belongs to him. In fact, no one at Sam's company accepts a signature for which

there does not exist a certificate generated by Susan. This gives Susan the power to revoke signatures if private keys are compromised, or no longer needed. There are even more widely accepted certificate authorities that certify Susan.

Let's say that Sam sends a signed document to Pat. To verify the signature on the document, Pat's software first uses Susan's (the certificate authority's) public key to check the signature on Sam's certificate. Successful de-encryption of the certificate proves that Susan created it. After the certificate is de-encrypted, Pat's software can check if Sam is in good standing with the certificate authority and that all of the certificate information concerning Sam's identity has not been altered.
Pat's software then takes Sam's public key from the certificate and uses it to check Sam's signature. If Sam's public key de-encrypts the signature successfully, then Pat is assured that the signature was created using Sam's private key, for Susan has certified the matching public key. And of course, if the signature is valid, then we know that Doug didn't try to change the signed content.

2.5.4 Certificate Authority of AS2

Trading partners can self-certify each other if an agreed-upon certification authority is not used. AS2 does not require the use of a certification authority. The use of a certification authority is therefore optional.

In the case trading partners are Sam and Susan:
Certify each other's public key

Now Susan's public key is Certified by Sam's private key and no one can fake Sam's public key.

Now Sam's public key is Certified by Susan's private key and no one can fake Susan's public key.

2.5.5 In the process of one AS2 message exchange:

Assume Sam  is the sender and Susan  is the receiver of this exchange.

	Who	Do	Key used	Comments
1		Certify Susan	Sam's Private Key Susan's Public Key	Sam uses his own Private Key to sign on Susan's Public Key along with some information of Susan. This gives a Digital Certificate for Susan
1		Certify Sam	Susan's Private Key Sam's Public Key	Sam uses his own Private Key to sign on Susan's Public Key along with some information of Susan. This gives a Digital Certificate for Susan
2		Encrypts the message to be sent	Susan's Public Key	
3		Use Hash fuction to get the message digest		This message digest is used to form the Digital signature
4		Encrypt the Message Digest	Sam's Private Key	This creates the Digital signature
5		Attaches the Digital signature to the Message		
6		Send over the Message		
7		Verify Sam's Certificate	Susan's Public Key	Because Susan used her own Private Key to sign on Sam's Public Key
8		Decrypt the Digital Signature	Sam's Public Key	This gives the Message Digest
9		Decrypt the Message data	Sam's Public Key	This gives the "original" Message Data but Susan still cannot make sure if the message data is modified by someone else in the middle of the transfer.
10		Perform Hash function on Message data to get the Message Digest		This gives another Message Digest. Compare this Message Digest with the one got from step 8
11		If two Message Digests are identical, it means the message is from Sam and has not been modified		Now Susan is certain that the Decrypted message data is the original message from Sam
11		If two Message Digests are not identical, it means the message is modified by someone else		Now Susan knows the Decrypted message data cannot be trusted. It has been modified by someone else.

2.6 Structure and Processing of an MDN Message

In order to support non-repudiation of receipt, a signed receipt, based on digitally signing a message disposition notification, is to be implemented by a receiving trading partner's UA. The message disposition notification, specified by RFC 3798, is digitally signed by a receiving trading partner as part of a multipart/signed MIME message. [4]

2.6.1 Required supports

The following support for signed receipts is REQUIRED:

	Support needed for signed receipts
1	The ability to create a multipart/report; where the report-type = disposition-notification.
2	The ability to calculate a message integrity check (MIC) on the received message. The calculated MIC value will be returned to the sender of the message inside the signed receipt.
3	The ability to create a multipart/signed content with the message disposition notification as the first body part, and the signature as the second body part.
4	The ability to return the signed receipt to the sending trading partner.
5	The ability to return either a synchronous or an asynchronous receipt as the sending party requests.

2.6.2 Usage of the signed receipt

The signed receipt is used to notify a sending trading partner that requested the signed receipt that:

	Usage of the Signed receipt
1	The receiving trading partner acknowledges receipt of the sent EC Interchange
2	If the sent message was signed, then the receiving trading partner has authenticated the sender of the EC Interchange.
3	If the sent message was signed, then the receiving trading partner has verified the integrity of the sent EC Interchange.

2.6.3 Processes on receiving an encrypted message

Regardless of whether the EDI/EC Interchange was sent in S/MIME format, the receiving trading partner's UA MUST provide the following basic processing:

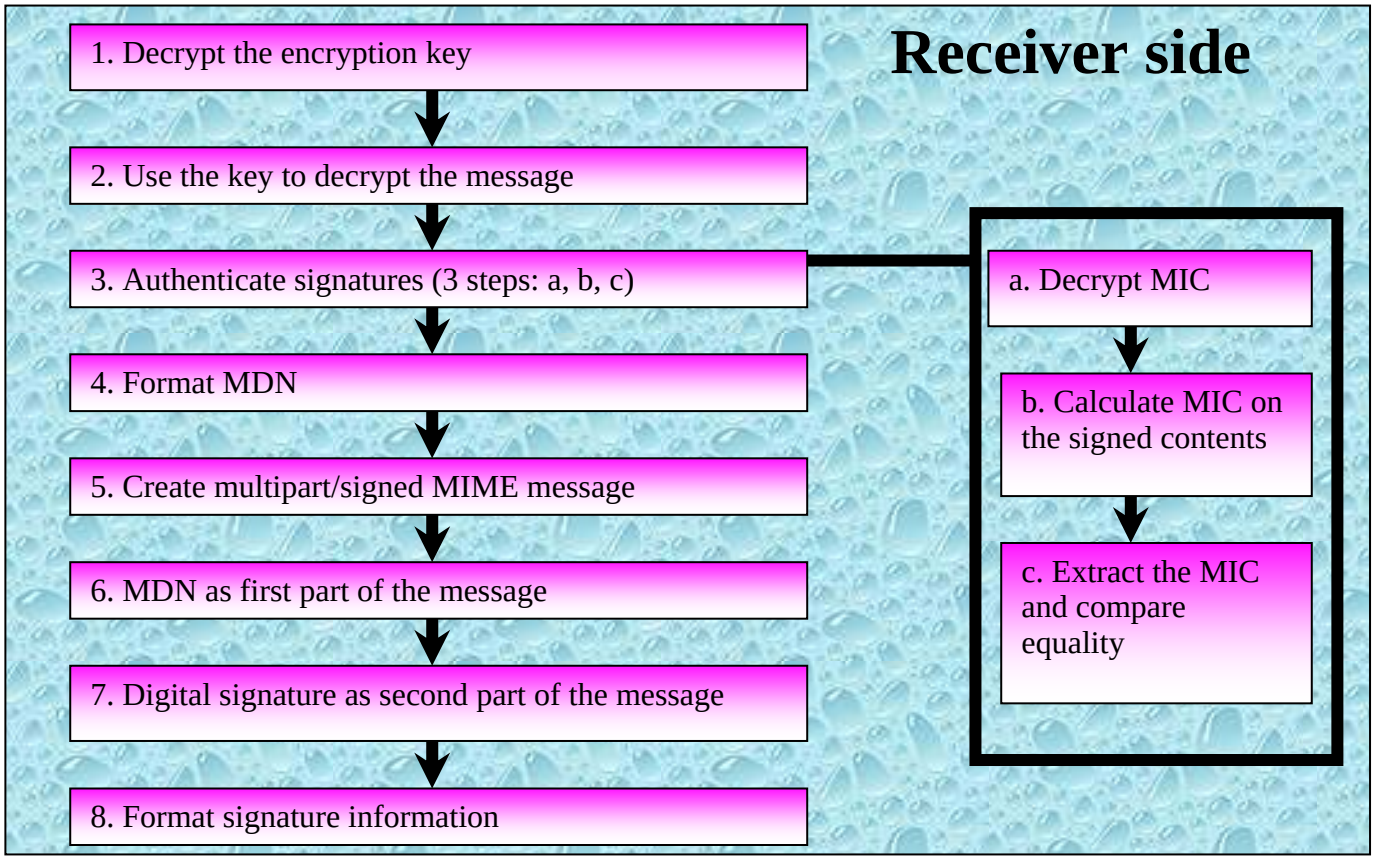


Figure - Processes on receiving an encrypted message

1. If the sent EDI/EC Interchange is encrypted, then the encrypted symmetric key and initialization vector (if applicable) is decrypted using the receiver's private key.

2. The decrypted symmetric encryption key is then used to decrypt the EDI/EC Interchange.

3. The receiving trading partner authenticates signatures in a message using the sender's public key. The authentication algorithm performs the following:

a. The message integrity check (MIC or Message Digest), is decrypted using the sender's public key.

b. A MIC on the signed contents (the MIME header and encoded EDI object, as per RFC 1767) in the message received is calculated using the same one-way hash function that the sending trading partner used.

c. The MIC extracted from the message that was sent and the MIC calculated using the same one-way hash function that the sending trading partner used are compared for equality.

4. The receiving trading partner formats the MDN and sets the calculated MIC into the "Received-content-MIC" extension field.

5. The receiving trading partner creates a multipart/signed MIME message.

6. The MDN is the first part of the multipart/signed message, and the digital signature is created over this MDN, including its MIME headers.

7. The second part of the multipart/signed message contains the digital signature. The "protocol" option specified in the second part of the multipart/signed is as follows:

S/MIME: protocol = "application/pkcs-7-signature"

8. The signature information is formatted according to S/MIME specifications.

The EC Interchange and the MIME EDI content header can actually be part of a multi-part MIME content-type. When the EDI Interchange is part of a multi-part MIME content-type, the MIC MUST be calculated across the entire multi-part content, including the MIME headers.

2.6.4 Usage of Signed MDN for the Sender of the EDI Interchange

The signed MDN, when received by the sender of the EDI Interchange, can be used by the sender as follows: [4]

	Usage of Signed MDN for the sender
1	As an acknowledgement that the EDI Interchange sent was delivered and acknowledged by the receiving trading partner. The receiver does this by returning the original-message-id of the sent message in the MDN portion of the signed receipt.
2	As an acknowledgement that the integrity of the EDI Interchange was verified by the receiving trading partner. The receiver does this by returning the calculated MIC of the received EC Interchange (and 1767 MIME headers) in the "Received-content-MIC" field of the signed MDN.
3	As an acknowledgement that the receiving trading partner has authenticated the sender of the EDI Interchange.
4	As a non-repudiation of receipt when the signed MDN is successfully verified by the sender with the receiving trading partner's public key and the returned MIC value inside the MDN is the same as the digest of the original message.

2.6.5 Synchronous and Asynchronous MDN

The AS2-MDN exists in two varieties: synchronous and asynchronous.

The synchronous AS2-MDN is sent as an HTTP response to an HTTP POST or as an HTTPS response to an HTTPS POST. This form of AS2-MDN is called synchronous because the AS2-MDN is returned to the originator of the POST on the same TCP/IP connection.

The asynchronous AS2-MDN is sent on a separate HTTP, HTTPS, or SMTP TCP/IP connection. Logically, the asynchronous AS2-MDN is a response to an AS2 message. However, at the transfer-protocol layer, assuming that no HTTP pipelining is utilized, the asynchronous AS2-MDN is delivered on a unique TCP/IP connection, distinct from that used to deliver the original AS2 message. When handling an asynchronous request, the HTTP response MUST be sent back before the MDN is processed and sent on the separate connection.

When an asynchronous AS2-MDN is requested by the sender of an AS2 message, the synchronous HTTP or HTTPS response returned to the sender prior to terminating the connection MUST be a transfer-layer response indicating the success or failure of the data transfer. The format of such a synchronous response MAY be the same as that response returned when no AS2-MDN is requested.

The following diagram illustrates the synchronous versus asynchronous varieties of AS2-MDN delivery using HTTP:

Synchronous AS2-MDN

The sender sets up an HTTP connection to the receiver and sends over the AS2 message as the HTTP request. After the receiver received the message it immediately sends back an MDN in the same HTTP connection as the HTTP response. There is only one HTTP connection. The figure below shows the Synchronous AS2-MDN exchanging process.

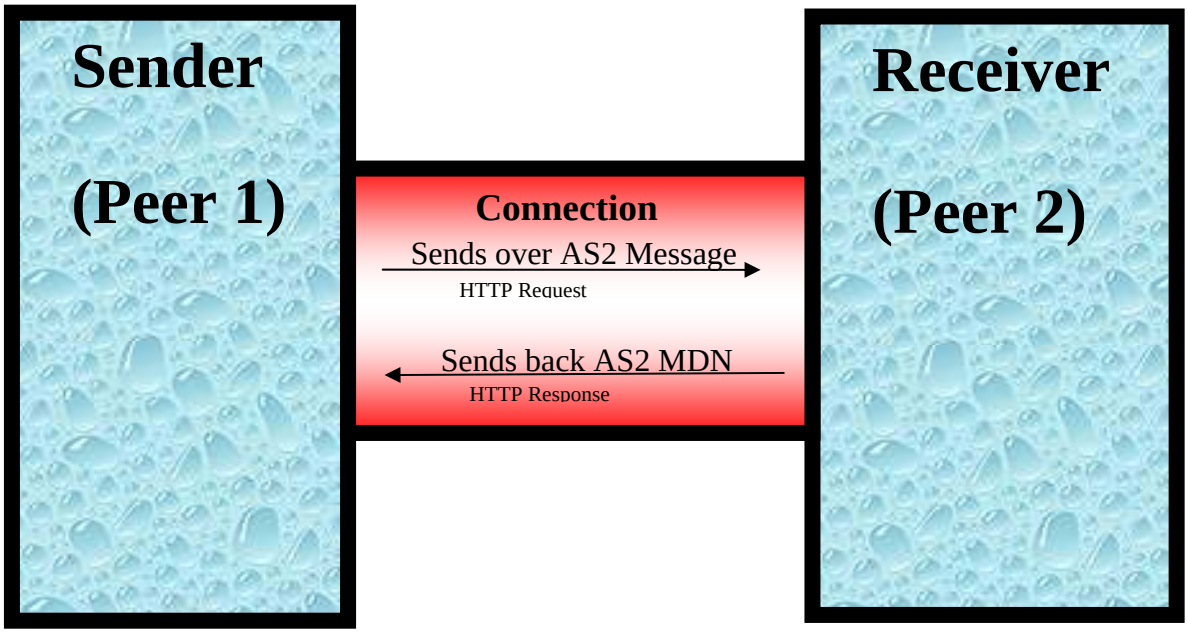


Figure - Synchronous MDN

Asynchronous AS2-MDN

The sender sets up an HTTP connection and sends over the AS2 message as the HTTP request. After the receiver received the message it however does not sends back the MDN immediately. Instead, the receiver waits for a while and sets up another HTTP connection to the sender and sends back the MDN as the new HTTP request. There are two HTTP connections.

The figure below shows the Asynchronous AS2-MDN exchanging process.

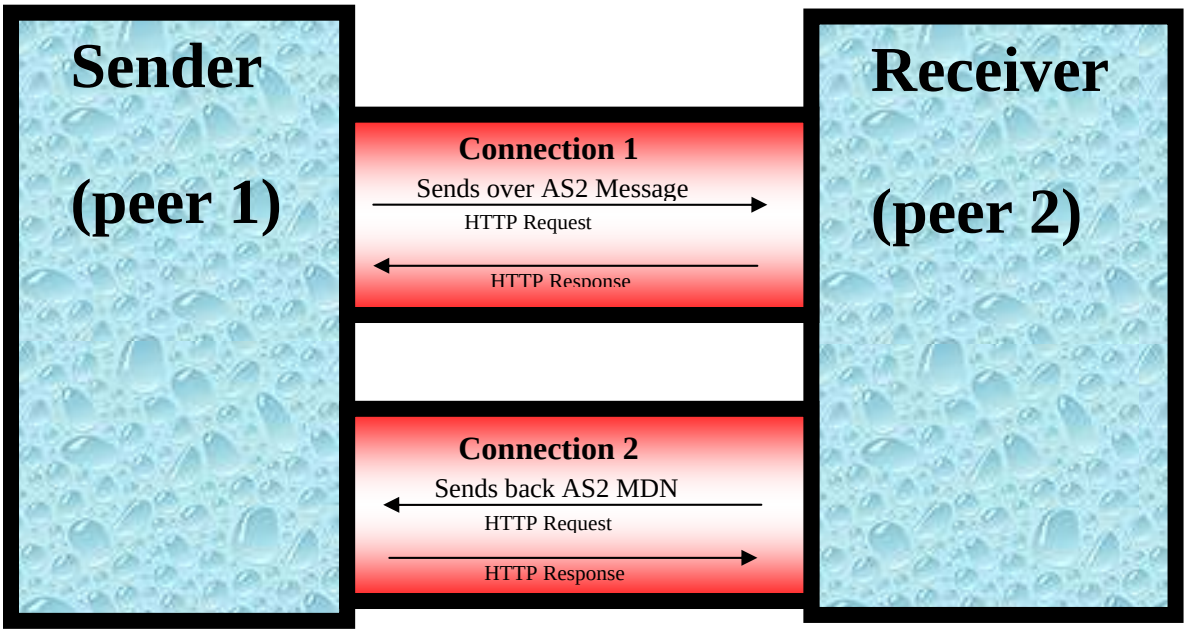


Figure Asynchronous MDN

* Note: An AS2-MDN may be directed to a host different from that of the sender of the AS2 message. It may utilize a transfer protocol different from that used to send the original AS2 message.

The advantage of the synchronous MDN is that it can provide the sender of the AS2 Message with a verifiable confirmation of message delivery within a synchronous logic flow. However, if the message is relatively large, the time required to process this message and to return an AS2-MDN to the sender on the same TCP/IP connection may exceed the maximum configured time permitted for an IP connection.

The advantage of the asynchronous MDN is that it provides for the rapid return of a transfer-layer response from the receiver, confirming the receipt of data, therefore not requiring that a TCP/IP connection necessarily remain open for very long. However, this design requires that the asynchronous AS2-MDN contain enough information to identify the original message uniquely so that, when received by the AS2 Message originator, the status of the original AS2 Message can be properly updated based on the contents of the AS2-MDN. [4]

3. The AS2 Project

3.1 Project management

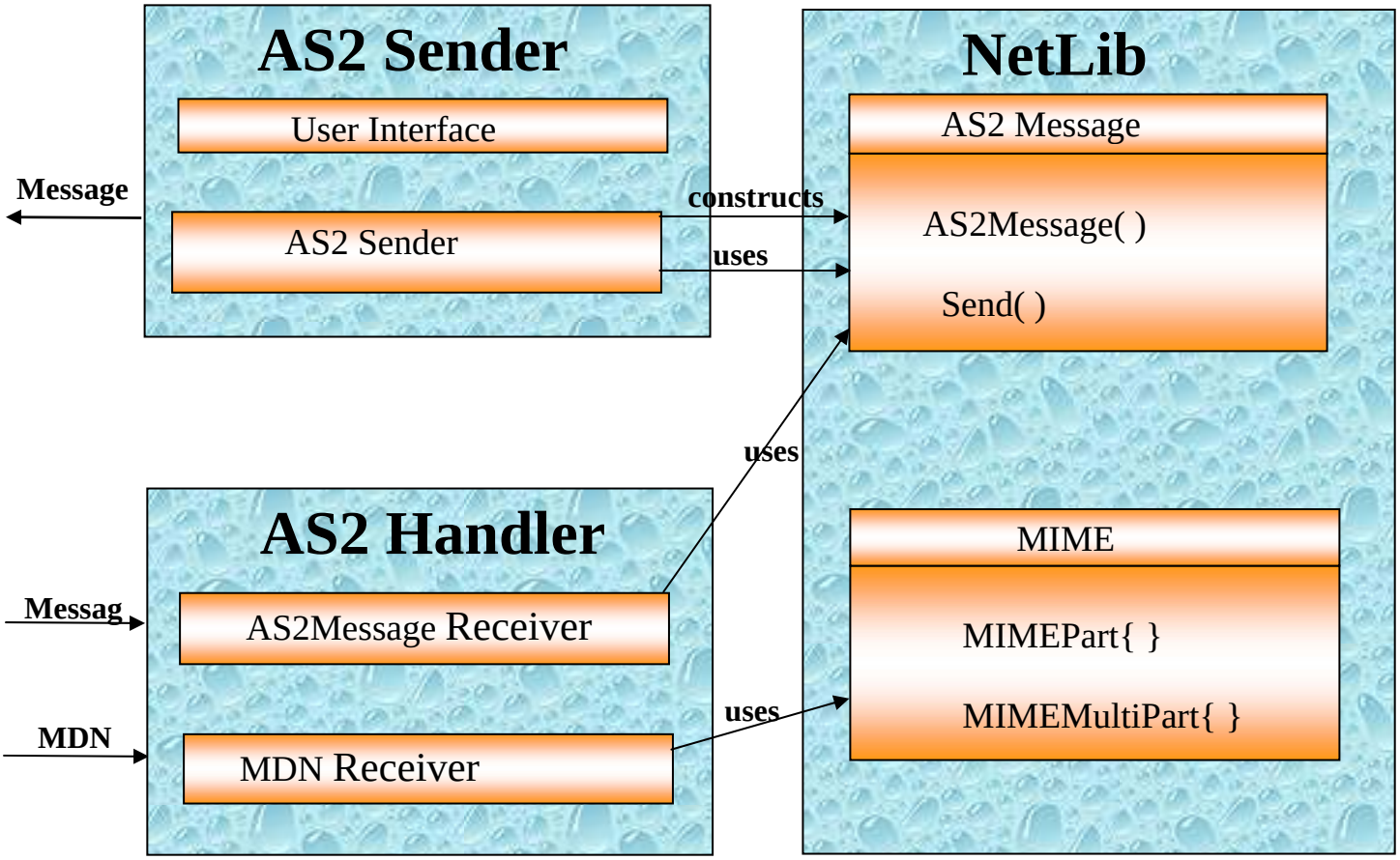
At the Kick-Off meeting, I discussed the project outline with my industry mentor Mr Barry Dowdeswell and my academic supervisors Dr Gerald Weber, Dr S Manoharan and Mr Christof Lutteroth. We set up the strategy and steps for the project, including the deliverable for the first half of the project.

Throughout the course of this project, I worked closely with Mr Lutteroth. We met weekly to discuss my progress and plan the next steps. These weekly meetings were an essential part of the project. They enabled me to receive feedback and also ensured that the program met AARN's expectations. It was at several of these meetings that we reviewed the existing program and made some important decisions regarding the design and implementation of the AS2 program, such as redesigning the architecture.

The project made use of the iterative development methodology. For the first iteration, we developed an AS2 program with very basic functions, such as sending and receiving messages. Then we refined the system architecture and added security features. Finally, the database layer was implemented.

We also used the prototyping methodology to implement AS2. Before implementing the security features, we developed a prototype which was a very basic windows form program that could encrypt and sign messages. We then migrated the code from the prototype to the AS2 program.

3.2 System Architecture



System Architecture

As shown in the diagram, the system contains 3 main parts: the AS2 Sender, which sends out AS2 messages; the AS2 Handler, which receives and processes incoming AS2 messages and MDN; and a library called NetLib.

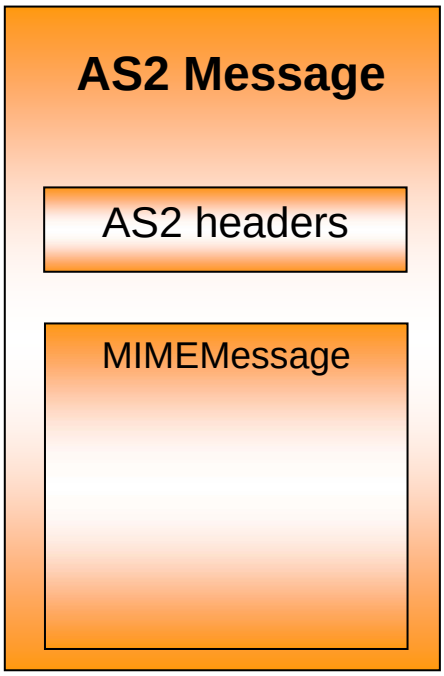
3.2.1 NetLib

This is the library of this program. Most of the business logic is implemented here. It mainly contains 2 classes: AS2Message and MIMEMessage.

AS2Message class formats the message that is going to be sent, add proper headers to the message and creates a MIMEMessage as body. AS2Message class has many instance variables that holds the information of the message such as Signed, Encrypted as well as the information of the sender and receiver such as SendFrom, SendTo, etc. There are 2 different constructors for different use. The most important method of this class is the Send() method, which sends over the formatted message via an HTTP Post method. This method returns the corresponding HTTP response. There will be some detailed introduction to this class in next section.

MIMEMessage class formats the message data of AS2.

The figure below shows the relationship between AS2Message and MIMEMessage:



The AS2Message has mainly two parts:

- AS2 headers – Name Value collection
- Message data – a MIMEMessage. The MIMEMessage object captures the actual message data, wrap it with some MIME headers to form the MIMEMessage object. Then AS2Message takes the MIMEMessage and wrap it up with AS2 headers.

We have 2 layers here. The MIMEMessage layer takes care of the security features of AS2 while the AS2Message layer is responsible for the connection and transfer.

3.2.2 AS2 Sender

The sender has 2 layers: The user interface and the business logic that responds to the user interface and sends out messages.

The user interface is not fully designed yet at this stage but there is a very simple interface for testing purpose. It looks like this:

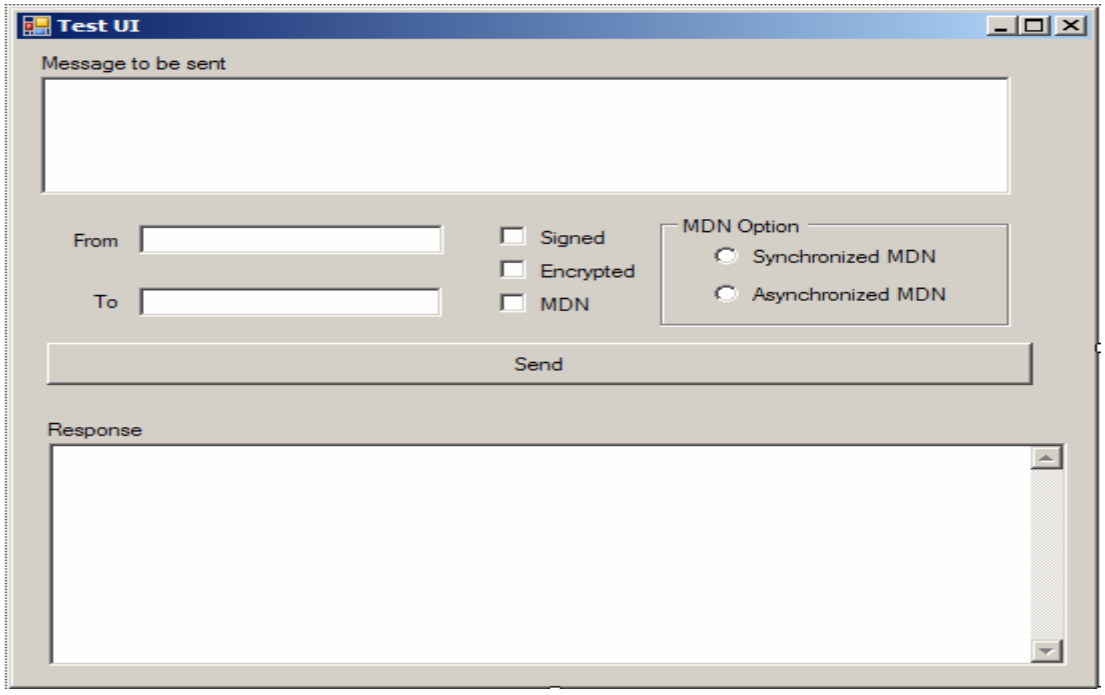


Figure - test UI

The idea of the test program is to send different types of AS2 messages according to the permutation table. The corresponding response is sent back and shown on the user interface.

Controls:

Control Name	Control Type	Usage
txtMessageTo	Text box	The user types in message data
txtResponse	Text box	The HTTP response received
txtFrom	Text box	Specifies the identity of the partner that sends the message
txtTo	Text box	Specifies the identity of the partner that the message is sent to
ckbSigned	Check box	Specifies whether the message is signed or not
ckbEncrypted	Check box	Specifies whether the message is encrypted or not
ckbMDN	Check box	Specifies whether a MDN is requested or not
grpMDN	A group of radio buttons	Specifies whether a synchronous or asynchronous MDN is requested
btnSend	Button	Send the message

Process

The figure Test UI shows how the interface looks like initially. When sending a message, the user types in the message data into the text box: txtMessageTo, check the checkboxes indicating whether the message is signed, encrypted, request MDN, etc. When the send button is clicked, the message is constructed and sent to the server over an HTTP connection. The HTTP response is then shown in the text box: txtResponse.

There are 3 steps to send an AS2 message:
Construct an AS2Message object
An AS2Message object is constructed using the constructor AS2Message( ).

Set the value of the instance variables of the constructed AS2Message object.
The instance variables of the AS2Message object are set according to the controls. For example, if the check box: ckbMDN is checked then the instance variable: Boolean MDN is set to be true.

Send the message.
The instance method Send( ) is then called to send the AS2 message using an HTTP Post method. The program will check if the AS2 message is to be signed and/or encrypted then sign and/or encrypt the message before sending it.

3.2.3 AS2 Handler

A .ashx file, which sits in the receiver's IIS server, is used as the handler to process received messages.

When a message is received

1. Construct an AS2Message object
   When a message is received through HTTP request, an AS2Message object is constructed by using the constructor AS2Message(HttpRequest request).

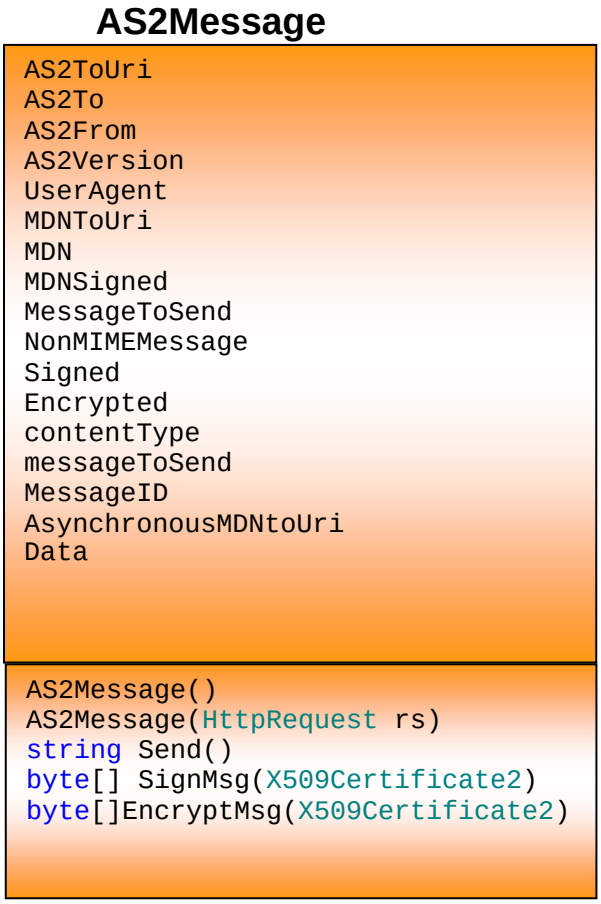
2. Extract information from HTTP request.
   This is actually a part of constructing an AS2Message object using the HttpRequest received. For example: `AS2From = rs.Headers.Get("AS2-From");`
   AS2From is an instance variable of the AS2Message object. It is set by the value of the "AS2-From" header in the HttpRequest. After the AS2 object is constructed, the information then can be accessed through instance variables on the receiver side.

3. Check if the AS2 message is encrypted and/or signed. If so, verify the signature and decrypt the message.

4. Save data into DB
   The received message data and some information are then saved into the database

5. Construct and send MDN
   If a MDN is requested, the handler will construct the MDN according to the requested MDN type and then send back to the sender.

3.2.4 Core Classes



This class is the core of the system. Both Sender and Receiver need to construct an AS2Message object for each data exchange.

The instance variables hold necessary information of this AS2 message or the data exchanging process.

There are 2 constructors:

- The one takes no parameter is used by the Sender. The client has to set instance variables of the AS2Message object.

- The other one takes an HttpRequest type parameter. This constructor is used by the Receiver. It extracts information from the HttpRequest, set instance variables.

The instance method Send( ) basically sends the constructed AS2Message over and HTTP connection using a Post method. It also returns the HttpResponse of the corresponding HttpRequest.

Figure - AS2Message class diagram

Instance Variables

Type	Name	Default Value	Usage
Uri	AS2ToUri		The uri of the receiver
String	AS2To	"Sender"	Sender's identity
String	AS2From	"Receiver"	Receiver's identity
String	AS2Version	"1.1"	AS2 version number
String	UserAgent	"EDISAS2 Client"	User Agent name
String	MDNToUri		The uri for the MDN to be sent to
MDNType	MDN	MDNType.None	enum MDNType{ None, Synchronous, Asynchronous }

Boolean	MDNSigned	False	Whether the MDN is signed
MIMEMessage	MessageToSend	Null	The MIME Message
String	NonMIMEMessage	“Test Message”	The non-MIME Message
Boolean	Signed	False	Whether the message is signed
Boolean	Encrypted	False	Whether the message is encrypted
String	contentType	“”	Http Content type header
String	MessageID	“id”	The message’s identity
Uri	AsynchronousMDNtoUri		The uri for the asynchronous MDN to be sent to
String	Data	“Empty”	The chunk of message data

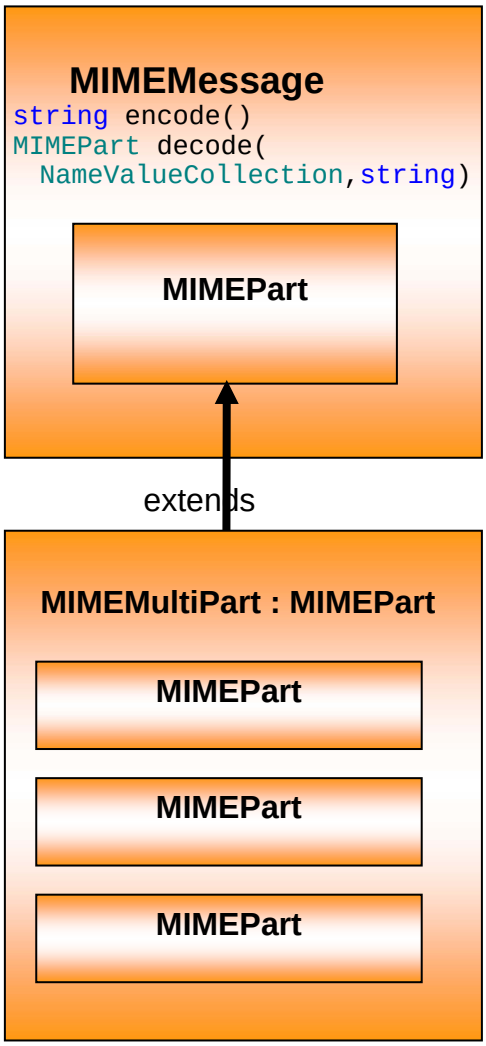
Constructors

Constructor	Parameter	Used by	Note
AS2Message()	None	Sender	The sender needs to set instance variables specifically afterwards
AS2Message(HttpRequest)	HttpRequest	Receiver	The instance variables are set automatically by the constructor.

Instance methods

Instance methods	Note
string Send()	The core method of the class. It sets the values of the AS2 headers; checks if the message needs to be signed and/or encrypted; sign/encrypt the message if needed; creates the message body in MIME format then send the message using an HTTP Post method and returns the response from the receiver as a string.
byte[] SignMsg(X509Certificate2)	This method uses the sender’s X509Certificate2 certificate (private key) to create a digital signature for the message data to be sent. It returns the digital signature in an array of bytes.
byte[] EncryptMsg(X509Certificate2)	This method uses the receiver’s certificate (public key) to encrypt the message. It returns the encrypted data in an array of bytes.

MIMEMessage, MIMEPart and MIMEMultiPart



The diagram to the left shows the structure of the MIMEMessage class as well as the relationship among MIMEMessage, MIMEPart and MIMEMultipart:

- The MIMEMessage contains a MIMEPart, i.e: there is an instance variable MIMEPart for a MIMEMessage object.
- MIMEMultiPart extends MIMEPart.
- MIMEMultiPart contains an array of MIMEPart.

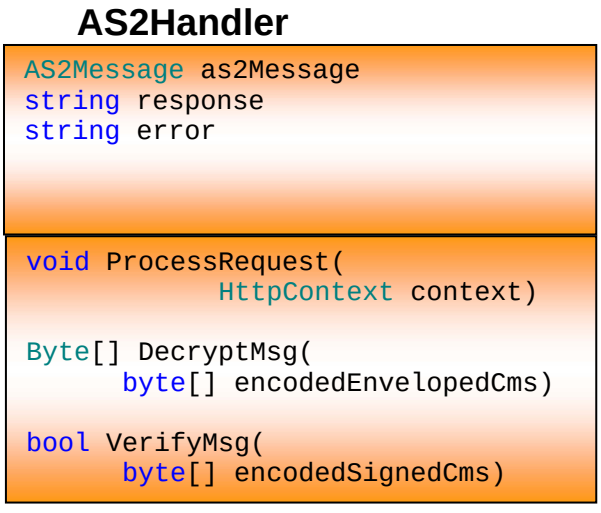
For a normal AS2 message, the message data is stored in a MIMEPart. When the AS2 message is signed, it has a digital signature and in this case we need to use MIMEMultiPart. Inside of the MIMEMultiPart, there is one MIMEPart holding the message data and another MIMEPart holding the digital signature.

There are two instance methods in MIMEMessage class:

1. string encode()
   The sender encodes the MIMEMessage into a string ready for sending.

2. MIMEMessage decode(NameValueCollection,string)
   The receiver uses this method to decode the received string into a MIMEMessage object.

AS2Handler class:



The AS2Handler sits on the Receiver's IIS server. It receives AS2 messages, processes them and sends back response.

When a message is received, AS2Handler creates an AS2Message object, checks if the message is signed and/or encrypted. If so, use the corresponding method DecryptMsg( ) or VerifyMsg( ) to process it. After the message is verified and decrypted, AS2Handler saves the received message into the database and send back response to the sender.

3.3 Implementation of Security features

In order to implement the digital signature and encryption, we mainly made use of the System.Security.Cryptography.Pkcs namespace. This name space provides programming elements for Public Key Cryptography Standards (PKCS), including methods for signing data, exchanging keys, requesting certificates, public key encryption and decryption, and other security functions.

Digital signature, the main functions are

1. Sign the message
   First, place message in a contentInfo object. This is required to build a SignedCms object, which is going to be used to store the signature.

```
ContentInfo contentInfo = new ContentInfo(msg);
```

   Instantiate SignedCms object with the ContentInfo above.

```
SignedCms signedCms = new SignedCms(contentInfo);
```

   Then Formulate a CmsSigner object for the signer.

```
CmsSigner cmsSigner = new CmsSigner(signerCert);
```

   Now we can sign the CMS/PKCS #7 message using the X509Certificate2 object cmsSigner

```
signedCms.ComputeSignature(cmsSigner);
```

   Return an array of bytes – the signature

```
return signedCms.Encode();
```

2. Verify Signature
   Verifiy the encoded SignedCms message and return a Boolean value that specifies whether the verification was successful.

   We need a SignedCms object to decode and verify the byte array (that was returned from the Sign Message method above)

```
SignedCms signedCms = new SignedCms();
signedCms.Decode(encodedSignedCms);
```

   With the SignedCms object we now can verify the signature. This statement throws System.Security.Cryptography.CryptographicException when the signature is not verified. So a try-catch block is used here and a false is returned if the signature failed the verification.

```
try{
        signedCms.CheckSignature(true);
}catch(System.Security.Cryptography.CryptographicException){… return false;}
```

For Encryption, the main functions are

1. Encrypt the message
   Encrypting message has a similar procedure as signing a message. The major difference is instead of using Signedcms, we use EnvelopedCms here.

```
EnvelopedCms envelopedCms = new EnvelopedCms(contentInfo);
```

   Then we need to formulate a CmsRecipient object that represents information about the recipient to encrypt the message for. Here we need the recipient's Public Key the encrypt the message. The Public Key is stored in the X509Certificate2 object: recipientCert.

```
CmsRecipient recip = new CmsRecipient(
             SubjectIdentifierType.IssuerAndSerialNumber,
             recipientCert);
```

   Now with the CmsRecipient created above we can encrypt the message using the recipient's Public Key:

```
envelopedCms.Encrypt(recip);
```

   Return an array of bytes. The encoded EnvelopedCms message contains the encrypted message and the information about the recipient that the message was encrypted for.

```
return envelopedCms.Encode();
```

2. Decrypt the message
   Similar as verifying signatures, we can simply call a method to decrypt the encrypted data. The recipient's information is stored in the envelopedCms object already and this statement will automatically locate the corresponding Private Key to perform the decryption.

```
envelopedCms.Decrypt(envelopedCms.RecipientInfos[0]);
```

4. Problems & Solutions

4.1 The decision of redesigning of the original program

There is an existing version of the AS2 system. The project originally was to understand the existing system, refine it and add security features on top of it. However a few critical problems were identified, which lead to the decision of redesigning the system.

One problem of the original AS2 system is that the architecture is not well designed.

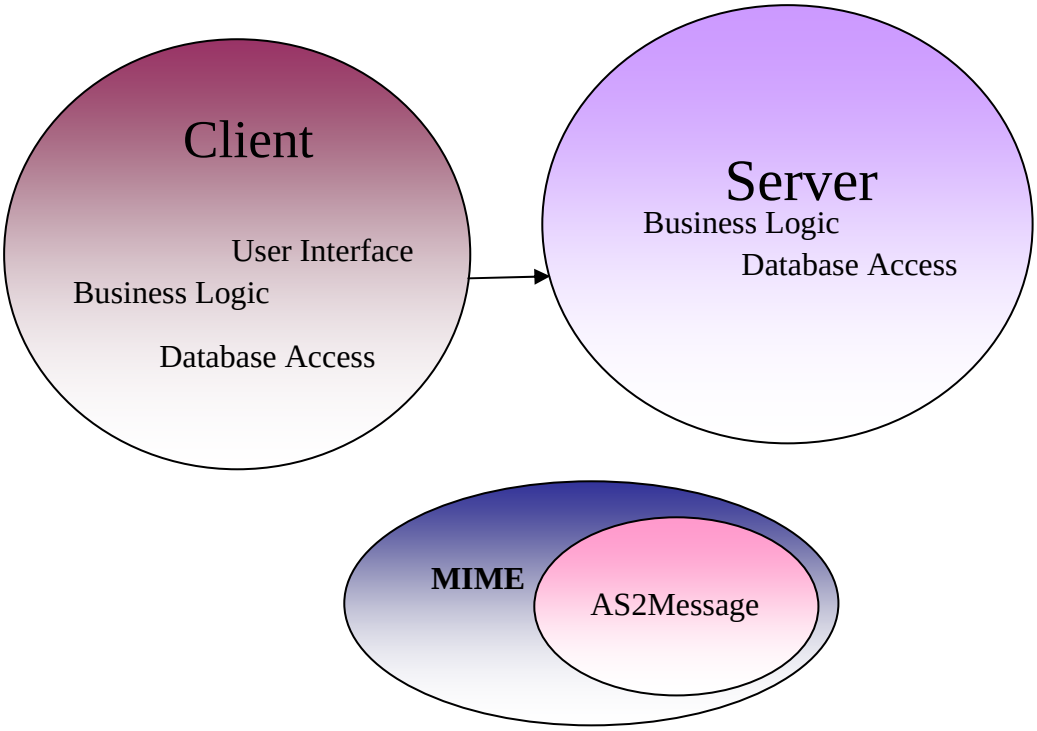


Figure - original AS2 implementation architecture

As shown above, different layers are mixed together. For example, on client side, the User Interface code, Business logic code and Database Access code are all mixed together.

Another problem is the connections between components are not clear. For example, on the server side, the logic of processing received messages suggests that AS2Message contains MIMEMessage, however, AS2Message class is inside of the MIME folder. Also, in AS2Message class the MIMEMessage type instance variable is declared but was never used, which means there is no actual connection between MIMEMessage and AS2Message class. Further more, AS2Message variables were never used in the program although many were declared.

In addition, most variables are declared as global variables and the program would not compile, in fact, the compiler picked up over 100 errors.

Because of these, the decision of redesigning the system was made. The new system architecture is described in section 4.2.

4.2 Sending different types of messages

Due to the permutation of the variables: Signed, MDN, Encrypted, MDNSigned, Synchronous MDN, there are many different types of the AS2 message. See Assumptions—Permutation summary section.

In order to send different types of messages, the AS2Message class was designed to be:

AS2Message

Instance Variables

```
AS2Message( HttpRequest rs)

static Send(String AS2To, String AS2From, string message,
string uri, bool encryptedMessage, bool signedMessage)

static SendWithSynchMDN(String AS2To, String AS2From,
String message, String receiverUrl, bool encryptedMessage,
bool signedMessage, bool encryptedMDN, bool signedMDN)

static SendWithAsynchMDN(String AS2To, String AS2From,
string message, string receiverUri, string receiptUri, bool
encryptedMessage, bool signedMessage, bool encryptedMDN,
bool signedMDN)
```

Figure – old send methods

As shown above, there are more than one send methods and all of them are static. According to this design, the sender does not need to construct any AS2Message object, instead the Sender can directly call the proper static send method to send the message.

However this design has a few disadvantages:

- The sender has to decide which send method to use and the corresponding message types of each send method are not defined clearly. This increases complexity of the system.
- In order to call the send method, a lot of parameters are needed. This increases complexity and/or inconvenience too.
- The sender does not actually need to use any of the instance variables. Because the sender passes parameters to the send method. However, the parameters and the instance variables hold the same data. This means duplication of information.

Because of the considerations above, the class was redesigned to *Figure - AS2Message class diagram.* In the new design, the instance variables are fully made use of and there is only one send method which takes no parameter. The new design resolved the disadvantages stated above.

4.3 Detach Signature from Message Data and Verify the Separated Signature

When I was implementing Digital signature, one problem I encountered is that the original message data is embedded somewhere inside of the Signature. This is not what we want because we want separated Signature and the message data so that we can put them in separate MIMEPart. After some research I found out the key to this problem is in the constructor of SignedCms:

```
public SignedCms ( ContentInfo contentInfo, bool detached )
```

If the detached state is false (the default), the content that is signed is included in the CMS/PKCS #7 message along with the signature information. If the detached state is true, clients that cannot decode S/MIME messages can still see the content of the message if it is sent separately. This might be useful in an archiving application that archives message content whether the message sender can be verified for authenticity.

So, if use:

```
SignedCms sm = new SignedCms(contentinfo, true);
```

Then the original message will be automatically detached from the signature.

For example, when I try to sign this string:

"Germany is a country in central Europe and a member of the European Union. Official Name:: Bundesrepublik Deutschland"

Get the signature:

¬– MIICXAYJKoZIhvcNAQcCoIICTTCCAkkCAQExCzAJBgUrDgMCGgUAMAsGCSqGSIb
3DQEHAaCCAZIwggGOMIIBOKADAgECAhEA5NA/Du9fokKfi6zibqtLMjANBgkqhkiG9w0
BAQUFADAaMQswCQYDVQQGEwJOWjELMAkGA1UEAxMCZGUwHhcNMDYwODEwM
DAwMDAwWhcNMDcwODA5MjM1OTU5WjAaMQswCQYDVQQGEwJOWjELMAkGA1UE
AxMCZGUwXDANBgkqhkiG9w0BAQEFAANLADBIAkEA5pvNr3CaD9wnJasQ8wxApzMR
k0dqXr0Yl006TTdlMS1uE1FWFAiOLekThMDR36qLIIZh3+tEGFcajD86dFl13wIDAQABo1
kwVzAiBgNVHSMBAQAEGDAWgBRy9EiyonBGYxG1P8/g04299EEtUzAgBgNVHQ4BAQ
AEFgQUcvRIsqJwRmMRtT/P4NONvfRBLVMwDwYDVR0PAQH/BAUDAwCAADANBgkq
hkiG9w0BAQUFAANBAKzvFoiN404mexOQ1ZsrEcCKbaty/L1yNQxg/cSpgpCuskZzGccw
vgem6Kfo1F2e5uKS0U1od9aFWDw42vt8L/cxgZMwgZACAQEwLzAaMQswCQYDVQQG
EwJOWjELMAkGA1UEAxMCZGUCEQDk0D8O71+iQp+LrOJuq0syMAkGBSsOAwIaBQA
wDQYJKoZIhvcNAQEBBQAEQKy8T2Pog2iTC1g44w/hKX5170+YzQcSzJdgHbj4d1/YPG
NaSArfhX5H4m0jgvmXNZvbBUMueMUgKOSLzKkvoJ0=

Problem solved. However, the signature then cannot be verified after the original message was detached.

From MSDN I found the following segment of code:

```
// Create a ContentInfo object from the inner content obtained // independently
from encodedMessage.
ContentInfo contentInfo = new ContentInfo( innerContent );
// Create a new, detached SignedCms message.
SignedCms signedCms = new SignedCms(contentInfo, true);
// encodedMessage is the encoded message received from // the sender.
signedCms.Decode(encodedMessage);
// Verify the signature without validating the // certificate.
signedCms.CheckSignature(true);
```

That was suggested in MSDN. But the problem is the parameter 'innerContent' that was used to create ContentInfo. The only clue it gave is "the inner content obtained independently from encodedMessage.".

There is only one constructor for ContentInfo that accepts one parameter and the parameter is a byte array. So both the encodedMessage and the innerContent are an array of bytes. I tried to use encodedMessage as the innerContent but did not work. So the question is: Which part of the byte array 'encodedMessage' should be used as the 'innerContent' byte array?

MSDN did not mention at all nor gave any link.

I thought of a solution:
Use 2 SignedCms objects. One has the original message attached with the signature and this is used to verify the signature. The other one has the original message detached and this is used to extract the signature for later use (create multi-MIME).

I modified the SignMsg() method. Added a Boolean variable 'detach' to indicate if we want the returned byte array have original message in it nor not.

```
static public byte[] SignMsg(
          bool detach,
          Byte[] msg,
          X509Certificate2 signerCert)
```

Call the new method twice with detach to be true and false. Get 2 byte arrays: encodedSignedCmsWithMsg and encodedSignedCmsWithoutMsg. encodedSignedCmsWithMsg is used to verify the signature.

encodedSignedCmsWithoutMsg is used to wirte to a file called signature.txt. We can use this to compose multi-MIME.

However this solution is not efficient and confusing. We found another much more robust solution in the end:

When we create the detached signature, we give the plain text message in variable "message":

```
ContentInfo contentInfo = new ContentInfo(message);
SignedCms signedCms = new SignedCms(contentInfo, true);
byte[] signature = signedCms.Encode();
```

Now we can create a MIME multipart: part 1 – the message data , part 2 – the signature
The receiver receives the MIME multipart and thus has the message and the signature.
Now, verify the signature

```
ContentInfo contentInfo = new ContentInfo(message);
SignedCms signedCms = new SignedCms(contentInfo, true);
signedCms.Decode(signature);
signedCms.CheckSignature(true);
```

The idea is that it works symmetrically to the creation of the signature: the sender used the message as contentInfo, so now the receiver does that, too. The sender created a SignedCms object with setting "detached signature", so the receiver does that, too.
Now, the receiver already has the signature which was created using the Encode method. Well, now we use the Decode method on the same data, i.e. the signature. The SignedCms has now a link to the original message (in the ContentInfo object) and the signature (decoded with Decode).

So now CheckSignature( ) works perfectly with the detached signature.

4.4 The problem with IIS Server

The biggest problem we encountered in this project is that security features did not function on IIS server.

For the security features we developed a test program to demonstrate how the digital signature and encryption can be implemented. This test program does signing, verification, encryption and decryption all together and everything worked well. We then migrated code for this part into the AS2 program. It signs and encrypts the AS2 message perfectly but the AS2Handler cannot verify nor decrypt the received message. In fact, it crashes whenever the verify message or decrypt message codes are executed.

The exception was thrown:

```
System.Security.Cryptography.CryptographicException: ASN1 bad tag value met.
```

The AS2Handler has the same code for the verifying signature and decryption part as we have in the sample program but it just does not work on the server side. After some google search we found out lots of complains about this. Seems it has something to do with this version of IIS server. IIS5.3 is not a completed version and the changes from

IIS5.3 to IIS6.0 are evolutional according to Barry. But the problem is IIS 6.0 is not a freeware anymore.

Before giving up, we had a last test about IIS and the codes for security features. The normal procedure is the sender signs and encrypts the message then sends it to the receiver. The AS2Handler on the receiver's IIS server receives the signed and encrypted message then it tries to verify the signature and decrypt the message. However in this test we made the AS2Handler do everything: sign, verify, encrypt, decrypt. This means all the codes are executed at server side. This is basically putting the test program we developed before on the IIS server and see if it still works. It failed however. The AS2Handler couldn't locate the correct Certificate and couldn't verify the signature nor decrypt the encrypted message.

We then conducted some more research and finally with Matthew's help in AARN we found the cause: IIS did not have the permission to access the certificate store that held the public and private keys. Certificate stores are normally under different user accounts and programs that do not belong to the same user account don't have access to the certificate store. However, IIS is not running under any particular user account, which means IIS cannot access any certificate store that belongs to any particular user account. In order to solve this problem, we have to use the root certificate store, which belongs to the local machine but not any particular user account. Also we have to set the assembly permission for the handler so that it can access the certificate store.

We stuck at this problem for a few weeks and did not find the cause until the very last minute. Even so we still have not figured out how to set the assembly permission for the handler correctly. Some more research still needs to be done on this part.

5. Conclusion

AS2 is a protocol for the secure exchange of structured business data using HTTP transfer protocol, where:

- The structured business data may be in XML or other structured formats;
- The data is packaged using standard MIME structures;
- Authentication and data confidentiality are obtained by using Cryptographic Message Syntax with S/MIME security body parts; and
- Authenticated acknowledgements make use of multipart/signed MDN responses.

The project achieved the following:

- The system architecture of the pre-existing program was redesigned to make it clearer and more effective;
- The basic operations of AS2 data exchange were implemented;
- AS2 security features such as digital signature and encryption were implemented; and
- The redesigned program was integrated into the message system of EDIS for Windows.

These results fulfil the project plan.

6. Next Step

Up to now, most AS2 functions/features are implemented except MDN. Currently we only implemented synchronous MDN without proper structure. So the next step is to construct proper MDN (both synchronous and asynchronous), send and handle it. This requires a new MDN class that is similar to AS2Message class in NetLib. We also need a MDN handler to receive and handle the MDN from the receiver.

7. Acknowledgement

Firstly I have to thank my supervisor Mr Lutteroth, who guided me through this project and was always there to help and support.

Also thanks to my supervisor Dr. Gerald and Dr. Manoharan and industry mentor Barry for their support and information provided.

Special thanks to AARN for their remote help and technical support.

8. Bibliography

[1] Barry Dowdeswell and Christof Lutteroth. A Message Exchange Architecture for Modern E-Commerce. In: Trends in Enterprise Application Architecture, LNCS 3888, Springer, March 2006.

[2] E.D.I. University online. Introduction to EDI http://www.ediuniversity.com/intermediate/as2.jsp

[3] Barry Dowdeswell. AS2 Overview. Feb, 2006

[4] D. Moberg, Cyclone Commerce and R. Drummond, Drummond Group Inc; "MIME-Based Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2)", RFC 4130, July, 2005

[5] David Youd http://www.youdzone.com/signature.html an introduction to Digital Signatures

9. Appendix -- terms

AS2: Applicability Statement 2

EDI: Electronic Data Interchange

EC: Business-to-Business Electronic Commerce

B2B: Business to Business

Receipt: The functional message that is sent from a receiver to a sender to acknowledge receipt of an EDI/EC interchange. This message may be either synchronous or asynchronous in nature.

Signed Receipt: A receipt with a digital signature.

Synchronous Receipt: A receipt returned to the sender during the same HTTP session as the sender's original message.

Asynchronous Receipt: A receipt returned to the sender on a different communication session than the sender's original message session.

Message Disposition Notification (MDN): The Internet messaging format used to convey a receipt. This term is used interchangeably with receipt. A MDN is a receipt.

Non-repudiation of receipt (NRR): A "legal event" that occurs when the original sender of an signed EDI/EC interchange has verified the signed receipt coming back from the receiver. The receipt contains data identifying the original message for which it is a receipt, including the message-ID and a cryptographic hash (MIC). The original sender must retain suitable records providing evidence concerning the message content, its message-ID, and its hash value. The original sender verifies that the retained hash value is the same as the digest of the original message, as reported in the signed receipt. NRR is not considered a technical message, but instead is thought of as an outcome of possessing relevant evidence.

S/MIME: Security MIME. A format and protocol for adding cryptographic signature and/or encryption services to Internet MIME messages.

Cryptographic Message Syntax (CMS): An encapsulation syntax used to digitally sign, digest, authenticate, or encrypt arbitrary messages.

SHA-1:A secure, one-way hash algorithm used in conjunction with digital signature. This is the recommended algorithm for AS2.

MD5: A secure, one-way hash algorithm used in conjunction with digital signature. This algorithm is allowed in AS2.

MIC: The message integrity check (MIC), also called the message digest, is the digest output of the hash algorithm used by the digital signature. The digital signature is computed over the MIC.

User Agent (UA): The application that handles and processes the AS2 request.