PROVIDING SCALABLE ACCESS TO LARGE XML DOCUMENTS

Arno Puder
San Francisco State University, Computer Science Department, 1600 Holloway Avenue, San Francisco, CA 94132, USA

Keywords: XML, DOM, Working Set.

Abstract: XML documents often tend to be voluminous and accessing them through a DOM (Document Object Model) interface poses particular challenges. All the existing DOM implementations require an XML document to be completely collocated before it can be parsed. This solution does not scale for huge XML documents. In this paper we introduce an architecture, called VDOM (Virtual DOM) that allows scalable access to large XML documents through a DOM interface. In the VDOM architecture, the actively used portions of an XML document are transferred to the application. The application can begin to traverse this portion without requiring that the complete DOM tree is collocated. As the application traverses the DOM tree, portions of the XML document are loaded on-demand. Using the VDOM architecture is transparent to the application which uses a standard DOM interface to access the DOM tree.

1 INTRODUCTION

Nowadays, the eXtensible Markup Language (XML) is largely used in various domains to structure and mark-up data. XML allows to define data with a tree-like extensible data structure using a software and hardware independent language, which led to its success. Numerous tools have been proposed to access XML documents. Standardized APIs such as DOM (Document Object Model (W3C, 2004)) and SAX (Simple API for XML (SAX Project, 2004)) allow applications to access XML documents. Other tools such as XSL, XPath, and XQuery provide powerful means to transform, reference, and query XML documents. These tools help to structure the XML tool-chain into components which ultimately leads to their better reuse (e.g., XPath is used in XSL and XQuery, and DOM is used in some XSL implementations).

Programmatic access to an XML document from a high-level programming language is possible through either a SAX or a DOM interface. Both offer the contents of an XML document through a standard API to an application. The SAX interface allows an application to read the content of an XML document from beginning to end in the same sequence in which it appears in the document. While this is suitable in some contexts, it cannot be used for applications that require random access to the contents of an XML document; in such cases a DOM parser is required. A DOM interface allows to map a whole XML document to a tree-like data structure of a programming language. The application is then free to traverse the contents in any order. There exist many DOM implementations for most programming languages. E.g., JDOM (JDOM, 2004) is one implementation for Java. Many applications using XML documents as well as higher-level XML tools such as some existing XSL implementations are built on top of JDOM.

More and more applications have to deal with XML documents of enormous size. We will introduce one such application in the following section. While a DOM API provides the convenient abstraction for a programmer to access the contents of an XML document, all the existing DOM implementations have the limitation that they need to first load the complete XML document into main memory in order to parse and build up the DOM tree. For huge XML documents this solution does not scale.

This paper introduces the notion of a Virtual DOM (VDOM) architecture. It allows scalable access to an XML document of arbitrary size through a DOM API. The complete XML document does not have to be collocated for building the DOM tree. Nodes of the document are loaded on-demand as the application traverses the data structure which is done transparently to the application. The application is also able

Puder A. (2007).
PROVIDING SCALABLE ACCESS TO LARGE XML DOCUMENTS.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 178-183
DOI: 10.5220/0001270801780183
Copyright © SciTePress

to access the XML document using any standardized DOM API.

This paper is organized as follows: Section 2 first introduces a use case from the geo-sciences to motivate our work. Section 3 introduces our VDOM architecture. Section 4 concludes this paper and presents an outlook to future work.

2 GEO-SCIENCE USE CASE

In this section we present a use case from the geo-sciences to serve as one example of a domain where large XML documents are common place. In a previous project called NetBEAMS (Networked Bay Environmental Assessment Monitoring System (San Francisco State University, 2003)) we have devised an end-to-end infrastructure to connect web browsers to commercially available sensors that measure environmental conditions of the San Francisco Bay (Zambrano and Puder, 2006).

In the course of the NetBEAMS project, we developed the *Sensor Data Markup Language* (SDML) that allows the description of sensor data and its meta-data via XML (W3C, 2006a). The following SDML document shows some sample markup resulting from our NetBEAMS application:

```
<sdml>

  <metadata id="temperatureRTC">
    <metadata id="description">
      Temperature @ Romberg Tiburon Center (RTC)
    </metadata>
    <metadata id="longitude">3.1245</metadata>
    <metadata id="latitude">-122.1341</metadata>
    <metadata id="altitude">4.242</metadata>
    <metadata id="unit">Celsius</metadata>
  </metadata>

  <!-- More meta-data definitions. -->

  <measurement name="temperatureRTC">
    <timeStamp>
      11/18/05 10:45:06 GMT
    </timeStamp>
    <value>15</value>
  </measurement>

  <measurement name="relHumidityRTC">
    <timeStamp>
      11/18/05 10:47:42 GMT
    </timeStamp>
    <value>87</value>
  </measurement>

  <measurement name="temperatureGG">
    <timeStamp>
      11/18/05 10:48:16 GMT
    </timeStamp>
    <value>13</value>
  </measurement>

  <!-- More measurements -->

</sdml>
```

Different measurements are provided in the above sample of the SDML document and they are denoted by the <`measurement`> tag. Each measurement is associated with some meta-data marked up via the <`metadata`> tag. The purpose of the meta-data in SDML is to attach a geographic location, a time-stamp, and an unit to the individual measurements. The XML excerpt above shows the temperature and relative humidity of two different locations in the San Francisco Bay: the Romberg Tiburon Center (RTC) and the Golden Gate Bridge (GG). Their respective meta-data definitions are referenced through identifiers `temperatureRTC`, `temperatureGG`, and `relHumidityRTC`.

One common application in geo-sciences is to correlate individual measurements by a domain specific function. One example is the computation of the probability of fog (Lowe, 1977). This function requires temperature and relative humidity readings from two locations within a certain time window (e.g., temperature and relative humidity readings from different days cannot be combined). The process of the computation is sufficiently complex and goes beyond the capability of the existing implementations of XQuery and XSL and requires a custom-implementation based on a high-level programming language. Different sensors often report their measurements in different units and it is necessary to convert them to a canonical format. The information about the units can be derived from the meta-data specification of the SDML. For that reason, the implementation of the fog-function requires access to different parts of the SDML document during the computation: the meta-data and the measurements. We refer to those parts as *localities*.

Because localities change in an out-of-order sequence during the computation of the fog-function, a DOM interface is preferred to allow random access to the content of a SDML document. There exist different DOM implementation for different programming languages and scientists are free to choose their preferred implementation to access the measurements and meta-data from the SDML document. E.g., the aforementioned JDOM can be used to allow a Java program to access a SDML document, and other scientists may prefer C++ in combination with the DOM implementation called Xerces. While the existence of these various implementations leverages the skillset of developers, all the implementations possess the same drawback: SDML documents can be very voluminous and easily reach several gigabytes in size. Dealing with an XML document of this size does not scale with any of the existing XML implementations.

While the use case described in this section is domain specific, we believe that huge XML documents are also common in other areas. The work described in this paper allows access to a large XML document independent of its size using a familiar DOM API.

3 VDOM ARCHITECTURE

We propose a VDOM architecture that allows scalable access to large XML documents through the DOM interface. The VDOM architecture is based on the client/server model where the required XML documents are at the server side and the applications accessing the documents are at the client side. Our general idea is as follows. The XML document is partitioned into smaller portions that are transferred to the application piece-by-piece. Without the need to transfer the whole XML document, the application can begin to explore the transferred portions of the document through a DOM interface. The VDOM architecture is transparent to the applications that can access the portions using a DOM API of their choice.

In this section we first explain the design goals of the VDOM architecture (Section 3.1), followed by its overall structure (Section 3.2). Section 3.3 describes the portions of a XML document that can be transferred in the VDOM architecture. The protocol to request and transfer these portions is presented in Section 3.4.

3.1 Design Goals

We begin our discussion of the VDOM architecture by formulating our design goals. The purpose of the design goals is to define *what* we want to accomplish with our VDOM architecture, but not *how* to accomplish them. Subsequent sections describing the architecture will focus on how they are achieved.

The following design goals guide the definition of our VDOM architecture:

- Client/Server architecture.
- Read-only access.
- Access transparency.
- De-coupling of client and server side technologies.

The VDOM architecture is based on the client/server model to allow access to XML documents. The server stores the XML document that is accessed by clients. We already established previously that it is not feasible to transfer a complete XML document, but that only actively used portions (the aforementioned localities) are transferred between client and server. At this stage, to limit the complexity of the VDOM architecture, we only allow read-only access to the document in order to avoid synchronization issues between different clients. While read-only access limits the scope of applications using our architecture, we believe that in many cases write-access is not required.

Another design goal is access transparency. The VDOM architecture follows the client/server paradigm in which there are two access points: on the client side, an application will access the VDOM architecture and on the server side we will need to connect to a data-source that stores the XML document. These access points should be designed such that neither client nor server need to be modified in any way in order to work with our VDOM architecture. The applications are free to choose a particular DOM API and they should not notice that the whole XML document might not be loaded. The servers can also store the XML document in various ways such as an XML database or in a regular file. By doing so, we achieve access transparency that will enable existing applications and data-sources to be easily integrated with our architecture.

The final design goal of our VDOM architecture is de-coupling of technologies. While there will be different ways to implement our VDOM architecture, we want to make sure that there is no technology dependency between the client and the server. It should be possible to provide independent implementations of the client and server of the VDOM architecture, each freely choosing their respective technologies such as the programming language. With this design goal we want to achieve an open architecture with independent, interoperable clients and servers.

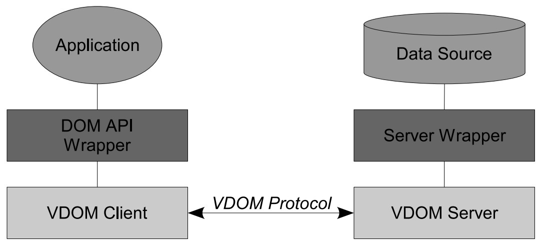

Figure 1: VDOM Architecture.

3.2 Architecture

Figure 1 depicts the VDOM architecture. The architecture is based on the client/server paradigm. The XML document is stored on the server side at some data source (at the right side in Figure 1). We impose no requirements on the data sources (e.g., a source could be a relational database or an XML database). A specific wrapper is required for each kind of data source to allow the data retrieval from the sources. The wrapper also transforms data to XML format in case it is stored using another format at the data server. The VDOM protocol defines the PDUs (Protocol Data Units) exchanged between the VDOM client and the VDOM server; the VDOM protocol will be explained later. The DOM API wrappers on the client side offer the underlying services of the VDOM architecture to the application. The purpose of the DOM API wrappers is to assure that the application developer is unaware of using the VDOM architecture. With the help of these wrappers the application is unaware that only a part of the whole XML document is locally available on the client side. They also allow the application to access the document portions via a familiar API. The API of each specific wrapper is identical to that of a given XML DOM API in the given language.

One possible end-to-end scenario is as follows: the application on the client side uses JDOM to traverse the XML document. But instead of using JDOM directly, the application uses the JDOM wrapper. The wrapper translates the request expressed using JDOM API into the internal request of the VDOM architecture. The internal request is forwarded to the VDOM client. Based on the application's needs, the VDOM client will request an appropriate portions of the XML document from the VDOM server. The VDOM server uses a wrapper to access the data source, to transform the relational data to XML format, and to retrieve the requested portion of the XML document. Upon sending the portion back the VDOM client, the JDOM wrapper returns to the application with the requested data expressed using JDOM.

The application is not aware that only a portion of the requested document is returned. As the application eventually reads nodes from the XML document that lie outside of the returned portion, the VDOM client will automatically retrieve neighboring portions including the newly requested nodes. For now we are only using some static heuristics to determine the size of the XML document portions to be transferred (i.e., the depth and breadth of the portion). We also envision to adapt the size by observing the pattern by which the application accesses the XML document.

3.3 Working Set

Instead of transferring the whole XML document, the VDOM architecture allows to transfer smaller portions of the document that are actively used by the application. We denote the portions of an XML document that can be transferred in our VDOM architecture as *working sets*. The term working set is inspired by a concept from operating systems where it refers to a set of pages in virtual memory actively used by a process (Tanenbaum and Woodhull, 2006). We adopt this principle of a working set and transfer it to XML documents. The working set of an XML document defines those portions of the document used by an application during a certain time interval.

Just like the working set in operating systems defines a certain locality in terms of pages in virtual memory accessed by a process, our definition of a working set for XML documents assumes certain localities in accessing its content. The localities are determined by the required portions of the document during a specific time interval.

Before defining the working sets used in our VDOM architecture, we first give a representation of an XML document as a tree in the mathematical sense consisting of nodes and edges $XML_D = (V, E, n_{Root}, f)$ where:

1. V is a finite set of nodes in the tree representing the XML document; they represent both the elements and the attributes of the document;
2. E represents the edges: every edge $(c, p) \in E$ is an edge from the parent p to the child c;
3. $n_{Root} \in V$ being the root node of the tree;
4. f: is an injective, partial function assigning an order number to all nodes except the root node n_{Root}. If a node has n children, then those children are assigned order numbers in the interval $[0, \ldots, n-1]$.

We define a working set WS of a XML document $XML_D = (V, E, n_{Root}, f)$ as a set of localities of the document $WS = \{L_1, L_2, ..., L_n\}$. Each locality L_i with $0 < i \leq n$ is defined as a subset of the nodes V with the following conditions:

1. $L_i \subseteq V$;
2. Graph (L_i, E) is connected;
3. For every child $c \in V \backslash L_i$ with parent p, $(c, p) \in E$: there does not exist $c_1, c_2 \in L_i$ with $(c_1, p) \in E$ and $(c_2, p) \in E$ with $f(c_1) < f(c) < f(c_2)$.

The conditions (1) and (2) basically state that each locality is a connected sub-tree of the tree corresponding to the XML document. Condition (3) states that all children of a node in a locality must be immediate siblings of each other with respect to the order function f. I.e., the node may have other children that are not part of the locality, but they can only appear “at the border” of the locality. Different localities of the same working set have to be disjoint. For every two localities L_j and L_k of the same working set WS, the intersection of nodes belonging to L_j and L_k has to be empty.

Figure 2 shows an example of an XML document. It also shows a working set containing one single locality whose root node is B. The locality as shown in this example does not include all children of node B, but note that all children belonging to the locality are immediate siblings of each other without a gap (i.e., a node that does not belong to the locality).

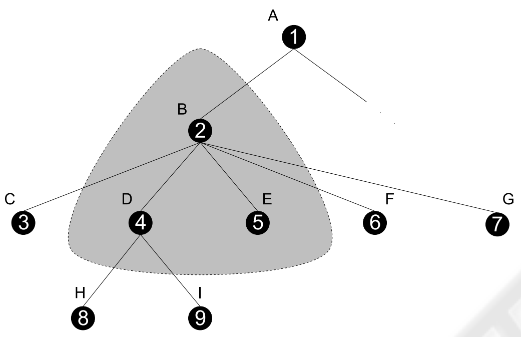

Figure 2: Working set of an XML document with one locality.

3.4 VDOM Protocol

The VDOM protocol defines the PDUs transferred between the VDOM client and the VDOM server shown in Figure 1. The protocol is based on a simple request/response exchange, where the client makes a request and the server responds with the appropriate working set. We first present the marshalling of the working set, i.e., the PDU the server responds to a client’s request. We will explain the parameters of the client request further below.

We use XML notation to represent working sets. Apart from the document content included in the working set, it is necessary also to encode additional information about the environment of the working set inside the whole document to facilitate decisions within the VDOM client during the application’s traversal of the XML document. E.g., it is beneficial to know if there are more children besides the one contained in a locality of the working set. This contextual information is embedded via XML attributes into the working set. For the example shown in Figure 2, the marshalled XML document of the working set is as follows:

```
<vdom:WorkingSet
    xmlns:vdom="http://vdom.org/response/">
  <B vdom:id="2"
     vdom:prev-child="3"
     vdom:num-prev-children="1"
     vdom:next-child="6"
     vdom:num-next-children="2">
      <D vdom:id="4" vdom:num-children="2"/>
      <E vdom:id="5" vdom:num-children="0"/>
  </B>
  <!-- Possibily more localities -->
</vdom:WorkingSet>
```

Every node in an XML document can be uniquely identified via the `vdom:id` attribute. The value of this attribute would usually be an XPath (W3C, 2006b) expression to make use of available XML standards. For the sake of make the example more readable, we use the numbers shown in Figure 2 as the node IDs. E.g., in Figure 2, the node having the ID 2 can be identified using the XPath expression `/A[1]/B[1]`. As can be seen, the additional markup tells the client about the context of the working set in the original XML document. Table 1 summarizes the different attributes available to describe the context.

Table 1: Attributes for describing context of a locality in a working set.

Attribute	**Description**
`id`	XPath expression of the node within the XML document.
`prev-child`	XPath expression of the child bordering before the locality. This attribute is missing if there are no more previous children.
`next-child`	XPath expression of the child bordering after the locality. This attribute is missing if there are no more children following.
`num-prev-children`	Number of children appearing before the locality. This attribute is assigned if and only if the `prev-child` attribute is present.
`num-next-children`	Number of children appearing after the locality. This attribute is assigned if and only if the `next-child` attribute is present.
`num-children`	If none of a node’s children are part of the locality, this attribute specifies the number of children of a node.

The working set is sent by the server in response to a client’s request. The client can request any working set of the XML document. By doing so, the client must provide the root node of every locality of the working set (denoted by the node's XPath expression) as well as the breadth and depth of each locality. The following request shows the markup for requesting the working set highlighted in Figure 2:

```
<VDOMRequest xmlns="http://vdom.org/request/">
  <LocalityRoot id="2">
    <LeftBorder id="3"/>
    <Breadth size="2"/>
  </LocalityRoot>
  <!-- Possibily more locality requests -->
</VDOMRequest>
```

The VDOM protocol also allows the reporting of error conditions (e.g., when the client requests a node with an invalid XPath expression). The VDOM protocol must be mapped to some transport mechanism. Since we use XML for the representation of the VDOM PDUs, Web Services seem to be a natural choice, although other transport mechanisms such as CORBA or plain TCP-connections also are possible.

The working sets are identified depending on the application at the client side and the XML document at the server side. For now we only use some static heuristics to determine the working sets but the VDOM client can also make use of different parameters to infer suitable working sets in order to minimize communication overhead. The schema of an XML document can be used to infer the working sets (e.g., the multiplicity of an element can give an indication to the size of a working set). The application can also be used to infer the size of working sets. E.g., different working sets will be delivered sequentially to the client if it prefers a breadth-first search or if it prefers a depth-first search. The usage history can also be used to help the decision of working sets.

There are two possible strategies for delivering working sets from the VDOM server to the VDOM client. The first consists in delivering working sets only when requested by the VDOM client. Every time the application reaches a portion of the tree that are not locally available, the VDOM client automatically forms a request to the VDOM server for a working set containing the needed portion. The second strategy consists in estimating the needs of the application and delivering some potential working sets before they are required. Among the two strategies, the first one makes requests only when some new portions are required by the application, the response time may increase. The second one estimates the suitable working set. It is more efficient if the estimate happens to be mostly correct; while in the inverse case, pre-fetching several working sets without using them may lead to lower performance.

4 CONCLUSION AND OUTLOOK

In this paper, we introduced the VDOM architecture that allows applications to transparently access large XML documents through a DOM API. In the VDOM architecture, an XML document is partitioned into working sets that are transferred individually to the client. A protocol has been proposed to specify the request and response PDUs of working sets. DOM API wrappers are defined to make the whole architecture transparent to the user application. Server wrappers have also been defined to be able to connect to different kinds of XML document data sources. We are working on a prototype implementation that uses JDOM as the client side DOM API and MySQL on the server side.

Apart from validating our ideas by running some benchmarks, we plan to generalize some internal processes of the VDOM architecture. In particular determining the size of the working set needs to be further investigated. We currently only use static (compile-time) heuristics to determine the size of the requested working set. One obvious extension would be to observe the application's behavior (i.e., the way the application traverses the DOM tree) to adapt the size of the working set at runtime. Other extensions of the work presented in this paper are read/write access to the server, as well as generalizing the client/server model to a peer-to-peer model where the XML document is distributed among different peers.

REFERENCES

JDOM (2004). *Java DOM-API*. http://www.jdom.org/.

Lowe, P. (1977). An approximating polynomial for the computation of saturation vapor pressure. *Journal of Applied Meterology*, 16:100–103.

San Francisco State University (2003). *NetBEAMS - Networked Bay Environmental Assessment Monitoring System*. http://www.netbeams.org/.

SAX Project (2004). *Simple API for XML (SAX)*. http://www.saxproject.org/.

Tanenbaum, A. and Woodhull, A. (2006). *Operating Systems Design and Implementation*. Prentice Hall, third edition.

W3C (2004). *Document Object Model (DOM)*. http://www.w3.org/DOM/.

W3C (2006a). *eXtensible Markup Language (XML)*. http://www.w3.org/XML/.

W3C (2006b). *XML Path Language 2.0*. http://www.w3.org/TR/xpath/.

Zambrano, B. and Puder, A. (2006). A flexible system for real-time oceanographic monitoring. Extended abstract, San Francisco State University.