Distributed and Scalable XML Document Processing Architecture for E-Commerce Systems

David Cheung, S.D. Lee, Thomas Lee, William Song, C.J. Tan

E-Business Technology Institute,
The University of Hong Kong,
Hong Kong

{dcheung, sdlee, ytlee, wsong, ctan}@eti.hku.hk

Abstract

XML has become a very important emerging standard for E-commerce because of its flexibility and universality. Many software designers are actively developing new systems to handle information in XML formats. We propose a generic architecture for processing XML. We have designed an XML processing system using the latest technologies such as XML, XSLT, HTTP and Java Servlets. Our design is very generic, flexible, scalable, extensible and also suitable for distributed network environments. A main application of the architecture and the system is to support data exchange in electronic commerce systems.

1 Introduction

Extensible Markup Language (XML) [1] is a highly flexible format for storing and exchanging data. It has recently received much attention from Web application developers, especially for E-commerce applications [3]. Because XML is a machine-architecture independent format, it facilitates the exchange of data between corporations, which are probably using very different internal formats for the data. As a recommendation of the World-Wide-Web Consortium (W3C) [3], XML is an open standard, which means that any developer can support XML in their products. So, a corporation using XML for data storage and exchange is not locked into a particular software vendor which uses proprietary formats for data storage and interchange. It can easily exchange data with any other corporation using XML. Whom, it can partner with, is no longer selected by the software vendor implementing the proprietary formats.

To enjoy the advantages brought about by XML, a system capable of handling XML files and messages is needed. Currently, many software vendors are actively adding XML capabilities, of varying degrees, to their products. A common approach is to add specially designed modules or enhancements to existing, well-established systems. The program code so developed is usually highly specialized, and hence difficult to reuse in other systems or even other parts of the system. In this paper, we take a different approach. We have designed a generic XML processing architecture. The heart of the architecture is a Document Integrator which determines how input XML documents are processed. The processing is based on high-level scripts written by the application programmers. Based on the scripts, the Integrator processes an XML document by passing it to different Transformation Modules. Each such module is designed for handling a special type of task. It processes the XML passed to it by the Integrator, and then returns a result document, also in XML format, to the Integrator. The Integrator then processes the resulting document, and invokes other Modules as necessary. Finally, the Integrator returns the final result to the caller as an XML document.

Under this architecture, the capabilities of the XML processing system can be extended by designing new Transformation Modules. Existing Modules and the Document Integrator needs no modification. Thus the design is flexible and extensible.

2 Design objectives

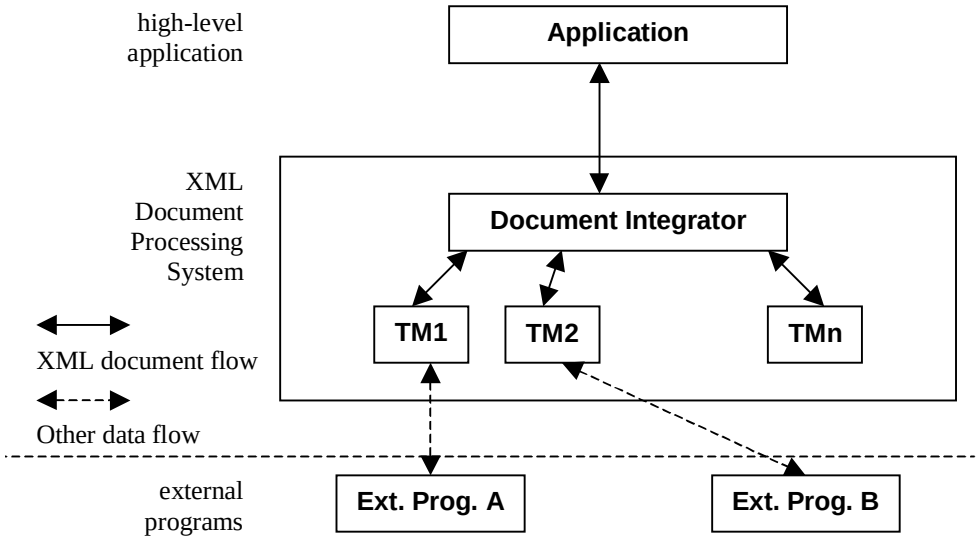
Our design architecture aims at achieving the following objectives:

1. Generality—The system could be easily adapted for the most common XML processing requirements without major modifications.
2. Modularization—Each module is responsible for providing one category of capabilities. This facilitates project management and software maintenance.
3. Distribution—The system is divided into various loosely-coupled modules. The different modules can then be run on different computers to improve processing power.
4. Extensibility—New capabilities can be introduced by adding new modules. The core parts of the system dos not need modification when new capabilities are added.
5. Flexibility—Processing logic is specified in script-like file instead of being hard-coded.
6. Reliability.
7. Robustness.

0-7695-0610-0/00 $10.00 © 2000 IEEE

3 System Architecture

The architecture of our Generic XML Document Processing System is depicted in the following diagram.



The system contains a Document Integrator as well as server Transformation Modules (TM).

3.1 The Document Integrator (DI)

The Document Integrator is the core of this architecture. It is responsible for receiving input XML data from the application program. Upon receiving an XML document, which may come from a disk file or from a network connection, the DI processes the input document according to script files written by the application programmer. According the logic described in the script file, the DI communicates with various Transformation Modules (TM), passing to them appropriate XML documents. The documents returned from the TM's are also XML documents. The DI may store them temporarily for further processing. It may pass these temporary XML files to other TM's if necessary. After collecting all the results from the TM's, the DI combines the results by means of XSLT (see below) and then returns the final result to the application program.

The major role of DI is to act as a bridge between the application program and the TM's, as well as to pass data among the TM's. It thus acts as a document switchbox. The actual processing of XML documents are delegated to the various TM's. However, the DI has to be able to massage the data in XML documents in order to pass them among the TM's. This means it has to transform XML documents frequently. Of course, the task of such a transformation could be delegated to an appropriate TM. However, for efficiency, we decide to add this capability to the DI. This is accomplished by including XSLT (see below) in the DI.

3.1.1 XML Transformation (XSLT). XML Stylesheet Language Transformation [2] specifies a stylesheet language, based on XML, which can be used to describe rules for convert one XML document to another. The language is expressive enough for describing all the transformations required in the DI of our system. Therefore, we adopt XSLT in our DI system for manipulating the input and intermediate XML files. Concatenation and merging of several intermediate work files (in XML format) can be achieved by creating a temporary "master" XML file which consists of one root *element*[1] with the intermediate files as child nodes immediately below the root node. This master file can then be manipulated with an XSL engine to produce the merged result, which is a new XML file.

3.2 Transformation Modules (TM)

Each Transformation Module is responsible for handling one category of tasks. A TM typically receives XML documents from the DI and then processes it according to the logic built into the TM (which may be configurable by means of TM specific script programs). The results of the processing are then encapsulated as an XML document, which is returned to the DI. The exact formats of the input and returned XML documents are up to the TM, although they must be applications of XML.

It is up to the system implementers to determine how a TM would process an incoming XML document. Below, we have identified some functionalities frequently needed and suggest how they can be handled using TM's.

[1] Refer to the XML specification[1] for the definition of "*element*".

3.2.1. Database Access TM (DATM). DATM is an instance of TM, which provides access to ordinary relational databases. The input XML document contains information specifying which tables and fields of which database on which database server are to be accessed. In case of database inserts and updates, the input XML also contains the data to be used for these operations. The DATM inserts new records or updates existing records in the database appropriately. The DATM may returns an XML document to the DI to indicate whether the operator was successful, and possibly the cause of error in case of failure. In case of database query, the query is specified in the input XML document. The DATM queries the database server, and returns the query result to the DI, after converting it to some XML format.

It is up to the particular implementation of DATM to design the formats of the input and returned XML documents. For example, it is possible to directly include SQL statements in the input to specify the database operation. Query results can be formatted into XML according to hard-coded logic in the DATM, or according to the specification of DATM-specific script files.

3.2.2. Message Generating TM (MGTM). An MGTM interprets the input XML document as a message. According to the message headers (or other appropriate logic), it sends the message out. It returns an XML document to the DI to indicate whether the message was sent successfully. An MGTM may send out a message via various media, such as e-mail, fax, Usenet newsgroup posting, pager message, a print job or even a mobile phone short message. Since the outgoing message must conform to the message format of the desired medium, an MGTM must be able to convert the incoming XML message to the format of the target medium before sending it out. The method of sending out the message is also dependent on the medium type. Most probably, an MGTM has to send out the message via external programs or external network connects, using external (non-XML) formats.

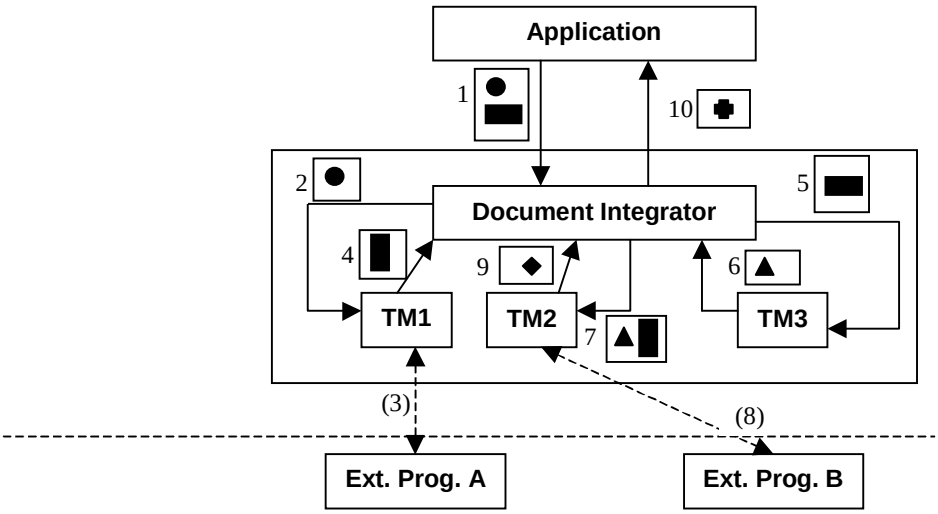
3.2.3. EDI Gateway TM (EDIGTM). Many corporations are using EDI to efficiently exchange messages. An EDIGTM accepts input XML documents as invoice, purchase order, or any kind of EDI message. It then converts the document contents into EDI format and sends it out using EDI channels. This acts as a gateway for *outgoing messages* between our XML Document Processing System and EDI systems.

3.2.4. Logging TM (LogTM). Activities can be logged by implementing a TM, which writes any submitted XML file onto the file system. Then, script files can be modified to select suitable contents from the input or intermediate XML documents and send them to a LogTM.

3.3 Document flow

The actual means by which the DI processes an input XML is driven by the script files written by the application programmer. The processing logic is not hardcoded. The DI only provides the capabilities (with the help of TM's) for the application programmer to manipulate the input XML file and any intermediately generated XML files.

The following diagram illustrates an example on how the DI processes the XML files.



In this example, the DI first receives XML message 1 from the application program. This message contains two data items (shown in the diagram as different solid shapes). DI consults its scripts and decides that it should first send the first data item to TM1, encapsulated in message 2. TM1 receives the message, and interacts (message (3)) with external program A. Then, it returns message 4 to DI, containing data in a new XML document. (As a concrete example, imaging XML message 2 to be containing an SQL query, TM1 be a DATM, interactions (3) be appropriate relational database operations and message 4 be the query results encapsulated as

an XML document.) While TM1 is processing message 2, DI *may* concurrently send document 5 (which contains the second data item from message 1) to TM3 for processing. Thus TM3 can operate in parallel with TM1. TM3 returns document 6 as its result. (A concrete example: TM3 could be a module which validates web "certificate".) Now, DI sends document 7, which is a combination of the data in documents 4 and 6, to TM2 for further processing. TM2 invokes external program B to perform its job, and returns its result as message 9. After receiving message 9, DI further uses its XSLT engine to reformat the XML document to produce document 10, which is returned to the application.

'target' in a `Makefile`) accordingly. The scripting language is illustrated below. It should be noted that the language is an application of XML. Therefore, a script file is itself an XML file.

A script file is a XML document which contains the descriptions on how to make (or generate) a list of XML Documents (i.e. X0…n-1). The following tags are defined:

```
<DOC id="#name" action="action" href="URL">
…
</DOC>
```

This defines a rule section describing how a document is composed. It has the following attributes.

`id="name"`	`name` is a unique identifier of the document. The input document that enters the DC has the name "`input`".
`action="submit" href="URL"`	The composed document will be submitted to `URL`, where a servlet implementing a TM is ready to receive and process the document.
`action="transform" href="URL"`	The composed document will be transformed (internally by DI) by applying the XSL file located by `URL`.
`action="return"`	The composed document will be returned to the caller

It should be reminded that the above is just an example of the workflow of the XML Document Processing System. The actual processing logic is programmable by means of writing script files.

4 Implementation

We are implementing the architecture as described above. We chose Java [4] as the programming language because of its platform-independence and richness of libraries for handling XML and network connections. Both DI and the TM's are implemented as Java Servlets [5]. The communication between the application program and DI is done via the HTTP protocol. Similarly, DI and each TM communicate using the HTTP protocol.

4.1 Implementation of DI

Our implementation of DI is conceptually analogous to the `make` utility (generally available on UNIX platforms). It requires a script file as input. This file describes rules on how to handle the input XML documents, based on the message types. It resembles a `Makefile` for the `make` utility. A `makefile` contains a list of targets. A target may depend on other targets. Therefore, before a target is made, `make` must have made the depending targets of that target. The DI uses the same idea. The script file for the DI instructs the DI how to handle the input XML document by creating a set of intermediate XML documents. To do that, it composes each intermediate document (analogous to

Within a `<DOC>…</DOC>` element, the following tags can be used to specify how the DI would compose the document to be sent (in the case of `action="submit"`) to a TM or transformed using XSLT (in the case of `action="tranform"`).

```
<INCLUDE href="#name"/>

<INLINE>…</INLINE>

<WAITFOR href="#name"/>

<INFO>…</INFO>
```

The `<INCLUDE>` tag specifies an XML segment to be inserted to the temporarily constructed XML file as a subtree of XML elements. If the `href` attribute is specified, the temporary XML document identified by `name` will be inserted. Note that this implies a dependency—the document identified by `name` must have already been composed before this current document (with a `name` specified by the current `<DOC>` element) is constructed. After examining all rules The DI can thus determines the sequence of operations required to construct the final result document. The application programmer thus only needs to declare the set of rules for composing the result and intermediate documents. The DI will figure out what to do.

The `<INLINE>` tag simply inserts the XML segment to be inserted. Note that a property of XML is that elements must be properly nested. So, the segment between `<INLINE>…</INLINE>` must be well-formed XML.

The <WAITFOR> tag specifies that before the current document is made, the DI must have made the document name first. Multiple declaration of this tag is allowed. It is similar to the <INCLUDE> tag except the content of the waited document will not be inserted into the working document.

The <INFO> tag gives human-readable comments, which is not processed by the DI[2].

The following example illustrates how a script file can be written with the above tags to specify how the DI should process the input XML document.

```
<DOC id="descriptions" action="submit"
   href="http://myDATM.eti.hku.hk/servle
   ts/myDATM">
  <INFO>fetch item descriptions from
   database</INFO>
  <INCLUDE doc="#input"/>
  <INLINE>
      <DATABASE name=”catalog”/>
      <TABLE name=”description”/>
  </INLINE>
</DOC>

<DOC id="price" action="submit"
   href="http://myDATM.eti.hku.hk/servle
   ts/myDATM?query=price">
  <INFO>fetch price from
   database</INFO>
  <INCLUDE doc="#input"/>
  <INLINE>
      <DATABASE name=”catalog”/>
      <TABLE name=”description”/>
  </INLINE>
</DOC>

<DOC id=”merged_list”
   transform=”http://xsllib.eti.hku.hk/l
   ib/merge.xsl”>
  <INFO>merge model no., description
   and price information</INFO>
  <INCLUDE doc=”#input”/>
  <INCLUDE doc=”#description”/>
  <INCLUDE doc=”#price”/>
</DOC>

<DOC id="send_result" action="submit"
   href="http://MGTM.eti.hku.hk/servlet/
   MGTM">
  <INFO> </INFO>
  <INCLUDE doc="#merged_list"/>
</DOC>

<DOC id="output" action="return">
  <INFO>reports whether the mail has
   been sent correctly</INFO>
  <INCLUDE doc="#send_mail "/>
</DOC>
```

Suppose the application now sends the DI a query:

```
<PRODUCT_QUERY>
  <MODEL>IBM-300GL</MODEL>
  <MODEL>IBM Intellistation</MODEL>
</PRODUCT_QUERY>
```

Then, DI determines from the root element[3] that this document has a type of “PRODUCT_QUERY”. Then, it will fetch a suitable script file for this document type. Suppose the script file is the one shown above, then according to the rules specified, the DI would perform the following operations:

- Send the input document to the TM servlet at the URL http://myDATM.eti.hku.hk/servlets/myDATM, and store the returned result (an XML document) as document “descriptions”. The TM is supposed to be a DATM, which queries a database and returns the query result after formatting it into XML. The database name and table name are passed by DI to TM as specified in the script file. TM receives the file:

```
<DOC>
  <PRODUCT_QUERY>
    <MODEL>IBM-300GL</MODEL>
    <MODEL>IBM Intellistation</MODEL>
  </PRODUCT_QUERY>
  <DATABASE name=”catalog”/>
  <TABLE name=”description”/>
</DOC>
```

- At the same time, DI sends the input document to the TM servlet (which in this case happens to be the same one as the above) to query another table of (possibly) another database. The returned document is stored and named “price”. Note that this step is independent of step 1, and hence can be performed concurrently with it.
- Next, DI concatenates the input XML document and the result documents from steps 1 and 2 and then applies XSLT to transform it into a new document, named “merged_list”. The XSLT is done according to the XSL file located at URL “http://xsllib.eti.hku.hk/lib/merge.xsl”.
- The merge result (“merge_list”) is then passed to the servlet at URL “http://MGTM.eti.hku.hk/servlet/MGTM” for processing, and the returned result is an XML document named “send_result”. This servlet is supposed to send the document out as e-mail using an external program (or via SMTP).
- Finally, DI returns the document “send_result” to the application program.

4.2 Integration with Web server and WAP server

[2] Except that this comment *may* be inserted into log files for tracing and debugging.

[3] An XML requirement is that each document contains exactly one root element.

There are several possibilities for integrating our XML Document Processing System with web servers providing various web services. Firstly, a web server can be an application using our system. As such, it accepts queries from users, generated with web forms, and then formats the query into an XML document and sends this XML document to the DI. The DI then does the processing according to the pre-written script file. The script file returns the results to the web server, which then presents it to the user as the web query result. Note the document returned to the web server by our DI is an XML file, but the web server may need to reformat it so that it can be displayed properly to the client. Nevertheless, this reformatting may be performed using our DI. We can program our DI so that it applies a suitable XSL file to the result just before it returns it to the web server. This XSL file should transform the XML result into a form suitable for presentation. It should use XHTML for this presentation, as XHTML is just an application of XML. In a similar manner, WAP (Wireless Application Protocol) servers can be integrated with our XML Document Processing System by acting as an application in this architecture. With WAP services, the whole system becomes accessible from mobile phones or any other WAP devices.

The other possibility of integration comes from our use of the HTTP protocol between DI and TM's. In the above, we have been saying that TM's be implemented as Java servlets. But actually, this need not be the case. As long as it speaks the HTTP protocol, understands the XML documents sent from the DI and returns XML documents to the DI, it can function as a TM. So, a TM can also be implemented as a CGI program on any web server. It may also be a standalone program that accepts HTTP connections and processes the files as expected.

5 Discussions

5.1 Distribution

High distribution is a goal of this design. This architecture is highly distributive. Since the DI and TM's communicate through HTTP, there is essentially no restriction of location and platform of the servers where DI and TM's run. However, the servers must support TCP/IP and should not be blocked by any firewall.

5.2 Portability

We highly suggest that the DI and TM's be written in Java, so that they are platform independent. However, if a TM needs to interact with other systems (e.g. database, external servers), then the platform-independence will be determined by these other systems. There may be some cases in which TM's require native code or platform-dependent code to interact with native applications such as EDI gateways. Natively coded TM's will affect portability.

5.3 Performance

Performance is concern in this architecture. The response time may be slow for a query that requires multiple data retrievals from other servers. This architecture attempts to address this issue by allowing parallel data retrieval whenever possible. The DI has intelligence to discover such potentials from the rules of the script file. Since HTTP is not the most efficient communication protocol, performance may be improved by replacing HTTP with Remote Method Invocation (RMI).

6 Conclusions

In this paper, we identified the need to process XML documents in E-commerce systems. We argue that instead of designing dedicated programs to handle each kind of XML document for each specific application, we could design a generic architecture for handling XML documents. The architecture is designed to be generic and flexible. It uses a Document Integrator (DI) to control the process flow, but delegates most capabilities to various Transformation Modules (TM). For efficiency concerns, the DI is designed to have the capabilities of merging and transforming XML documents with XSLT. We have discussed on how to implement the DI and TM's with a simple example. However, since the design is flexible and highly extensible, we believe the architecture is suitable for many E-commerce systems.

7 References

[1] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, **Extensible Markup Language (XML) 1.0**, 10-February-1998, "`http://www.w3.org/TR/1998/REC-xml-19980210`".

[2] James Clark, **XSL Transformations (XSLT) Version 1.0**, 16 November 1999, "`http://www.w3.org/TR/1999/REC-xslt-19991116`".

[3] Dan Connolly, **XML Homepage on W3C website**, April 1997, "`http://www.w3c.org/XML`".

[4] Sun Microsystems, **Java Homepage**, "`http://java.sun.com/products/servlet/`".

[5] Sun Microsystems, **Java Servlet API Homepage**, "`http://java.sun.com/products/servlet/`".

[6] W3C, **World Wide Web Consortium (W3C) Homepage**, "`http://www.w3c.org/`".