A Comparison of XML Processing in .NET and J2EE

Biswadeep **Nag** <biswadeep.nag@sun.com>

Abstract

In the modern world of web service enabled enterprise applications,the ability to parse and process XML documents efficiently is of fundamental importance. The two premier development platforms of today, .NET and J2EE offer extensive capabilities for processing XML. In this paper, we make certain technical comparisons about the different XML processing techniques that are offered in these two frameworks.

We discuss the basics of the different parsing alternatives available in the two platforms such as SAX, DOM, pull parsing, as well as the XML binding technologies JAXB and XmlSerializer. We then utilize a suite of open-source XML processing programs that explores various components of the parsing infrastructure and offer examples of which technique might be most suitable during the lifecycle of an XML document. At the end of it all, the reader will have a balanced view of how one can take advantage of the best of both worlds. This is only possible because the XML language is universal across both .NET and J2EE.

Table of Contents

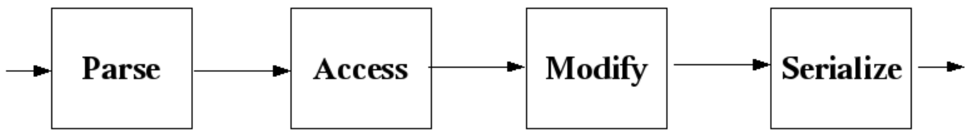
1. Introduction

Processing of XML documents is assuming growing importance in the modern-day IT infrastructure. The most well-known use case is in the implementation of web services that rely on XML as the underlying data exchange format. Not only are the arguments and attachments involved in web service calls transferred as XML messages in the SOAP format, all the associated infrastructure describing the web service starting from the overall description specified in WSDL (Web Services Description Language) down to the message types (XML Schema) are expressed in XML.

What has grabbed less attention, yet has become equally prevalent, is the use of XML in seemingly more mundane tasks such as specifying the content of industrial-strength web sites, describing system configurations and even in creating office documents. Now that StarOffice, Microsoft Office and the Lotus suite have decided to store their documents natively in XML format, it is quite possible that the killer apps for XML have finally arrived.

In this paper we would like to draw your attention to the many different options for processing XML documents that are available should you choose either of the two dominant software platforms today: Java 2 Platform, Enterprise Edition (J2EE) and Microsoft .NET. Both Java and .NET offer facilities for parsing, manipulating and creating documents using a variety of streaming, tree-building and binding models. Though the alternatives sometimes seem similar, there are subtle differences in the technology that make one of the alternatives a better choice than the other, depending on the situation. We hope to leave you with a clear understanding of the relative strengths and weaknesses of the two platforms so that you can figure out what will work best for you and when.

2. The Lifecycle of an XML Document



The steps involved in processing a XML document, whether as part of a web-service enabled application or as part of a standalone document authoring program, typically involves several of the following functions:

Parse	This is the process of reading a document from a stream (either a network or a file stream) parsing it either to construct an in-memory tree to to extract tokens and optionally also validate it using a DTD or an XML Schema Descriptor.
Access	The whole point of parsing an XML document is to extract the information contained in the elements, attributes or text content. The access can either be complete, in case all the elements in a document are processed, or it can be selective, in which case, the application is only interested in a subset of the elements.
Modify	Some applications may desire to modify either the content of the text nodes or even to change the structure of a document by inserting or deleting sub-trees of nodes.
Serialize	Finally, when all the desired modifications are done, the XML document has to be serialized into textual form and written out to a file or network stream.

Of course, depending on the need, some applications may choose to skip some of these steps. For example, in the case of XML in web service calls we may need the *serialize* step to marshall program objects to XML, the *parse* step to convert XML elements back to program objects at the other end, and then possibly the *access* step to examine some of the objects. The appropriate weightage for each of these processing steps will also be very application-dependent.

To understand the various tradeoffs in program design as well as in performance, we made use of a suite of XML processing programs that were originally based on the work of Dennis Sosnoski (http://www.sosnoski.com/opensrc/xmlbench/index.html). The input data for these programs was synthetically generated using a hypothetical *invoice* document that was based on a corresponding UBL schema (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ubl). The invoice schema is quite complex and complex and as such is pretty realistic.

Broadly speaking, our test invoice has 3 main sections, a fixed-length *Header*, a variable number of *LineItem* elements, followed by a fixed-length *Summary*.

```
<complexType name="InvoiceType">
   <complexContent>
      <sequence>
         <element name="Header" type="InvoiceHeaderType"/>
         <element name="LineItem" type="InvoiceLineItemType"
                 minOccurs="1" maxOccurs="unbounded"/>
         <element name="Summary" type="InvoiceSummaryType"/>
      </sequence>
   </complexContent>
</complexType>
```

By varying the number of lineitems from 10 to 1000, we generated 5 different documents ranging in size from 6 KB to 900 KB. Though we started with Sosnoski XmlBench, we ended up changing the program code substantially in the following respects:

- The original code was in Java. We also created a C#/.NET version.
- Adapted it to work with JAXB (and MS XML binding).
- It originally counted the number of element tags and attributes in a document. This is impossible to do in binding frameworks like JAXB. Instead we actually retrieve the GrossUnitPriceAmount from each LineItem.
- Added a validation test that validates the document against an XML schema. Most of the parsers tested by Sosnoski do not support validation.
- We added a selective walk test that retrieves only some of the lineitems. This can be used to distinguish between SAX/Pull Parser and DOM/JAXB.
- Instead of the modification test removing whitespace from text nodes, in our test, we increase the price of each item by 10%. This is probably more realistic.
- Added a structure modification test where a certain number of element nodes are deleted and then added.
- Created a multi-threaded version. Sosnoski XmlBench was originally single-threaded.

3. Java and .NET XML Processing

Most of the infrastructure for XML processing in the Java platform comes bundled in the Java Web Services Developer Pack (http://java.sun.com/webservices/downloads/webservicespack.html) that contains implementations of the J2EE API (http://java.sun.com/j2ee/1.4/docs/api/index.html). We used JWSDP 1.2 that included the Java API for XML Processing (JAXP) v1.2.3 and the Java Architecture for XML Binding (JAXB) v1.0.1. The Java Virtual Machine we used came from J2SE 1.4.2 (http://java.sun.com/j2se/1.4.2/download.html) We tried out the Java software on both Sparc/Solaris and Windows/Intel platforms without any noticeable differences in functionality.

Microsoft offers two alternatives for processing XML. One is the MSXML package currently at v4.0 (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/htm/sdk_intro_6g53.asp) and the other is the .NET framework v1.1 (http://msdn.microsoft.com/netframework/). For this work we concentrated exclusively on the System.Xml and other associated namespaces of the .NET framework because that seems to be the preferred direction for the Microsoft developers. The development platform we used was C# and Visual Studio .NET on Windows Server 2003.

For the XML lifecycle functions we defined previously (Section 2, “The Lifecycle of an XML Document”), both Java and .NET offer a number of processing alternatives. First, there are the stream based parsers: SAX in Java and the pull parser implementation in .NET. However neither can be used for document modification or serialization and for that, both Java and .NET offer implementations of DOM. In the case of Java, developers using SAX or DOM are expected to write to the JAXP API so that the application code can be independent of the parser used. The parser that we used for Java was the version of Xerces that comes bundled with JAXP. Most interesting perhaps are the XML binding frameworks: JAXB that can convert XML documents to Java objects while the XmlSerializer class in .NET performs the same function for C#. What we ultimately aimed to realize was to understand the whole space of XML processing in Java and .NET as described by the following matrix.

		Parse	**Access**	**Modify**	**Serialize**
Stream	SAX			X	X
	Pull Parser			X	X
DOM	Java				
	.NET				
Binding	JAXB				
	XmlSerializer				

Table 1.

X = not implemented. We can now provide exact definitions of our implementations of the XML document lifecycle functions.

Parse	Build a complete document tree in memory (in the case of DOM and binding) and optionally validate it. In the case of SAX and pull parser, scan through the whole document and validate it against the invoice schema (if so specified).
Complete Access	Retrieve the Currency attribute and the GrossUnitPriceAmount for each LineItem. Also retrieve the total PriceAmount from the Summary section.
Selective Access	Same as complete access except that it retrieves the GrossUnitPriceAmount for only a specified percentage of lineitems. Also does not access the Summary.
Content Modification	Increase the GrossUnitPriceAmount for all lineitems as well as the PriceAmount in the Summary by 10%. Not implemented for SAX / pull parser.
Structure Modification	Delete a specified number of lineitems and insert the same number at the end of the set of LineItems. The new lineitems must have LineIDs in the properly increasing sequence. Not implemented for SAX / pull parser.
Serialize	Convert the entire in-memory tree to a serialized XML form written to a stream. Not implemented for SAX / pull parser.

4. Stream Parsers: SAX and Pull Parsing

4.1. The Java SAX Parser

Stream-based, forward-only, read-only parsers are often preferred because of their smaller memory requirements. However the stream based parsers available in Java and .NET are markedly different in their execution models. Java uses a push-based model called Simple API for XML (SAX) (http://www.saxproject.org/) that lets the parser itself drive the parsing process. User code can get control via a callback mechanism to a *SAXEventHandler* class registered by the application. Most commonly, the user applications need to implement the *startElement* and *endElement* functions that are called by the parser when the appropriate events are encountered. For example, the code to perform the access functions would be as follows.

```
public void startElement(String space, String name,
                                         String raw, Attributes atts) {

  if (name.equals("GrossUnitPriceAmount")) {
        String curr = atts.getValue("currencyId");
        System.out.print(curr + " ");
        isGrossUnitPrice = true;
  }

  if (name.equals("PriceAmount")) {
        String curr = atts.getValue("currencyId");
        System.out.print(curr + " ");
        isPriceAmt = true;
  }
}

public void endElement(String uri, String name, String qName)
          throws SAXException {

  if(name.equals("GrossUnitPriceAmount")) {
       isGrossUnitPrice = false;

       if (++linesRead >= selectLines)
             throw new SAXException("Done walk");
  }

  if(name.equals("PriceAmount"))
       isPriceAmt = false;
}
```

One of the characteristics of stream-based parsers is that these functions would be called for **any** element regardless of the fact that we are only interested in two types. This source of inefficiency results in a lot of string comparisons to match the element names. To retrieve the textual content of an element, we have to use the *characters* function.

```
public void characters(char[] ch, int start, int length) {
   if(isGrossUnitPrice || isPriceAmt)
       System.out.println(new String(ch,start,length));
}
```

A particular intricacy of SAX is the need to use state variables (in this case *isGrossUnitPrice* and *isPriceAmt*) that are used to keep track of the current location of the parser. This is necessary because the callback functions do not call each other and so cannot pass state directly through arguments. As the parser makes calls to these event handlers, the only way to communicate state is by such class member variables. The additional complexity of this programming model has been cited as a drawback of SAX, particularly when more complicated state machines such as push-down automata have to be built in the user code to keep track of the parser state. However it is clear that more diligent schema design where element names are unique across different parent elements will simplify the problem.

Another interesting point to note is the use of a *SAXException* to terminate parsing in the case of selective access once we have found the required number of lineitems. This is a way to avoid parsing the complete document when we do not need to. Finally we need to point out that it is difficult to isolate the cost of parsing from the cost of access because it all gets done in one step. The only way of doing that might be to time the parsing using a default event handler that does not call any user code and compare that to the above example.

4.2. The .NET Pull Parser

The streaming parser implementation in .NET uses a pull-based model where the user code is more in control. Unlike SAX where the parser calls back to the application on appropriate events, in pull parsing, the application makes explicit calls to the parser to retrieve the next token. The following program snippet performs the equivalent selective access.

```
for (int i = 0; i < selectLines; i++)
{
  while (reader.MoveToContent() != XmlNodeType.Element ||
         (reader.Name != "GrossUnitPriceAmount" &&
         reader.Name != "PriceAmount"))
    reader.Read();

  if (reader.Name == "PriceAmount")
      break;

  // We have found the GrossUnitPriceAmount
  reader.MoveToAttribute("currencyId");
  Console.Write(reader.ReadAttributeValue() + " ");
  reader.MoveToElement();
  Console.WriteLine(reader.ReadElementString());
}

// Print the PriceAmount from the Summary
reader.MoveToAttribute("currencyId");
Console.Write(reader.ReadAttributeValue() + " ");
reader.MoveToElement();
Console.WriteLine(reader.ReadElementString();
```

Pull parsers use a different model for storing the parser state, that some developers may find more intuitive to program to. Since all the above processing is being done in one single function, the state of the parser can be sort of recorded in the program flow. For example, once the program has exited the *for* loop, then we know that we have finished with all the lineitems. In situations where the implementation of a stack is required, that can be done fairly naturally by using the programs own stack and recursive function calls. This is the hallmark of a top-down parser.

Having said all that, our results indicate that the efficiency claims made by pull-parsing advocates are often exaggerated. Both SAX and pull-parsing are stream based parsers, meaning that the underlying parser must scan all document elements in proper sequence. So does the application program, which in both cases must compare the current element name to the ones it is interested in. Notice that the above C#/.NET program must make the same number of string comparisons with *GrossUnitPriceAmount* and *PriceAmount* as the Java/SAX program. In fact the pull parser application will encounter additional whitespace nodes that are not elements and these must be skipped over by the *MoveToContent* calls. Note that neither of these models support random access, i.e. it it not possible to directly access the *PriceAmount* node as is possible in the case of DOM. In fact the pull parser requires additional navigational functions such as *MoveToAttribute* and *MoveToElement* that are not required for SAX.

5. Tree Builders: The DOM Model

The versions of DOM supported by Java and .NET are rather similar and as a result the implementations port easily from Java to C# and vice versa. However the names of the actual classes used in the System.Xml namespace in .NET and and in the org.w3c.dom package in Java are consistently different. The following table summarizes the differences.

Java	**.NET**
Document	XmlDocument
Element	XmlElement
Node	XmlNode
NodeList	XmlNodeList
Node.getNodeValue	XmlNode.InnerText
DocumentBuilder.parse	XmlDocument.Load
XmlSerializer.serialize	XmlDocument.Save

Table 2.

The actual implementations of the parsers are also quite different. While Java just puts a JAXP wrapper around the Xerces parser, the .NET DOM parser is actually layered on top of the .NET pull parser. This obviously causes differences in parsing performance. It also calls for a different mechanism for XML schema validation. While in JAXP this is done by setting several attributes for the *DocumentBuilderFactory* that generates the parser, for .NET validation is done by using an *XmlValidatingReader* as the underlying pull-parser.

The access test in DOM involves quite a few string comparisons as the following Java code shows. (The C#/.NET code is very similar). However once the tree is built in memory, it is possible to directly access nodes by position (in the children list of the parent element node). That would make it more efficient than SAX, as long as the cost of parsing the document is paid in advance. One of the factors that affects the ease of tree traversal is the presence or absence of *ignoreable whitespace* nodes that represents whitespace in-between two element tags but not contained in a text node (in which case it is called significant whitespace).

```
for (int i = 0; i < selectLines && li != null;)
{
  if (li.getNodeType() == Node.ELEMENT_NODE &&
      li.getNodeName().equals("LineItem"))
  {
      NodeList prices =
        ((Element)li).getElementsByTagName("GrossUnitPriceAmount");

      Element price = (Element) prices.item(0);
      // Print the prices
      System.out.print(price.getAttribute("currencyId") + " ");
      System.out.println(price.getFirstChild().getNodeValue());

      i++;
   }
   li = li.getNextSibling();
}
```

The modification tests also involve string comparisons (because we have to locate the GrossUnitPriceAmounts to increase or the LineItems to insert or delete). In addition there is also a large amount of object creations and deletions. This along with the fact that the complete document tree has to be kept in memory, creates a lot of pressure on the heap of the JVM and the CLR. The platform that does a better job of managing garbage collection performs better in this test. Also being able to increase the size of the heap or utilize parallel GC (as is possible for JVMs) offers particular advantage on machines having large amounts of memory or multiple CPUs.

Serializing XML from the in-memory tree is not a part of the current DOM or JAXP APIs. Therefore we had to use the *XMLSerializer* class that comes with the Xerces parser implementation. In the .NET case this can be done using the extension method *save* that is part of the *XmlDocument* class.

6. Binding Frameworks: JAXB and XmlSerializer

Binding XML to business objects automatically is an area of growing interest. Not only is this the most straight-forward application of XML in the area of web services, it is also the most programmer friendly of all the XML parsing paradigms. This is because the programmer needs to deal only with Java and C# objects that represent business entities. The code for marshalling them to XML or for unmarshalling an XML document to an object tree is automatically generated by a schema compiler and can be invoked by simply calling deserialize/unmarshall or serialize/marshall on the root element class. This is possible because of the type rich nature of an XML schema specification, that naturally determines the class structures that correspond to the schema. For example, given this outline of the invoice schema

```
<complexType name="InvoiceType">
   <complexContent>
      <sequence>
         <element name="Header" type="InvoiceHeaderType"/>
         <element name="LineItem" type="InvoiceLineItemType"
                           minOccurs="1" maxOccurs="unbounded"/>
         <element name="Summary" type="InvoiceSummaryType"/>
      </sequence>
   </complexContent>
</complexType>
```

the *xjc.sh* utility that comes with JAXB generates the following Java *interface*

```
public interface InvoiceType {

  InvoiceSummaryType getSummary();

  void setSummary(InvoiceSummaryType value);

  java.util.List getLineItem();

  InvoiceHeaderType getHeader();

  void setHeader(InvoiceHeaderType value);
}
```

This interface is what the JAXB user directly uses. However there is also an underlying class (also generated by JAXB) that implements this interface. In addition to having member objects for *Summary*, *Header* and the *LineItem* list, and implementing the corresponding getter and setter functions, this class also contains the code for marshalling and unmarshalling the *Invoice* (including relevant attributes and contained objects) to and from XML. In contrast, the class generated by the *xsd.exe* schema compiler that comes with .NET is rather thread-bare.

```
public class InvoiceType {

  public InvoiceHeaderType Header;

  [System.Xml.Serialization.XmlElementAttribute("LineItem")]
  public InvoiceLineItemType[] LineItem;

  public InvoiceSummaryType Summary;
}
```

Note the complete absence of any methods or any code at all in the *InvoiceType* class generated by .NET. The application is expected to access the member objects directly. What seems to happen at deserialization/unmarshall time when an *XmlSerializer* object is initialized with the above *InvoiceType* as argument, is that the .NET framework generates a temporary class under the covers containing the deserialization/serialization code.

Having a rich implementation class confers several specific advantages to the JAXB implementation. Clearly, there is the opportunity to do a more detailed validation either at marshall/unmarshall time or on demand (using the *Validator* class). For example if application wants to enforce the *minOccurs="1" maxOccurs="unbounded"* restriction for the *LineItem* element, there just is not enough information to do that in the *InvoiceType* generated by .NET. On the other hand, it would be straight-forward to encode this restriction in the *createValidator* method of the underlying JAXB generated class for *InvoiceType*. Another point worth commenting on is the choice of an *array* to represent a multiply-occurring element in the .NET framework as compared to a *list* in the case of JAXB. Besides being obviously more expensive when performing insertions and deletions, arrays also have the disadvantage of being of fixed size. For example if an application wants to add just one additional *LineItem* to an *Invoice* it has just unmarshalled, it has to allocate a brand-new array and copy all the existing lineitems over before adding the new one.

We also noted some deficiencies of the *xsd.exe* schema compiler in .NET regarding its inability to handle multiple namespaces in the same schema and also the lack of support for the *import* directive that allows other schemas to be included in the referring schema. Both these requirements are likely to prove crucial in large, industrial-quality XML schemas such as those defined by UBL. Both these requirements are met seamlessly by the *xjc.sh* compiler in JAXB. On the other hand, only .NET allows the facility of converting a set of C# classes to a new XML Schema. This is possible with the help of additional annotations such as *[System.Xml.Serialization.XmlElementAttribute("LineItem")]* as shown above. This gives a hint to the schema compiler that *LineItem* is a sub-element of *Invoice*. There are similar annotations to specify attributes instead. Currently, Java does not allow these kinds of annotations and so there is no way of unambiguously mapping arbitrary Java classes to an XML schema.

The deserialize/unmarshall functions for XML binding are necessarily more expensive than the parse function in DOM because of the additional cost of creating strongly typed objects (such as InvoiceType) instead of a generic *Node*. In addition, a certain level of validation is mandatory to ensure that the objects contained in an *Invoice* are really of type *Header*, *Summary* and *LineItem*. In fact, JAXB can do stricter validation to enforce the cardinality of multiply-occurring elements. A similar argument holds for serialization/marshalling.

It is in the access and modification tests however that both XML binding techologies prove to be far superior to the DOM and streaming-parser implementations. This is because locating the desired node (either for access or modification) in the other parsing techniques require a bunch of string comparisons. On the other hand, locating the *GrossUnitPriceAmount* for the first *LineItem* is a matter of simply following pointers. This boils down to *invoice.LineItem[0].GrossUnitPriceAmount* in .NET and *invoice.getLineItem().get(0).getGrossUnitPriceAmount()* in JAXB.

7. Tying it all together

Both the J2EE platform and the .NET framework offer a multitude of XML processing technologies to the application developer. These can broadly be classified into streaming parsers, DOM implementations and XML binding frameworks. While the two DOM implementations are virtually identical, we have found the SAX and pull parser to be surprisingly similar in capabilities and processing cost. In the area of XML binding though Java and .NET use very different implementation ideas, the interfaces exposed to the developer are conceptually quite similar.

Perhaps the more important decision facing the XML developer today is the choice of the three different parsing techniques. If memory is an issue, then obviously only the streaming parsers make the cut, otherwise both DOM and XML binding offer more flexible alternatives. This is particularly true if there is a need to access elements in reverse, random or in any order different from the natural sequence of the document. Of course if there is a need for multiple passes through the document or for content or structural modifications, only DOM and XML binding parsers are viable. DOM clearly results in more portable application programs and lower costs for parsing and serialization. On the other hand, if there is frequent and repeated access of elements or modification of the document tree, the XML binding frameworks are hands-down winners. The right choice of XML processing technology is likely to greatly influence the ease of application development as well as be a major determinant of run-time performance.

Glossary

J2EE	Java 2 Platform, Enterprise Edition
SAX	Simple API for XML

Biography

Biswadeep **Nag**
Staff Engineer
Sun Microsystems, Inc.
Performance and Availability Engineering
Menlo Park
United States of America
biswadeep.nag@sun.com

Biswadeep Nag has a Ph.D. in Computer Science from the University of Wisconsin-Madison. He presents regularly at conferences on databases, web services, JavaOne and OracleWorld. Dr. Nag is currently working at Sun Microsystems on areas related to database performance, XML processing and web services performance. Inside Sun he is an active presenter on database and XML performance issues.