Design of a Pull and Push Parser System for Streaming XML

Aleksander Slominski
Indiana University Computer Science Department *
{e-mail: aslom@extreme.indiana.edu }

Abstract

An XML parser facilitates in simplifying the process of manipulating XML documents. The two commonly used models for parsing XML are pull and push. The Simple API for XML (SAX) [5] is the industry standard for parsing based on the push model. However, no standard exists for pull parsing. In this paper we propose the design, API and implementation, XML Pull Parser 2.0 (XPP2), that allows for both pull and push based parsing. We discuss the various features of SAX and provide a description of existing technologies to parse XML. We describe our implementation of this API and show how a SAX driver can be built on top of XPP2.

Keywords: XML, parsing, push, pull, SAX.

1 Introduction

This paper addresses the following questions:

- What are the shortcomings of the event based push model such as SAX? Why is not the optimal solution for applications?
- How does one design an XML parser that provides benefits of both the push and pull parsing model?
- Is it possible to have a simple and efficient pull API for this purpose?
- What are the design decisions that allow for high performance and low memory utilization?

2 Current XML APIs for streaming parsing

The XML 1.0 [6] specification was adopted in February 1998 as a W3C Recommendation. A number of programming APIs have since been developed to parser XML. These include SAX,

*Department of Computer Science, 150 S Woodlawn Avenue, Bloomington, IN 47405. Ph: 812 855 8305 Fax: 812 855 4829

DOM, JDOM, DOM4J, libxml, RXP and NanoXML. Amongst these SAX has emerged as a de-facto standard for event-based XML parsing. It was developed as collaborative effort by the members of the XML-DEV mailing list. The current version of SAX, named SAX2, has support for XML namespaces [3], filter chains, querying and setting features and properties in the parser.

There exist some APIs for pull parsing but none in widespread use. For example kXML [9] is pull parser specifically designed for small devices and needs to be used with Java 2 Micro Edition. It provides simple API however the API and the implementation are tied together. With kXML it is easy to create an XML object tree in memory. However it is not easy to achieve streaming performance as for each event a new object to represent it is created and returned to the user even if the user just wants to skip parts of XML.

Xerces Pullable is a pull parser API that is used by Xalan. It has a pullable SAX model. The version for Xerces1 and Xerces 2 are both different. An application needs to request the Xerces parser [2] to parse only some portion of input and as soon as the parser invokes SAX callback it stops parsing. Then the application can ask for more input that it needs to continue parsing. This is a very good approach for applications that have already invested in SAX infrastructure but want more control over parsing. However this API is not standardized and building more clearly defined pull API is desirable. This is exactly the intent of XPP2 API, described in later sections, to allow to use Xerces 2 Native Interface (XNI) with API that was specifically designed for XML pull parsing.

GNOME C libxml [4] and XMLIO [7] are other examples of XML parsers. GNOME C libxml is not a streaming parser as it just produces a list of events that can be traversed. XMLIO allows for pulling data from XML but its API is specifically designed to unmarshal data structures.

3 Push and Pull: complementary sides of XML parsing

Most SAX parsers are built on top of a pull parsing layer. It is an interesting challenge to expose to the user both pull and push layers. This allows an application to use pull parsing when needed without having to stop using SAX API.

Pull and push parsing models are not the only two ways to parse XML. It is possible to convert pull parser into a push model. This is possible as during pull parsing the caller has control over parsing and can push events (as an example please see description SAX2 driver for XPP2 at the end of this paper). It is also possible to convert push into pull parser but requires to buffer all events converted from SAX callbacks or an extra thread that can be used to "pull" more data from SAX parser and is kept suspended until the user asks for more events.

This approach is best exemplified by Pull Parser Wrapper for SAX [8] that allows conversion from SAX compliant parser into a XML pull parser. However as it requires an extra thread to convert events and this requirement has proved to be very difficult in server environments and impossible in EJB container environments. For an example of such difficulties it is useful to look on design considerations of Apache SOAP Axis (see for example notes from Axis Face-to-Face Meeting at http://xml.apache.org/axis/docs/F2F-2.html)

Pull parsing is ideally suited for applications that needs to transform input XML to other

formats. As such transformation is typically complex they must maintain state during parsing. Using SAX would require to maintain state between callbacks to be able to determine correct action to SAX event. In pull parsing, application can be naturally structured and information can be pulled from XML when needed as application can pull next event when is ready to process it.

Lets look at an example data record below that represents information about one person that has name and two addresses:

```
<person>
<name>Alek</name>
<home_address>
<street>101 Sweet Home</street>
<phone>333-3333</phone>
</home_address>
<work_address>
<street>303 Office Street</street>
<phone>444-4444</phone>
</work_address>
</person>
```

This XML input can naturally be mapped to Java classes:

```
class Person {
        String name;
        Address homeAddress;
        Address workAddress;
}

class Address {
        String street;
        String phone;
}
```

To process it with SAX one would need to have startElement callback to do some work depending on start tag name but this will be not enough as phone must be put in different place depending on whether or not the previous tag was home_address or work_address. Here we give an example of such code:

```
    Person person = new Person();
    StringBuffer elementContent = new StringBuffer();
    Address address;

    public void startElement(String uri, String local, String raw,
                             Attributes attrs) throws SAXException {
        buf.clear();
    }
```

```
    public void characters(char ch[], int start, int length)
        throws SAXException {
        buf.append(ch, start, length);
    }
    public void endElement(String uri, String local, String raw)
        throws SAXException {
        if("name".equals(local)) {
           person.name = elementContent.toString;
         } else if("home_address".equals(local)) {
           address = person.homeAddress = new Address();
         } else if("work_address".equals(local)) {
           address = person.workAddress = new Address();
         } else if("phone".equals(local)) {
           address.phone = buf.toString();
         } else if("street".equals(local)) {
           address.street = buf.toString();
         } else {
           throw new SAXException("unexpected element "+local);
         }

    }
```

Although on the first look the code may look sufficient there are some inherent problems because it does not maintain state between endElement callbacks. Therefore the code does not know what is its position in the parsed XML structure. One particular problem with such approach is that such SAX program will not validate input and can even produce incorrect conversions. For example this code would process input XML from below by incorrectly overriding home phone to 666-666 instead of 333-3333.

```
<person>
<name>Alek</name>
<home_address>
<phone>333-3333</phone>
</home_address>
<phone>666-6666</phone>
</person>
```

This can be fixed by keeping track of how deeply nested the start/end element is and using some extra flags to make sure that phone is set only once and address always corresponds only to the current home or work address. But it requires adding extra state variable and code to do validation.

When using pull parser the conversion of XML input into Person object is very natural and follows hierarchical relation between Person and Address objects. Let look on how it could be done if a simple pull parser was available:

```
        parser = new PullParser(input)
```

```
Person person = parsePerson(parser);

public Person parsePerson(PullParser parser) throws ValidationException
{
        Person person = new Person();
        while(true) {
           int eventTyppe = parser.next();
           if(eventType = parser.START_TAG) {
              String tag = parser.getStartTagName();
              if("name",equals(tag)) {
                person.name = readContent(parser);
              } else if("home_address",equals(tag)) {
                person.homeAddress = readAddress(parser);
              } else if("work_address",equals(tag)) {
                person.workAddress = readAddress(parser);
              } else {
                throw new ValidationException(
                  "unknown field "+tag+" in person record");
              }
           } else if(eventType == parser.END_TAG) {
             break;
           } else {
             throw new ValidationException("invalid XML input");
           }
        }
}
public Address parseAddress(PullParser parser) throws ValidationException
{
        Address person = new Address();
        while(true) {
           int eventTyppe = parser.next();
           if(eventType = parser.START_TAG) {
              String tag = parser.getStartTagName();
              if("street",equals(tag)) {
                address.street = readContent(parser);
              } else if("phone",equals(tag)) {
                address.phone = readContent(parser);
              } else {
                throw new ValidationException(
                  "unknown field "+tag+" in person record");
              }
           } else if(eventType == parser.END_TAG) {
             break;
           } else {
             throw new ValidationException("invalid XML input");
```

```
            }
    }
    public String readContent(PullParser parser) throws ValidationException
    {
            if(parser.next() != parser.CONTENT) {
              throw new ValidationException("expected string content");
            }
            String content = parser,readContent();
            if(parser.next() != parser.END_TAG) {
              throw new ValidationException(
                "expected end tag after string content");
            }
            return content;
    }
```

The structure naturally reflects the organization of data structures and therefore is much easier to maintain. The state is kept naturally on the stack as a consequence of method calls that can be nested as much as necessary. Notice also that with pull parsing the second input will trigger a ValidationException.

4 Design of Pull API

As we have seen pull parsing has some advantages and therefore it is very important to provide developers with one consistent and simple API to work with.

For such an API to stay minimal we need to expose only a few things to the user. We need to expose the parser state, such as : START_TAG, END_TAG, CONTENT, END_DOCUMENT. This number of states is minimal for any XML parser. We will also need an ability to to get next state, for example by calling method named next(). Finally for each state we will need to retrieve information associated with the parser state (current event). When we reach END_DOCUMENT there is no more input and in this case no extra information is needed. However for CONTENT, to get element content as String, readContent() can be used. Then for start and end tags we need the name of the tag and its namesapace: getLocalName() and getNamespaceUri(). Finally START_TAG is the hard part as it can also contain the list of attributes declared in this start tag. It is a good idea to make it look like SAX Attributes/AttributeList to simplify potential conversion of pull events into SAX push callbacks.

So we can say that such minimal Pull Parser API should have at least:

```
public interface XmlPullParser {
    /** signal logical end of xml document */
    public final static byte END_DOCUMENT = 1;
    /** start tag was just read */
    public final static byte START_TAG = 2;
    /** end tag was just read */
    public final static byte END_TAG = 3;
```

```
    /** element content was just read */
    public final static byte CONTENT = 4;


    int next() throws PullParserException;
    public String readContent() throws XmlPullParserException;
    public void readEndTag(XmlEndTag etag) throws XmlPullParserException;
    public void readStartTag(XmlStartTag stag) throws XmlPullParserException;
    public String getLoacalName();
    public String getNamespaceUri();
}
```

It is useful to represent XmlStartTag and XmlEndTag as separate interfaces but sharing common functionality in XmlTag. It is possible to have state retrieval methods in XmlPullParser instead of creating extra interfaces such as XmlEndTag and XmlStartTag. However having separate interfaces allows us to maintain parsing state in those classes easily and allows to record state of parsing for any number of steps in the past.

```
public interface XmlTag {
  public String getNamespaceUri();
  public String getLocalName();
}

public interface XmlEndTag extends XmlTag {
}

public interface XmlStartTag extends XmlTag {

    public int getAttributeCount();
    public String getAttributeNamespaceUri(int index);
    public String getAttributeLocalName(int index);
    public String getAttributeValue(int index);
    public String getAttributeValueFromName(String namespaceUri,
                                            String localName);

}
```

This API will have to be extended to allow for more efficient handling of namespace declarations, setting parser input, resetting parser state for parser reuse and some other utility methods. However the core API for XML pull parser will probably be similar to what is shown above.

We should mention here that use of XmlTag interfaces is not essential for such API and may be abandoned when memory size constraints are of premium such as is the case with handheld devices in J2ME. In these cases embedding all state into one XmlPullParser interface will be very advantageous.

5 Implementing XML Pull Parser

Any XML parser will have to do XML tokenization. For push parsers this can be combined with higher level parser as push callbacks can be called as soon as interesting input is read. For pull parsing this is different as when interesting event is seen the pull parser must return to the user. Therefore it needs to maintain internal state to be able to continue parsing when the user requests it.

In XPP2 we have made it possible to parse some parts of input using pull API and then for some document XML subtrees to use push parses that provide SAX2 API. From a SAX2 callback user can continue to use pull parsing or even create another nested SAX2 push parser and this recursive nested parsing can be as deep as required.

5.1 Java Performance Constraints

When implementing XPP2 one of the most important task was to assure good overall performance. We identified that creation of XmlStartTag object for every start tag would be incurring a very high overhead but we did not want to remove possibility to record parser state for later use (otherwise building XML object model in memory from XPP2 events would be difficult). The other consideration was to make sure that the parser performance is not too low when compared with simple string tokenization and other XML parsers. This requirements also helped to estimate acceptable XML parsing overhead. An interesting fact was determining the difference in processing char[] as compared to String even when advanced JIT such as Hotspot is used (see [1] for more details on performance evaluations).

5.2 Tokenizer

The most important part of XML Pull Parser is tokenizer that is responsible for breaking input into tokens that are later used to produce events.

To make this task as efficient as possible the tokenizer in XPP2 is a state machine driven by input characters.

To avoid reading the input character by character, the input is internally buffered. In J2ME it is important to make sure that buffer size does not exceed some hard limits (so that XML parsing does not consume the available memory). If hard limit is exceed such as for very long element content an exception will be thrown and parsing will be stopped. It is also desirable to establish soft limit on internal buffer size to indicate what is desired buffer size (it must be always less than hard limit).

5.3 Parser

The parser in XPP2 is implements all of XML 1.0 specification requirements for non-validating XML parser except for parsing internal DTD. This decision was made to ensure that the size of XPP2 is not too big and moreover as XML schemas are going to replace DTDs we think that supporting DTD parsing is no longer desired by users. The other limitation of the parser is that input must be in UNICODE as represented by Java Reader and

char type. The user is responsible for detecting input encoding (such as UTF8) and use Java built-in mechanisms to transform it to Reader.

5.4 Support for streaming

XPP2 pull model is inherently well suitable for streaming. The internal buffer support for soft and hard limit sizes allow us to ensure that during parsing memory consumption is kept under tight control.

5.5 Namespace handling

In some situation when XML input does not contain XML namespaces there can be slight boost in performance (around 5%) by avoiding overhead of validation and maintaining XML namespace prefixes declarations. Therefore XPP2 allows to decide before parsing if namespaces should switched on or off.

6 Supporting multiple implementations of XPP2 API

XPP2 consists of two distinctive APIs and a default implementation. The XPP2 API was designed to allow plugging different implementations in such way that user code does not have to change at all. This is achieved with use of XmlPullParserFactory that is responsible for creation of XmlPullParser instances. The factory to be used is parametrized based on existence of special file with factory name on CLASSPATH or name of factory class passed to XmlPullParserFactory.newInstance() method. We do not use java System properties by default as they are not available on J2ME platform, however are easy to use with this code:

```
XmlPullParserFactory.newInstance(
  System.getProperty(XmlPullParserFactory.DEFAULT_PROPERTY_NAME));
```

As an example of such additional implementation of XPP2 API we are providing implementation that uses Xerces 2 XNI pull model. Therefore the user can use XPP2 either with fast default implementation or fully validating Xerces2 engine.

6.1 SAX2 driver for XPP2

As it was mentioned earlier it is desirable and clearly feasible to translate pull events into SAX2 push callbacks. As part of XPP2 we are providing utility class that can take XPP2 API (implemented by default implementation or Xerces 2 or any other) and transform it into SAX2 XmlReader.

6.2 C++ version

We have also written a C++ version of XPP1 (Xml Pull Parser version 1). As of now, the C++ version of XPP2 is not available however differences between XPP1 and XPP2 are not that big for just XmlPullParser interface. C++ version has some additional limitations

such as it does not support UNICODE fully (it is possible to compile it with chat or wchar_t support) and user needs to assume that input is encoded as UTF8 (or when compiled with wchar_t assume UTF16). Additionally XPP1/C++ does not support streaming and requires all input to be passed in one array.

7 Conclusions

We hope that this paper will be useful introduction to pull parsing and will give a motivation to use as an alternative to SAX API. It also demonstrates advantages of this alternative mode of parsing, compares pull with push models (with emphasize on SAX2) and argues that both are needed.

An API that allows both push and pull parsing should be made available to users to allow them to use the best of both parsing models (and assure that user has the most powerful tools to accomplish XML related tasks). XPP2 API and its implementations are hopefully a first step into this direction.

We would like to welcome readers to visit our web page http://www.extreme.indiana.edu/soap/xpp/ and browse more detailed documentation, download source code and send us comments.

References

[1] Aleksander Slominski. On Performance of Java XML Parsers, visited 05-01-01. http://www.cs.indiana.edu/~ aslom/exxp/.

[2] Apapche Foundation. Xerces Java Parser 2, visited 09-01-01. http://xml.apache.org/xerces2-j/.

[3] World Wide Web Consortium. Namespaces in XML, 1-14-99. http://www.w3.org/TR/REC-xml-names/.

[4] Daniel Veillard. The XML C library for Gnome (libxml), visited 04-01-01. http://xmlsoft.org/.

[5] D. Megginson et al. Sax 2.0: The simple api for xml, visited 07-01-00. http://www.saxproject.org/.

[6] Tim Bray et al. Extensible markup language (xml) 1.0 (second edition 6 october 2000), visited 03-01-01. http://www.w3.org/TR/2000/REC-xml.

[7] Paul T. Miller. XMLIO - An XML input/output library for C++ applications, visited 01-01-01. http://www.fxtech.com/xmlio/.

[8] Stefan Haustein. XML pull wrapper for SAX parsers, visited 02-01-01. http://www.trantor.de/xml/.

[9] Stefan Haustein. kXML Project, visited 05-01-01. http://www.kxml.org/.