# UNIVERSITY OF TECHNOLOGY OF MAURITIUS

## COHORT : MSE 2012 PT

**STUDENT: Gulshan Bhaugeerothee**

**ID: 120463**

**MODULE: Service Oriented Architecture**

## Introduction

The extensible markup language (**XML**) is currently used almost in every applications around the web.

It is a mean of storing and management of data in a very efficient way which is the root of its success. However because it is verbose and of self-descriptive nature, XML can become the subject of big performance issues in an application.

For the data to be usable, the XML file has to be read, analyzed, and the data has to be broken down to parts and categorized in a proper way. This process is called **parsing**.

First way of speeding up that process, to limit performance issues, is to upgrade the hardware with more processors for example but that will be very cost effective so that is where the types of parses come into play. Currently in 2012 there are 4 types of parses which has been identified/created by programmers all around the world to make the parsing process much cost effective.

Below are the named types of parsers:

1. DOM
2. SAX
3. StAX
4. VTD

The above methods of parsing modes have each a unique way of interpreting the XML data which distinguish them from each other with different performance “achievements” in the application and different ways of representing the data in the XML file.

## DOM (Document Object Model)

With DOM , a programmer can create whatever the structure he wants as long as all the processing required to implement the semantics are done or executed.

To be stronger , the DOM parser must be able to do these things :

1. When looking for an element
   1. Ignore comments , attributes and processing instructions
   2. Allow for the possibility that subelements don’t occur in the expected order
   3. Skip over text (CDATA) nodes that can contain ignorable whitespaces, if not validated.
2. When extracting data from the node
   1. Extract text CDATA nodes as well as text nodes
   2. Ignore comments , attributes and processing instructions when gathering as text
   3. If an entity reference node or another element node is encountered , recurse (apply text extraction to all subnodes).

All the processing is done in the memory which may prove to be a huge disadvantage if ever the XML file is big and contains lots of data. It is highly recommended to parse small XML files in DOM parser.

## SAX (Simple API for XML)

SAX is fast and efficient, but its event model makes it most useful for such state-independent filtering.

SAX works by calling methods upon different types of events , like for example upon detecting an opening node , it can call a method and for finding a text it can call a different type of method.

SAX uses less memory than the DOM parser, this is due to the absence of the tree structure for the XML data representation (like in DOM) because SAX simply sends information to the application as it is being read. This parser acts as an input/output stream. It cannot go back a phase or skip ahead to a next position.

## StAX (Streaming API for XML)

STAX is a hybrid invention of the DOM & SAX parsers. STAX starts by analyzing each character of the file and its location is determined with the help of a cursor to know which character has already been parsed , which hasn’t and where in the document it is situated.Totally different from its primary parent SAX which is an event based API, STAX fetches its data needed from the parser whenever the cursor is moved forward and a condition is met. Interestingly STAX can also do both XML writing and reading.

The StAX specification defines a number of uses cases for the API:

* **Data binding**
  + Unmarshalling an XML document
  + Marshalling an XML document
  + Parallel document processing
  + Wireless communication
* **SOAP message processing**
  + Parsing simple predictable structures
  + Parsing graph representations with forward references
  + Parsing WSDL
* **Virtual data sources**
  + Viewing as XML data stored in databases
  + Viewing data in Java objects created by XML data binding
  + Navigating a DOM tree as a stream of events
* **Parsing specific XML vocabularies**
* **Pipelined XML processing**

## VTD (Virtual Token Descriptor)

This is what can be called as the revolutionary XML parser which is known till date to be the world’s fastest XML parser, 1.5 -> 3 times faster than a SAX parser with null content handler and numerous other advantages.

VTD is usually used for large XML files , that DOM cannot handle , performance critical web applications / services or network based XML content.

Of course there are also some disadvantages like for example

* Does not yet support external entities
* Not yet process DTD
* Very long node names will return a parse exception in the application

It uses 64-bit integers to encode offsets , lengths, token types , depths while the XML document is kept undecoded.

Below are some classes of a VTD parser:

VTDGen : encapsulates the parsing , indexing routines

VTDNav : navigator allows cursor based random access and various functions operating on VTD records.

Autopilot : containes xpaths and node iteration functions

XMLModifier : Incrementally update XML

**XML processing operational characteristics**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Xml processing stage** | **DOM** | **SAX(push)** | **StAX(pull)** | **VTD** |
| Parsing | 1. Extract token as objects. | 1. Extract token as objects. | 1. Extract token as objects. | 1. Use integers to extract tokens. |
|  | 2. Build tree by objects. | 2. Create events by objects. | 2. Create events by objects. | 2. Builds location caches and VTD records |
|  | 3. Not ready for access | 3. Ready for access(go to step 8) | 3. Ready for access(go to step 8) | 3. Not ready for access |
|  | 4. Do not destroy and object. | 4. Destroy object after handling the event. | 4. Destroy object after handling the event. | 4. Do not destroy and object. |
|  | 5. Repeat from step 1 until all tokens are processed. | 5. Repeat from step 1 until all tokens are processed. | 5. Repeat from step 1 until all tokens are processed. | 5. Repeat from step 1 until all tokens are processed. |
|  | 6. Destroy the original document after building the entire document (optional). | 6. Destroy the original document after building the entire document (optional). | 6. Destroy the original document after building the entire document (optional). | 6. Kept the original document in the memory. |
|  | 7. Ready for access. | 7. Access is complete (got to step 9). | 7. Access is complete (got to step 9). | 7. Ready for access. |
| Access | 8. Back and forth access: parsing provides sufficient data structures (tree). | 8. Sequential access (no skip): The application creates its own data structures if more advance access or modification is required (go to step 4). | 8. Sequential access (skip forward): The application creates its own data structures if more advance access or modification is required (go to step 4). | 8. Back and forth access: parsing provides sufficient data structures (VR and LC). |
| Modification | 9. Update the tree. | 9. Update the data structure from step 8. | 9. Update the data structure from step 8. | 9. Update by making new copy of the document. |
|  | 10. Write the tree in xml format. | 10. Write the data structure from step 9 in xml format. | 10. Write the data structure from step 9 in xml format. | 10. The document is already in xml format. |
|  | 11. Destroy the tree. | 11. Destroy the data structure. | 11. Destroy the data structure. | 11. Destroy the VR and LC. |

# References

<http://research.binghamton.edu/Innovation/RB262.php>

<http://docs.oracle.com/javaee/1.4/tutorial/doc/JAXPSAX2.html#wp72451>

<http://www.ximpleware.com/vtd-xml_intro.pdf>