A distributed editing environment for XML documents

Claude Pasquier and Laurent Théry

Inria Sophia Antipolis,
2004, route des Lucioles - BP 93,
06902 Sophia Antipolis, France

Abstract. XML is based on two essential aspects: the modelization of data in a tree like structure and the separation between the information itself and the way it is displayed. XML structures are easily serializable. The separation between an abstract representation and one or several views on it allows the elaboration of specialized interfaces to visualize or modify data. A lot of developments were made to interact with XML data but the use of these applications over the Internet is just starting.
This paper presents a prototype of a distributed editing environment over the Internet. The key point of our system is the way user interactions are handled. Selections and modifications made by a user are not directly reflected on the concrete view, they are serialized in XML and transmitted to a server which applies them to the document and broadcasts updates to the views.
This organization has several advantages. XML documents coding selection and modification operations are usually smaller than the edited document and can be directly processed with a transformation engine which can adapt them to different representations. In addition, several selections or modifications can be combined into an unique XML document. This allows one to update multiple views with different frequencies and fits the requirement of an asynchronous communication mode like HTTP.

1 Introduction

XML is the central point of the far-reaching process of standardization that is going to alter the way information is handled. The deployment of XML related technologies will have a large impact on operations like structured editing, storage, information exchange, data transformation, querying, and rendering.

Two essential aspects of XML have been inherited from SGML [8]: the modelization of data in a tree like structure and the separation between information itself and the way it is displayed. An XML document is typically a serialization of a tree with every node and leaf tagged. It is a convenient user-readable and platform-independent representation, well-suited for transmission over a network. The separation between an abstract representation and one or several views on it allows the elaboration of specialized interfaces to visualize or modify data.

A lot of developments were made to interact with XML data but the use of these applications over the Internet is still under development. The World Wide Web Consortium proposes several standards to visualize XML documents (CSS [4], XSL [13]) and to transform them (XSLT [14]). However, these standards are mainly defined to display documents, not to interact with them. If we consider real e-commerce applications, displaying and modifying data will be needed. For example, a vendor may want to update via Internet (using a portable computer or a mobile phone) the database of its company with new information relative to a visited client.

On new XML browsers, some interaction with a displayed document can nevertheless be realized. Documents are internally modelized as DOM trees and can be manipulated with methods that access public DOM APIs. The problem with this solution is that only the concrete view of a document is edited, not the data itself. This paper presents a prototype of a distributed editing environment over the Internet. Its architecture is based on the software development environment Centaur [1] and ideas presented in [2] and [5]. The key point of our system is the way user interactions are handled. Selections and modifications made by a user are not directly reflected on the concrete view. They are serialized in XML and transmitted to a server which applies them to the document and broadcasts updates to the views. Bidirectional correspondences between a source and a result tree are expressed in the declarative language XPPML. It is used to transform serialized user-interactions made on the source structure to equivalent operations applicable to the result structure.

2 Interacting with XML documents

Editing an XML document over an asynchronous, unstable and rather slow protocol like HTTP implies minimizing the amount of information exchanged. In particular when editing large documents, it is not a very good strategy to retransmit every time the modified document. User interactions are encoded as XML documents. Data transmitted on the network represent actions. The first set of actions declares external operations on the structure (selection, modification, redisplay, etc). The second set concerns updates to be applied on the structure (selection of a given subtree, deletion of a node, etc.). In our prototype, we have defined two actions, selection and modification, that are both based on an unambiguous way to identify a node on a tree.

We take as an example the simple mathematical expression *'2*(8/2+5)'* which will be used throughout this paper. It is expressed by the following XML document called *'exp'*:

```
<?XML version='1.0'?>
<exp>
  <mult>
    <int value='2'/>
    <plus>
      <div>
```

```
        <int value='8'/>
        <int value='2'/>
      </div>
      <int value='5'/>
    </plus>
  </mult>
</exp>
```

2.1 Paths

One can identify a node in the tree with an expression specifying a path to this node. This can be expressed with the XPath [12] standard by starting from the document's root and specifying either the name of all the encountered child nodes or their relative positions. For example, the node *'8/2+5'* can be equally designated by the following XPath expressions:

```
/exp/mult/plus
/*[1]/*[1]/*[2]
```

Note that identifying subtrees by a list of node names is ambiguous, since two different subtrees could have the same path. For this reason, we prefer the second solution. In order to transmit paths, we propose to represent a XPath location which uses relative positions by an ipath (IntegerPath) XML element. The location of the node *'8/2+5'* is thus expressed by:

```
<ipath>
  <move num='1'/>
  <move num='2'/>
</ipath>
```

The starting point of the path is the root node (called documentElement in DOM) and not "the parent of the document element" as it is specified in XPath. Then, the empty element `<ipath/>` identifies the root of the document, `<ipath> <move num='1'/></ipath>` its first child.

2.2 Selection path

We use selection path to designate different elements of a document. These elements may belong to different selections. Selections are represented by symbolic names. A *spath* element is defined by the following DTD:

```
<!ELEMENT spath  (select*, (move, spath)*)>
<!ELEMENT select EMPTY>
<!ATTLIST select name NMTOKEN #REQUIRED>
<!ELEMENT move   EMPTY>
<!ATTLIST move   num  NMTOKEN #REQUIRED>
```

A *spath* element is composed of two sets of elements. The first one is used to declare and name the selections on the current node. The second one is composed of pairs of *move* and *spath* elements. Each *spath* represents the selection path corresponding to the sub-element denoted by the *move*.

For example, a selection called *'selA'* of the node *'5'* and a selection *'selB'* of the node *'8/2+5'* is expressed by the following spath element:

```
<spath>
  <move num='1'/>
  <spath>
    <move num='2'/>
    <spath>
      <select name='selB'/>
      <move num='2'/>
      <spath>
        <select name='selA'/>
      </spath>
    </spath>
  </spath>
</spath>
```

Selection paths are general enough to be used for different purposes. On the client side it can be used to represent extension to an existing selection. On the server side it can be used to represent all the existing selections.

2.3 Modification path

A modification path memorizes modifications done on a document. The following DTD describes the structure of a *mpath* element:

```
<!ELEMENT mpath  (element, (move, mpath)*)>
<!ATTLIST mpath type (move|delete|insert|change) move>
<!ELEMENT element (%targetElt;)>
<!ELEMENT move   EMPTY>
<!ATTLIST move   num  NMTOKEN #REQUIRED>
```

A *mpath* has an attribute *type* which indicates the kind of modification to perform (deletion, insertion or replacement). By default, if no attribute is given, the type is *move*. Values that are changed or inserted are specified under the tag *element*.

As an example, one can evaluate the node *'8/2'* and replace it with its integer value. This is expressed by the following *mpath* element:

```
<mpath>
  <move num='1'/>
  <mpath>
    <move num='2'/>
    <mpath>
```

```
      <move num='1'/>
      <mpath type='change'>
        <element>
          <int value='4'/>
        </element>
      </mpath>
    </mpath>
  </mpath>
</mpath>
```

A modification path can be applied to a document to perform the memorized operations. Note that this application can be delayed. Two *mpath* elements can be combined in a new modification path which amalgamates the successive operations. For example, the previous modification path and the one that corresponds to the deletion of the node identified by the selection *'selA'* gives the following combined path:

```
<mpath>
  <move num='1'/>
  <mpath>
    <move num='2'/>
    <mpath>
      <move num='1'/>
      <mpath type='change'>
        <element>
          <int value='4'/>
        </element>
      </mpath>
    <move num='2'/>
    <mpath type='delete'/>
  </mpath>
</mpath>
```

When applied to a document, this *mpath* will replace the element *'8/2'* with *'4'* and will remove the element *'5'*. The same *mpath* can also be applied to the current selection to compute a new selection path compatible with the updated structure of the document.

Modification paths are generic enough to represent any kind of modification. They represent a convenient way to interact with large documents over a network since only the new elements of the document need to be transmitted.

3 Communication with a user interface

Except for XML editors that let the user directly write the tags of XML documents, standard editing environments propose an interface between the user and the logical structure. Usually, selections and modifications are done on a concrete view of the data. User interactions are transformed into actions on the logical structure and the layout of the view is recomputed.

Our concepts of selection and modification paths can be applied to modelize the communication between a concrete view and an abstract representation. Representing actions by valid XML documents makes it possible to use standard tools to manipulate them. Let's take the example of an XML browser in which the displayed document is obtained by transforming an initial document. All modifications made on the source document can be reflected to the view by transmitting only a modification path.

Modification paths on the logical tree and the concrete tree can largely differ. However, one can be generated from the other by using a transformation process similar to the one used to get the initial document. This means that the transformation tool should be capable not only of processing the initial document but also of transforming selection and modification paths from one structure to another.

Standard transformation languages, like DSSSL [7] or XSLT [14] are well-suited for processing XML documents. However, current implementations of these languages are based on a batch process that transforms the whole source document into a target one. Once this is done, no link is kept between the two structures. In addition, there is no available implementation capable of doing a reverse transformation (from a target structure back to the source one). Adding this dynamic capability to engines based on DSSSL or XSLT standards seems difficult because of the expressiveness of these languages. In our project, we have developed a transformation engine based on XPPML (Xml Pretty-Printing Meta Language) that is strictly less powerful than XSLT but satisfies all our dynamic requirements. It is a modified version of the transformation engine developed for the Aïoli system [11].

4 XPPML

XPPML is an XML extension for the pretty-printing meta language PPML [10] defined in the Centaur system. An XPPML specification is a collection of unparsing rules associated with abstract syntax patterns. The concepts of XPPML are very close to those found in XSLT. Basically, it is a language for transforming XML documents into other representations. A transformation in the XPPML language is expressed as a well-formed XML document. It describes rules for converting a source tree into a result one by associating patterns with templates. The formatting machine generates a result structure by traversing the source tree and looking for a pattern that matches each node. When a match is found, the corresponding template is instantiated to create parts of the result tree. Features of XPPML include contextual formatting, conditional layout over external boolean functions and inclusion of user-defined external functions.

The formatting machine makes use of the XPPML rules to generate the path on the result tree corresponding to a given path on the source one but is also able to retrieve the position on a source structure corresponding to a position on a result one.

A typical XPPML document has the following structure:

```
<x:prettyPrinter ppName="..." langName="..."
         xmlns:x='http://www-sop.inria.fr/lemme/xppml/1.0'>
  <x:extension name="..."/>
  <x:import package="..."/>
  <x:rule>
    ...
  </x:rule>
</x:prettyPrinter>
```

An XPPML specification is fully identified by both the name of the pretty printer and the name of the language to which it applies. This allows one to refer and retrieve, for example, the standard pretty printer for the Java language or the pretty printer Y for the language Z. These two identifiers are stored in the attributes *'ppName'* and *'langName'* of the element *'prettyPrinter'*.

Modularity of pretty-printing declarations is realized through one or several *'extension'* elements where the names of other XPPML definitions to be included are specified. The *'import'* element is used to declare the package where external Java functions should be searched.

The core of the XPPML declaration is defined by a list of *'rule'* elements composed of a *'pattern'* element and a layout. A pattern is defined by a *'template'* element and zero or more additional constraints. The *'match'* attribute in the *'template'* element is a pattern that identifies the source node to which the rule applies. For example, the following XPPML specification defines the layout of our *'exp'* document:

```
<x:prettyPrinter ppName='std' langName='exp'
         xmlns='http://www-sop.inria.fr/lemme/figue/1.0'
         xmlns:x='http://www-sop.inria.fr/lemme/xppml/1.0'>
  <x:rule>
    <x:pattern><x:template match='mult(*x,*y)'/></x:pattern>
    <h>
     <x:variable name='x'/>
     <atom value='*'/>
     <x:variable name='y'/>
    </h>
  </x:rule>
  <x:rule>
    <x:pattern><x:template match='mult/plus(*x,*y)'/></x:pattern>
    <h>
      <atom value='('/>
      <x:variable name='x'/>
      <atom value='+'/>
      <x:variable name='y'/>
```

```
      <atom value=')'/>
    </h>
  </x:rule>
  <x:rule>
    <x:pattern><x:template match='plus(*x,*y)'/></x:pattern>
    <h>
      <x:variable name='x'/>
      <atom value='+'/>
     <x:variable name='y'/>
    </h>
  </x:rule>
  <x:rule>
    <x:pattern><x:template match='x=int'/></x:pattern>
    <x:extFun name='identitypp'>
      <x:arg value='x' type='var'/>
    </x:extFun>
  </x:rule>
</x:prettyPrinter>
```

XML namespaces are used to distinguish between XPPML constructs and elements corresponding to the output structure. This example uses Figue's syntax [3] for the layout but other output format like HTML or xsl:fo can equally be used. Figue is an incremental bi-directional layout engine that handles a limited set of combinators like *'h'* to specify an horizontal layout, *'v'* for a vertical one, *'atom'* for a terminal box and some other specialized mathematical constructs.

The first rule matches a *'mult'* node composed of exactly two children. Its layout part specifies that an horizontal box has to be created (use of Figue's horizontal combinator *'h'*) and filled with the concrete tree obtained by a recursive call of the formatting machine on the node's first child, a concrete leaf with the textual value *'*'* and the layout of the second child.

Contextual pattern are specified in the left-hand side of a rule, as in the second rule which matches a *'plus'* node that have a *'mult'* node as parent. It is also possible to define rules identified by a context name. When a contextual pretty-printing is required, a context name is added to the recursive call.

In the fourth rule, an external Java method called *'identitypp'* is used to display the attribute *'value'* of the node representing an integer. External functions can also be used with conditional layout constructs defined with the XPPML instructions *'if'* and *'case'*.

Editing and rendering are handled by Figue. During the editing process, Figue communicates to the formatter the location of concrete subtrees which have been selected or modified. The formatter retrieves the corresponding abstract trees, performs the necessary modifications and returns back to the layout engine the selection or modification paths applicable on the concrete tree. With this information, Figue performs an incremental update on the view.

Aïoli can be used as a server which communicates with clients through HTTP requests. Documents are stored on the server side where updating and formatting operations are performed. Information manipulated on the server side is represented by DOM objects. These are transformed to XML text format and are transmitted to the client through HTTP(S). Standard HTTP protocol is used; marshaling and unmarshaling are done with application methods.

In our current implementation, several clients can be connected to the same server. Figue is one of the possible clients but standard web browsers can also be used. New clients may join and watch the progress of existing editing sessions but problems specific to the implementation of a concurrent editing environment [9] have not yet been studied.

5 A typical editing session

If our example document is accessible at the url *'docURL'* and our server is located on *'myserver'*, then a typical invocation of the transformation engine is done by the following HTTP request:

```
http://myserver&doc=docURL&ppml=std&type=figue
```

The arguments *'ppml'* and *'type'* correspond respectively to the name of the prettyprinter to use and the type of syntax used to codify the concrete tree. The client first declares a new selection called *current*:

```
clientMess #1: <setSelection type='Single' name='current'/>
```

The server answers with a reference to the new selection. This information is needed because if several clients are accessing the same document, the name of the selection may be already used. In that case, the server returns a selection with a different name:

```
serverMess #1: <selection type="Single" name="current" />
```

Then the client requests for a copy of the document using the appropriate prettyprinting:

```
clientMess #2: <redraw />
```

The server sends the corresponding concrete tree using the Figue syntax:

```
serverMess #2: <redraw><H><Atom Value="2"/><Atom Value="*"/>
               <H><Atom Value="("/><H><Atom Value="8"/>
               <Atom Value="/"/><Atom Value="2"/></H>
               <Atom Value="+"/><Atom Value="5"/>
               <Atom Value=")"/></H></H></redraw>
```

The client is now able to display the expression:

Let's suppose we want to edit the division. We first select the character *'/'* with the mouse:

In order to update a selection, the client needs to send a message to the server. It first computes the *ipath* corresponding to the selected expression in the result tree and sends a message asking for the *current* selection to be modified:

```
clientMess #3: <updateSelection selName='current'><ipath>
               <move num="3"/><move num="2"/><move num="2"/>
               </ipath></updateSelection>
```

The server acknowledges this message:

```
serverMess #3: <done/>
```

The server translates the selection made on the result tree into a selection on the source tree. In our case, it is the tree `<div><int value='8'/> <int value='2'/></div>` that has generated the character *'/'*.

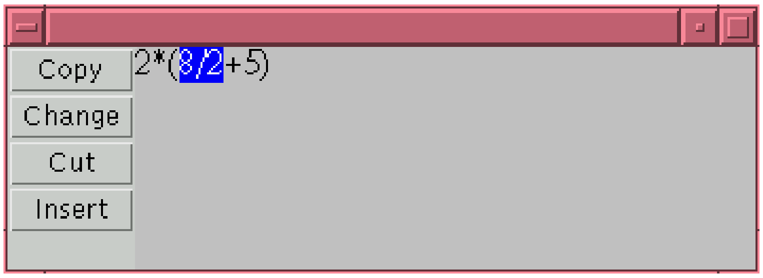
In order to get the value of the selection, the client requests the list of selection's changes that are memorized on the server side:

```
clientMess #4: <commit type="select"/>
```

The server responds with a structure specifying that the *current* selection must be placed on the second child of the third node, i.e. the node representing *'8/2'*.

```
serverMess #4: <commit type="select"><Change>
               <select name="current"/></Change><Path Rank="3">
               <Path Rank="2"><ExtendSelection Name="current" />
               </Path></Path></commit>
```

This information can be used to highlight the current selection in the editor:



Note that what we obtain is a structured selection: by selecting the single character '/' we get the whole division expression. Now that the division is selected, we can decide to evaluate it and replace it with 4:

```
clientMess #5: <change selName='current'><int value="4"/></change>
```

The modification is done on the abstract tree stored on the server side and an acknowledgement is sent.

```
serverMess #5: <done/>
```

To get the modification, the client just asks for the list of modifications that have been done on the abstract structure since the last commit request:

```
clientMess #6: <commit type="modif"/>
```

The server returns the modifications to be applied on the concrete tree:

```
serverMess #6: <commit type="modif"><mpath type="move">
               <move num="3"/><mpath type="move"><move num="2"/>
               <mpath type="change"><element><Atom Value="4"/>
               </element></mpath></mpath></mpath></commit>
```

The client can then reflect these modifications on the editor:

The modification may have changed the values of some selections. So, the client asks for their new values:

```
clientMess #7: <commit type="select"/>
```

The modification has only replaced a selected tree; nothing has changed:

```
serverMess #7: <commit type="select"><Change>
               <select name="current"/></Change><Path Rank="3">
               <Path Rank="2"><ExtendSelection Name="current"/>
               </Path></Path></commit>
```

The client can update the editor with the selection:

6 Conclusion

The solution described in this paper has several advantages. XML documents coding selection and modification operations are usually smaller than the edited document and they are directly processed by a transformation engine which adapts them to several representations. In addition, the possibility to group several selections or modifications into a single XML document allows us to update multiple views with different frequencies and fits the requirement of an asynchronous communication mode like HTTP.

Several clients with different connection speeds can collaborate in a single editing session. For the moment, specific aspects concerning concurrent editing problems have not yet been tackled. We are aware that a lot of problems have to be resolved in order to provide a full featured collaborative environment. Still, we believe that what we have presented here can be used as a valuable basis for more elaborated protocols.

XPPML is less expressive than XSLT. Any XPPML specification can be easily translated into XSLT. XPPML has been designed so that a dynamic link between the source structure and the transformed one could easily be maintained. Doing this for XSLT seems more problematic. For example, a transformation rule in XSLT can have access to any node of the source document, even those localized outside the matched subtree. This means that a single modification in the source document can potentially affect the overall result.

Our system has been successfully tested with the distributed editing of large Java programs. User-interactions are quickly processed and reflected to the client. The unique problem of performance we had to face was only with the initial transmission of the formatted document which may be very large. We are studying possibilities to transmit only the part of the structure needed by the client (the subtree visible in the window for example) or to allow several clients to access different subtrees of a same document.

For the moment, selections and modifications can only be done on nodes of the tree. It is sufficient for editing highly structured structure with few unconstrained text fields, like the kind of documents we have presented here. However, for editing general XML documents, it is necessary to represent selections or modifications of parts of textual fields. The concept of *range* used in the specification of DOM level 2 [6] which represents a selection by a pair *node + offset* will be implemented in the next version of our system.

In our organization, the amount of software on the side of the client is kept to a minimum. It is composed of a communication layer that sends and receives messages over the network and a layout engine. All the other components are concentrated in the server. It is then particularly adapted to situations where clients have sparse resources.

7 Acknowledgements

This work has been done in the framework of Dyade, the Bull-Inria Research Joint Venture.

References

1. P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual, CENTAUR: The system, Proceeding of the Third Symposium for Software Development Environments (SDE3), Boston, December 1988.
2. D. Clément, A distributed architecture for programming environments, Proceedings of the fourth ACM SIGSOFT symposium on Software development environments, December 3-5, 1990, Irvine, CA USA.
3. B. Conductier, L. Hascoët, L. Théry, Figue's Documentation, see http://www-sop.inria.fr/croap/figue/
4. CSS2 World Wide Web Consortium. Cascading Style Sheets, level 2 (CSS2). W3C Recommendation. See http://www.w3.org/TR/1998/REC-CSS2-19980512
5. A.M. Déry and L. Rideau, Distributed Architecture for Programming Environment, INRIA Research Report no 2918, June 1996.
6. DOM-Level-2 W3C (World Wide Web Consortium) DOM Level 2 W3C Recommendation. See http://www.w3.org/TR/DOM-Level-2.
7. DSSSL International Organization for Standardization, International Electrotechnical Commission. ISO/IEC 10179:1996. Document Style Semantics and Specification Language (DSSSL). International Standard.
8. ISO 8879:1986, Information processing – Text and office systems – Standard Generalized Markup Language (SGML).
9. Israel Z. Ben-Shaul, Gail E. Kaiser and George T. Heineman, An Architecture for Multi-User Software Development Environments, Computing Systems, The Journal of the USENIX Association, 6(2):65-103, University of California Press, Spring 1993.
10. Ian Jacobs and Janet Bertot, editors, Centaur 1.2, chapter The PPML Manual, Inria Sophia-Antipolis, 1993.
11. L. Théry, Presentation of Aïoli, see http://www-sop.inria.fr/lemme/Laurent.Thery/aioli.html.

12. XPath World Wide Web Consortium. XML Path Language. W3C Recommendation. See http://www.w3.org/TR/xpath
13. XSL World Wide Web Consortium. Extensible Stylesheet Language (XSL). W3C Working Draft. See http://www.w3.org/TR/WD-xsl
14. XSLT World Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation. See http://www.w3.org/TR/xslt