XML Processing Performance in Java and .NET

The XML Performance Team

Sun Microsystems Inc.

The Promise of XML

The processing of XML documents is assuming growing importance in modern-day IT infrastructures. 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 web services 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 consider the performance of the most popular technologies for processing XML documents that are available in the two dominant software platforms today: Java and .NET. Both Java and .NET offer similar facilities for parsing, manipulating and creating XML documents. We consider the streaming and DOM parsers in both platforms and compare their performance.

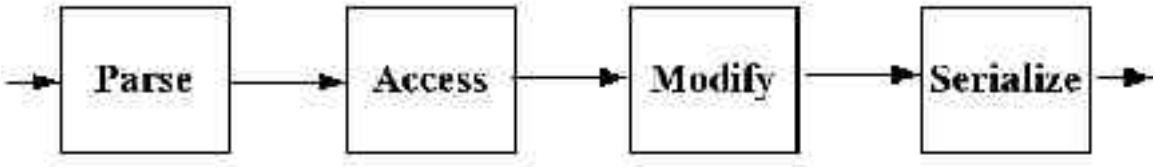
Summary of Findings

There are two classes of XML parsers: streaming parsers (e.g. SAX and pull parsers) and tree building parsers (e.g. DOM). Our tests can be partitioned into these two categories. In the case of streaming parsers, we found that the Java SAX parser outperformed the .NET pull parser for the majority of use cases. When comparing tree-building parsers, the contrasts were even more stark. The Java DOM implementation was significantly faster than the .NET DOM parser. We took care to ensure that the application code used on both platforms are as similar as possible in order to make an apples-to-apples performance comparison. Rather than publish our results here we provide the reader with the information and tools necessary to conduct the test independently and confirm the superior XML processing performance of Java technology.

Test Description

XML Test is an XML processing test developed at Sun Microsystems. It is designed to mimic the processing that takes place in the lifecycle of an XML document. Typically that involves the following steps:

- **Parse** - Scan through the XML document processing elements and attributes and possibly build an in-memory tree (for DOM parsers). Parsing is a pre-requisite for any processing of an XML document.
- **Access** - Extract the data from the elements and attributes of parts of the document into the application program. For example, given an XML document for an invoice, the application might want to retrieve the prices for each item in the invoice.
- **Modify** - Change the textual content of elements or attributes and possibly also the structure of the document by inserting or deleting elements. This does not apply to streaming parsers. As an example, an application might wish to update the prices of some of the items in an invoice or may want to insert or delete some items.
- **Serialize** - Convert the in-memory tree representation to a textual form that is written to a disk file or forwarded to a network stream. This makes sense only for tree building parsers and is necessary in the cases where the XML document has been modified in memory.



XML Test simulates a multi-threaded server program that processes multiple XML documents in parallel. This is very similar to an application server that deploys web services and concurrently processes a number of XML documents that arrive in client requests. Since we wanted to concentrate on XML processing performance, rather than use some sort of web container, we designed a standalone multi-threaded program implemented in both Java and C# that processed XML document files. To avoid the effect of file I/O, the documents are read from and written to memory streams.

XML Test measures the throughput of a system processing XML documents. The notion of an XML transaction here corresponds to a complete lifecycle of an XML document. For tree building parsers this requires the four steps of parse, access, modify and serialize while for streaming parsers it just involves parse and access. XML Test reports one metric: **Throughput** - Average number of XML transactions executed per second.

The XML documents used with XML Test are based on a business invoice document. Each invoice has a fixed length header and summary and a variable number of lineitems. The schematic of the invoice schema is given below in XML schema form.

```
<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>
```

Though this schema can be used to generate an invoice document of almost any size, we made use of two particular document sizes in our experiments. Streaming parsers are typically used to process large XML documents, and so to compare SAX and the pull parser, we used an invoice with 1000 lineitems (about 900 KB). Tree-building DOM parsers have to operate under memory constraints since they construct a representation of the whole document in memory. Therefore for the DOM parsers we used a smaller invoice document containing 100 lineitems (about 90 KB).

Based on the above schema the lifecycle of an XML document as applicable to XML Test can be defined as follows.

- **Parse** - Build a complete document tree in memory (in the case of DOM). In the case of streaming parsers, scan through the XML document.
- **Access** - Retrieve the Currency attribute and the PriceAmount for the number of LineItems specified.
- **Content Modification** - Increase the PriceAmount for all lineitems 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.

Test Details

XML Test has been used to characterize these XML parsers:

	Java	**.NET**
Streaming	SAX	Pull
Tree Building	DOM	DOM

XML Test can be configured using the following parameters:

- **Number of threads** - This is tuned to maximize CPU utilization and system throughput.
- **StreamUsage –** Whether stream parsers are being tested.
- **DOMUsage –** Whether DOM parsers are being tested.
- **Selection** - The percentage of lineitems retrieved in the access phase.
- **ContentModification** - The percentage of lineitems whose textual content is modified.
- **Structure Modification** - The percentage of lineitems inserted or deleted from the DOM tree.
- **RampUp** - Time allotted for warmup of the system.
- **SteadyState** - The interval when transaction throughput is measured.
- **RampDown** - Time allotted for rampdown, completing transactions in flight.
- **XmlFiles** - The actual XML documents used by XML Test.

XML Test reads these properties at initialization into an in-memory structure that is then accessed by each thread to initiate a transaction as per the defined mix. To keep things as simple as possible, XML Test is a single-tier system where the test driver that instantiates an XML transaction is part of each worker thread. A new transaction is started as soon as a prior transaction is completed (there is no think time). The number of transactions executed and the response time is accumulated during the SteadyState period and is reported at the end of the run.

System Configuration

We ran XML Test on the following system config (the same hardware and Windows operating system was used for both Java and .NET):

- Dell Server 4600
- 2 Intel Xeon cpus at 2.6 GHz (2-way hyper-threaded)
- 2 GB main memory
- Windows Server 2003 Standard Edition
- .NET framework 1.1
- J2SE 1.4.2 SDK (uses 1 GB heap)

- JWSDP 1.3 containing JAXP 1.2.4

Streaming Parser Performance

As mentioned before, the streaming parser test (SAX for Java and pull parser for C#/.NET) involves parsing and accessing the PriceAmounts for a certain percentage of lineitems in the invoice document. The tests show that for most selection percentages, the push-based SAX parser (Xerces, shipped with JAXP) outperforms the pull parser from Microsoft .NET.

Although that may seem counter-intuitive to some, this finding is easily explained. 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 C#/.NET program must make the same number of string comparisons of element names with `PriceAmount` as the Java/SAX program. Note that neither of these models supports 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. Microsoft makes the argument (http://msdn.microsoft.com/msdnmag/issues/01/01/xml/default.aspx and http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcomparingxmlreadertosaxreader.asp) that the pull parser has the ability to skip over unwanted content. Though this feature appears attractive, a careful examination shows that it is actually less useful than it appears. The Skip method of the XmlReader class just allows the application to skip over the contents of the current element upto the end tag. This includes all children of the current node. Note however that if the XmlReader is currently positioned deep inside the element hierarchy (or at the leaf elements), such as the PriceAmount of the LineItem, the Skip method just moves to the next sibling element (i.e. it degenerates to the Read method for retrieving the next element). In the Microsoft .NET framework, there is no good way to go back up the tree to skip to a completely different branch (such as the next LineItem).

In short, the pull parser may have a slight advantage when only very few sections of the document are required by the application but the Java SAX parser performs significantly better when most of the document has to be processed.

DOM Parsers Performance

Tree building DOM parsers are used when direct access to some elements of the XML document are required. Our first test using DOM involved selecting the PriceAmounts of a certain percentage of LineItems (after building the DOM tree in memory). Functionally this is similar to the SAX test. The tests reveal that for *all* ranges of selection, the Java DOM parser has significantly higher system throughput than the .NET DOM parser.

A second major use of DOM is in modifying the content or structure of the XML document. We did a series of tests involving content and structure modification, varying

the percentages of lineitems in the invoice document that were modified (content - CM) or deleted and inserted (structure - SM). This test lays heavy stress on the garbage collection mechanism of the system because of the large numbers of objects created and discarded. We found that in the case of DOM modification (both element textual content and DOM structure), the Java Xerces DOM implementation clearly outperforms the C#/.NET DOM implementation by a dramatic margin, probably because of a superior garbage collection mechanism. In fact when most of the document needs to be modified, then the C# test does not even complete.

Conclusion

XML documents form the foundation of data exchange in service-oriented architectures. The performance of XML processing systems is therefore a crucial component when evaluating the scalability of alternative web services platforms. In this paper we used XML Test to compare the performance of the Java and .NET platforms when processing realistic, industrial-strength XML documents. We strongly encourage the reader to conduct the test independently and review the specific performance results. The reader willconfirm our findings: that for almost all meaningful cases, the XML processing solutions of the Java platform offer the best in performance and scalability. Moreover, since the Java platform is completely portable, developers can expect to see this top-of-the-line performance on the Linux and Solaris platforms as well.