Novel Approaches to the Delivery of XML and Schemas

Stephen James Davis
University of Wollongong

Novel Approaches to the Delivery of XML and Schemas

A thesis submitted in fulfilment of the
requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Stephen James Davis
Bachelor of Engineering (Honours Class I)
University of Wollongong, 2001

SCHOOL OF ELECTRICAL, COMPUTER
AND TELECOMMUNICATIONS ENGINEERING
2007

Abstract

Typically XML documents are delivered as whole documents, and the transmission does not consider if all of this data may actually be relevant to the user. This results in inefficiencies in terms of both bandwidth (transferring unnecessary data) and computing resources (extra memory and processing to handle the entire XML document). Through exploitation of XML's tree-like structure, a simple and lightweight protocol is introduced (referred to as RXPP). Designed with mobile devices in mind, RXPP provides users with the ability to navigate and retrieve data from remote documents on a node-by-node or branch-by-branch basis, allowing users to retrieve only fragments of interest. By skipping unwanted XML nodes, this avoids the need to always maintain a full copy of the XML document locally as processing of the document is performed remotely. When only partial views of XML documents are maintained, the processing requirements of mobile devices are less demanding and requires less memory. Furthermore, time and money can be saved when using mobile devices in bandwidth limited environments where data is often charged per kilobyte as only the relevant data is retrieved when the user selects the next node or branch.

Through extension of RXPP, a two-way exchange of XML documents is introduced called RXEP. RXEP allows users to receive XML fragments and also update remote XML documents. In addition to the navigation features of RXPP, RXEP further allows users to construct queries (e.g., using the XPath language), requesting many XML nodes from a remote XML document. In some cases, users can construct well crafted queries to retrieve all the relevant XML fragments using only a single request. RXEP locators are introduced which extend the path features of XPath to the provide precise location of received XML fragments within the clients own local version. RXEP locators provide extra information such as the nodes

absolute location and total number of sibling nodes. RXEP locators thus allow clients to retrieve fragments of XML whilst replicating the exact structure of the original XML document. Through exploitation of RXEP locators and RXEP's two-way exchange, office suites using XML as a document format (such as MS Office and Openoffice), becomes an ideal target for collaborative editing amongst many users. This allows users to download only relevant parts of a document and upload corrections or modifications without the need to upload the entire document.

To further increase the efficiency of RXEP, a binarised (i.e., compressed) version of the protocol is explored. By utilising well established tree-based binarisation techniques significant savings can be achieved through compression of the RXEP structure and requested XML data. A new technique called SDOM is introduced which merges the structural information from XML Schemas with the requested XML document. SDOM allows users to request XML fragments using RXEP techniques where the requested XML data can be compressed on-the-fly using the information contained within SDOM. BinRXEP thus allows users to perform queries or navigation on remote XML documents and receive the results in a compact and compressed form. In many cases, the overhead added by RXEP, is reduced to less than a byte when using binRXEP.

Techniques for the transmission of both XML and XML Schema fragments within a single RXEP packet are proposed. Utilising RXEP, a user can request fragments with a of XML data from a remote document with a further option to request the XML Schema fragment required for validation of that fragment. In this way, the user can avoid retrieving all XML Schemas associated with an XML document, and may only retrieve the relevant XML Schema fragments.

Finally, the collaborative creation of XML Schemas is introduced. Utilising RXEP XML and Schema techniques, users can all contribute to the creation of a schema in realtime, while seeing the progress of other users. This collaborative creation of schemas can lead to quicker creation of XML Schemas. Users may then extend the current set of descriptors or generate new descriptors using ideas from the previous schema updates, thus resulting in a richer set of descriptors.

Statement of Originality

This is to certify that the work described in this thesis is entirely my own, except where due reference is made in the text.

No work in this thesis has been submitted for a degree to any other university or institution.

Signed

Stephen James Davis

18th of February, 2007

Acknowledgments

I would like to thank my supervisor Associate Professor Ian Burnett for his guidance, encouragement and support.

Thanks to my parents, Robert and Evelyn Davis, for their support and sacrifices over the years to get me here. Thanks to Eva Cheng, Chris Davis and Daniel Franklin for proof reading chapters of this thesis.

Finally, I would also like to thank my friends and colleagues of Whisper, SECTE and TITR for their support and enjoyable memories.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation number One
AU	Access Unit
BinRXEP	Binary Remote XML Exchange Protocol
BiM	Binary format for MPEG-7
CODEC	enCOder / DECoder
DI	Digital Item
DID	Digital Item Declaration
DOM	Document Object Model
DTD	Document Type Definition
DIDL	Digital Item Declaration Language
FTP	File Transfer Protocol
FRU	Fragment Request Unit
FUU	Fragment Update Unit
GPRS	General Packet Radio Service
HTTP	HyperText Transport Protocol
IEC	International Electrotechnical Commission
IP	Internet Protocol
ISO	International Organization for Standardization
MIME	Multipurpose Internet Mail Extensions
MPEG	Moving Pictures Expert Group
OWL	Web Ontology Language

PSVI	Post Schema Validation Infoset
RAM	Random Access Memory
RXEP	Remote XML Exchange Protocol
RDF	Resource Description Framework
RXPP	Remote XML Pull Protocol
SAX	Simple API for XML
SDOM	Schema Document Object Model
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TeM	Textual Encoding format for MPEG-7
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLC	Variable Length Coding
UTF-8	UCS Transformation Format 8
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WSDL	Web Services Description Language
XML	eXtensible Markup Language
XOP	XML-Binary Optimized Packaging

Chapter 1

Introduction

In recent years, XML [1] has become the ubiquitous standard for the storage and transmission of structured data over the Internet due to XML's platform independence and interoperability. However, these potentially verbose XML documents require increased transmission times, which especially affect mobile networks.

XML tends to be verbose by nature as it transforms raw data into metadata by representing the raw data as text and creating structure by surrounding the data with text tags. Consequently, XML documents are often verbose and can result in large file sizes. As small portable devices become increasingly popular, more and more users can potentially be exchanging XML. Many of these portable devices operate in limited bandwidth environments where data is charged by the kilobyte transferred.

There have previously been several attempts to allow efficient use with XML documents by [2]:

- Reducing file size - techniques have been developed which attempt to reduce the overall size of the entire XML document through various compression methods. Some compression techniques offer good compression ratios by utilising knowledge from the associated XML Schema [3] (if present). However, as methods attempt to maintain the original structure in the compressed output, there is often a reduction in the overall compression ratio. Conversely, methods that obtain very high compression ratios often lack the ability to be manipulated within the compressed domain; the entire file needs

to be decompressed, modified, then recompressed.

- Improve performance - increasing the performance generally consists of optimising the tools handling the XML data; creating simple, yet fast XML parsing techniques. Many of these techniques (such as SAX [4]), attempt to increase speed by only loading small parts of the XML document at a time, rather than the DOM [5] approach whereby the entire XML is loaded into memory.

Whilst compression techniques often represent a significant reduction of the XML document's filesize, in many cases there is still redundant information contained within the XML document. Some efforts have addressed efficient delivery of XML documents, whereby the data is rearranged to send important information first [6], or is sent as compressed XML fragments [7]. These approaches however, do not give the user any control over what fragments they receive and when they receive them.

This thesis presents novel techniques that allow users or applications to efficiently exchange XML documents. Through the introduction of a new protocol, parts of an XML document can be requested utilising navigation approaches while not retrieving the unwanted parts. This technique is especially advantageous for mobile devices where bandwidth is limited (and often charged by the kilobyte) and are restricted by processing power and memory/storage.

The simple, lightweight protocol is then extended to provide a richer feature-set for users to exchange XML documents. One such feature is the addition of queries; users are able to retrieve many nodes from a remote XML document from a single request. Furthermode, users can utilise the two-way exchange capability to receive and update XML fragments, making collaborative editing between multiple users possible.

This thesis then investigates the binarisation (i.e., compression) of the new XML protocol, utilising existing XML binarisation methods, where both the protocol and the XML data are compressed. Due to the dynamic nature of the protocol, XML fragments can not be binarised in advance. Thus, this thesis presents a new technique which merges both the XML and XML Schema data into a single structure to provide quick access to dynamically binarise selected fragments.

This thesis then presents a technique to exchange XML and XML Schema fragments within the protocol. As users are requesting XML fragments, the XML Schema fragments required to validate those fragments are also transmitted. This allows users to build XML Schema documents locally where only the relevant portions are received. The thesis finally investigates the use of the XML protocol with the XML Schema fragmentation techniques to allow users to create XML Schemas, together, in a collaborative manner.

1.1 Thesis Outline

This thesis is comprised of seven Chapters and are organised as follows:

Chapter 2 provides a literature review of relevant work incorporating XML, XML Schemas, XML Processing, XML compression and delivery;

Chapter 3 details the development of a simple and lightweight XML Protocol (targeted at mobile devices) designed to deliver only the necessary fragments of XML as requested from the XML document;

Chapter 4 investigates extending the simple XML Protocol from Chapter 3 into a two-way exchange protocol. This provides users with the ability to navigate and query remote XML documents, as well as allowing them to make changes on a remote XML document;

Chapter 5 compares different compression techniques applied to XML documents. New extensions were developed to enable MPEG-B BiM to be capable and efficiently binarise (compress) MPEG-21 XML Digital Item Declarations. This Chapter also applies a modified tree-based binarisation technique to the XML protocol to provide reductions in the size of the XML protocol and XML fragments;

Chapter 6 proposes a method whereby only required XML Schema fragments are exchanged with the requested XML fragments to further reduce bandwidth. This Chapter also investigates if this technique is viable for many users all contributing to the creation of an XML Schema; and

Chapter 7 provides a conclusion to the thesis and details future work.

1.2 Contributions

The main contributions as a result of this thesis are:

1. Introduced a simple, light-weight XML protocol to allow users to navigate through remote XML documents, only retrieving fragments of XML which are of interest to the user (Chapter 3);

2. Extension to XPath to allow users to provide additional information about the elements position with respect to all sibling nodes, to allow exact placement when fragments are randomly accessed (Section 4.3.3);

3. Created and contributed to a new part of MPEG-B (Part 2) to provide MPEG-7 TeM [7] and MPEG-B BiM [8] with the ability to request fragments of XML documents. Fragment Request Units (FRU) were created and designed to integrate with the already standardised Fragment Update Units (FUU) from MPEG-7 (Section 4.8);

4. Demonstrated that RXEP can be effectively used in collaborative editing environments, to allow many authors to contribute to a single document (Section 4.7);

5. Experiments comparing the two major XML document structures provided by Openoffice and Microsoft. Experiments revealed that the organisation of the data within an XML office document affects the savings achieved using navigation techniques (Section 4.7);

6. Performed experiments on various XML compression techniques and discovered that schema-based compression techniques perform well on MPEG-21 DIDs (Section 5.2);

7. Demonstrated that higher compression ratios on MPEG-21 DIDs can be achieved with MPEG-B BiM by implementing two new extensions. These extensions utilise information present in the DID to detect embedded XML or base64 content, and choose the appropriate compression technique (Section 5.3);

8. Modification and application of a tree-based XML compression technique to produce a binarised (compressed) Remote XML Exchange Protocol (binRXEP) (Section 5.4);

9. Creation of a new technique called Schema DOM (SDOM) to merge XML Schema information into an XML instance document (Section 5.4.1);

10. Application of the Schema DOM (SDOM) technique combined with a tree-based XML compression technique to allow methods to store and manipulate Schema documents in memory also allowing faster compression times when used with dynamic schemas (Section 5.4.1);

11. Created a novel technique to compress RXEP XPath Locators utilising SDOM and binarisation techniques (Section 5.4.3);

12. Introduced a technique to allow partial XML schema fragmentation and delivery (Section 6.2); and

13. Performed experiments to demonstrate the effectiveness of utilising RXEP to create XML Schema fragments to provide groups of users to create XML Schemas in a collaborative situation (Section 6.4.2);

1.3 Publications

The following publications were a result from this thesis:

1.3.1 Conference Publications

- S.J Davis, I.S. Burnett, "On-Demand Partial Schema Delivery For Multimedia Metadata", *in proc. IEEE International Conference on Multimedia & Expo (ICME) 2006*, Toronto, Canada, July 9 - 12, 2006

- E.C. Cheng, S.J. Davis, I.S. Burnett, "Efficient Delivery of Hierarchically Structured Meeting Audio Metadata with a Bi-Directional XML Protocol", *in proc, International Conference on Computing and Informatics (ICOCI 06)*, Kuala Lumpur, Malaysia, June 6 - 8, 2006.

- S.J. Davis, I.S. Burnett, "Efficient Delivery within the MPEG-21 Framework", *in proc. Automated Production of Cross Media Content for Multi-channel Distribution (AXMEDIS)*, Italy, 30 Nov. - 2 Dec., 2005.

- S.J. Davis, I.S. Burnett, "Collaborative Editing using an XML Protocol", *in proc. IEEE TENCON'05*, Melbourne, 21 - 24 November, 2005.

- S.J. Davis, I.S. Burnett, "Exchanging XML Multimedia Containers Using A Binary XML Protocol", *in proc. IEEE International Conference on Multimedia & Expo (ICME) 2005*, Amsterdam, July 6-8, 2005.

- S.J. Davis, I.S. Burnett, "RXPP: A New Protocol for Efficient XML Exchange", *in proc. Australian Telecommunications and Networking Association Conference (ATNAC '04)*, Sydney, Dec. 2004.

1.3.2 Journal Publications

- I.S. Burnett, S.J. Davis, G.M. Drury,"MPEG-21 Digital Item Declaration and Identification - Principles and Compression", *IEEE Trans. on Multimedia*, Volume 7, Issue 3, June 2005 Page(s):400 - 407.

1.3.3 Patents

- S.J. Davis, I.S. Burnett, "Methods and Systems For Facilitating Access To A Schema", US Provisional Patent, filed September 2005.

- S.J. Davis, I.S. Burnett, "A System, Method and Software for Providing A Computing Device with Data, and an Apparatus, Method and Software for Obtaining Data", US Provisional Patent, filed January 2005.

1.3.4 MPEG

- S.J. Davis and G. Drury, "MPEG B - Part 2: Fragment Request Units", ISO/IEC 23001-2 FCD, 2007

1.3.5 MPEG Contributions

- S.J. Davis, G. Drury, I.S. Burnett, "Contribution on TuC for MPEG-7 Systems", ISO/IEC JTC1/SC29/WG11 MPEG2006/M12846, January 2006.

- S.J. Davis, I.S. Burnett, “Improved and clarified FRU text”, ISO/IEC JTC1/SC29/WG11 MPEG2005/M12573, October 2005.
- S.J. Davis, I.S. Burnett, “FRUs - Suggested Text for Fragment Request Units”, ISO/IEC JTC1/SC29/WG11 MPEG2005/M12404, July 2005.
- S.J. Davis, I.S. Burnett, “FRUs - Requirements for Fragment Request Units and Suggested Text”, ISO/IEC JTC1/SC29/WG11 MPEG2005/M11926, April 2005.
- S.J. Davis, I.S. Burnett, “Proposed extra components for 15938-1 amd/2 and/or 21000-16”, ISO/IEC JTC1/SC29/WG11 MPEG2005/M11612, January 2005.
- S.J. Davis, I.S. Burnett, “Binarisation of MPEG-21 DID’s”, ISO/IEC JTC1/SC29/WG11 MPEG2004/M10644, March 2004.
- S.J. Davis, I.S. Burnett, “Comments on DIA structure compression”, ISO/IEC JTC1/SC29/WG11 MPEG2003/M10181, December 2003.
- S.J. Davis, I.S. Burnett, “Response to MPEG-21 CfP On Digital Item Adaptation”, ISO/IEC JTC1/SC29/WG11 MPEG2002/M8264, July 2002.
- S.J. Davis, I.S. Burnett, “Recommended Improvements and Changes to the WD/AM”, ISO/IEC JTC1/SC29/WG11 MPEG2002/M8566, May 2002.

Chapter 2

Literature Review

2.1 Introduction

This Chapter investigates XML and XML schemas (See Section 2.2), which are becoming increasingly popular for the representation and storage of data. XML tools which are used to parse and validate XML documents are also reviewed. The possibility of large XML documents are examined, in particular, the extensive use of XML within the MPEG-21 standard.

Current XML delivery techniques (see Section 2.3) are then reviewed on their ability to effectively transmit an XML document to a user. Comparison between XML-based delivery techniques and standard file transfer techniques are described.

Finally, this Chapter reviews lossless compression techniques (See Section 2.4) which compares the differences between the three main XML compression techniques to reduce the overall filesize of XML documents: redundancy, schema based/XML aware and hybrid.

2.2 XML and XML Tools

2.2.1 Extensible Markup Language

The eXtensible Markup Language (XML) [1] has gained popularity due to its ability to describe data in a semi-structured, human readable text format. This has some advantages over binary formats, such that users can modify the text using a simple text editor, where as in bi-

nary, the user requires knowledge of how the data is stored, number of bits used etc. Consider for example a company storing personnel records that contain a record id, a persons name, address and date of birth. Figure 2.1 illustrates how a simple textual record might look.

```
Record Id: 1
Name: Allen Person
Address: 123 Fake Street
DoB: 01/02/1900
```

Figure 2.1 Example format for storing the a Personnel record in a plain text format

Figure 2.2 illustrates an example format for representing the personnel as a binary document. The length and id fields would be stored as binary rather than text. For example, one character (8 bits) can represent a positive integer from 0 to 256, which would otherwise require three characters (24 bits) stored as text. The disadvantage with the binary approach is that viewing in a text editor is not always possible as some characters (i.e., the id field) may represent the ASCII code for a non-printable character. Furthermore, without knowing the exact field layout (i.e., can not determine if the length fields are one or two characters) inserting or changing data may corrupt the original data format.

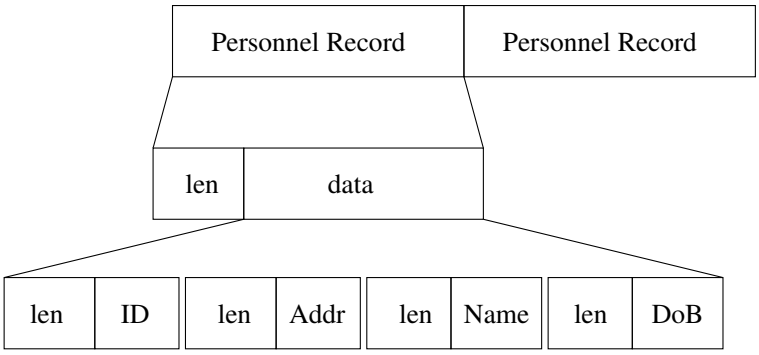

Figure 2.2 Example format for storing the Personnel record in a binary format

XML relies on a more structured textual approach, and the above example could be respresented in XML as illustrated in Figure 2.3. This layout shows that there is a tree-like structure to the data of the record.

XML allows additional information to be added to this structured content providing some indication of what role the content plays (i.e., section heading, footnote, music track etc)

```
<Personnel>
  <Person id="1">
    <Name>Allen Person</Name>
    <Address>123 Fake Street</Address>
    <DoB>01/02/1900</Dob>
  </Person>
</Personnel>
```

Figure 2.3 Example format for storing the Personnel record in a XML format

[9]. This additional information is generally referred to as metadata which is literally, data describing data. For example, by adding extra metadata to the Personnel record in Figure 2.3, the name can be further described using a first name and a last name (this is illustrated in Figure 2.4).

```
<Personnel>
  <Person id="1">
    <Name>
       <First>Allen</First>
       <Last>Person</Last>
    </Name>
    <Address>123 Fake Street</Address>
    <DoB>01/02/1900</Dob>
  </Person>
</Personnel>
```

Figure 2.4 Extra metadata added to the Personnel record from Figure 2.3

XML also allows users to insert pictures and other content within the structured document. For example, to associate a picture to each person in the Personnel records, an additional Photo field may be added as illustrated in Figure 2.5.

Within XML, an element node may contain nested elements (if they appear in between the start and end tags [10]), better known as a 'child' node. These child nodes may further be element nodes also containing child nodes, which creates a tree-like structure [1]. Figure 2.6 illustrates Figure 2.5 as a tree structure.

XML provides the ability to add one or more attributes to an element node; the attributes are specified within the start tag. An attribute allows additional information to be attached to the element tag. For example, to give the Person element of the personnel data an additional 'sta-

```
<Personnel>
  <Person id="1">
    <Name>
       <First>Allen</First>
       <Last>Person</Last>
    </Name>
    <Address>123 Fake Street</Address>
    <DoB>01/02/1900</Dob>
    <Photo>Picture data goes here</Photo>
  </Person>
</Personnel>
```

Figure 2.5 Example of adding a photo to the Personnel record from Figure 2.4

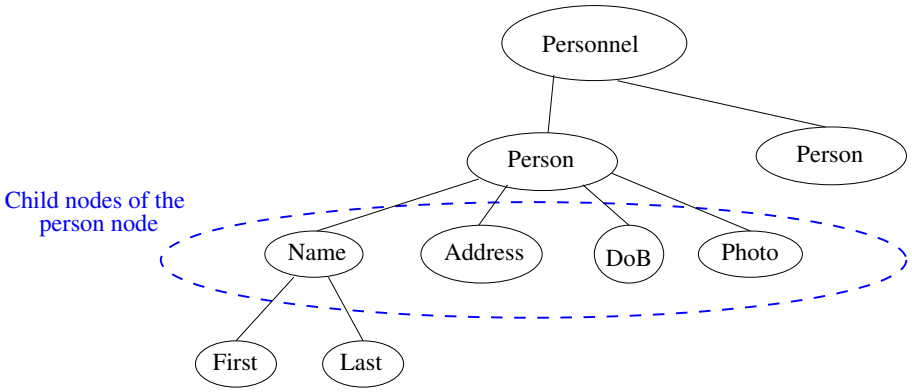

Figure 2.6 The XML from Figure 2.5 illustrated graphically as tree structure, where Elements are represented as ellipses and lines joining ellipses representing parent/child relationship

tus' attribute, the start tag would look like `<Person id="01" status="Employed">`.

Since XML is a text only format, an XML document is unable to directly support embedded binary files. It may be useful to include binary files (e.g., photos or music) as they may be relevant to the structured information in the XML. For example, a persons photo can be included in the record with the persons information as illustrated in the in Figure 2.5. Binary files can be embedded into an XML document if they are first converted into its equivalent ASCII representation (such as using a binary to text technique like base64 [11]).

2.2.2 XML Schema Languages

Although XML documents provide a method to represent structured information, different users may represent the same information in very different structures. To ensure that an XML document follows a particular structure and pre-defined element names, a schema can be used.

A schema is a formal definition of the syntax of an XML-based language. A schema language is a formal language for expressing schemas [12].

Schema processors are the tools which check to ensure that a given XML document is syntactically correct for a defined schema [12]. If this is the case then it is said that the XML document is valid with respect to the schema. For example, if the schema defines that a Person node must have a Name, Address and DoB, then the validation process checks the XML instance to ensure that these rules are met.

If a document is valid to the schema, the schema processor may further normalise the document; normalisation may insert default elements and attributes and remove excess whitespace [12]. This concept of XML validation is illustrated in Figure 2.7.

A schema language should satisfy the following three requirements to be considered useful [12]:

1. The schema language must provide sufficient expressiveness such that most syntactic requirements can be formalised [12];

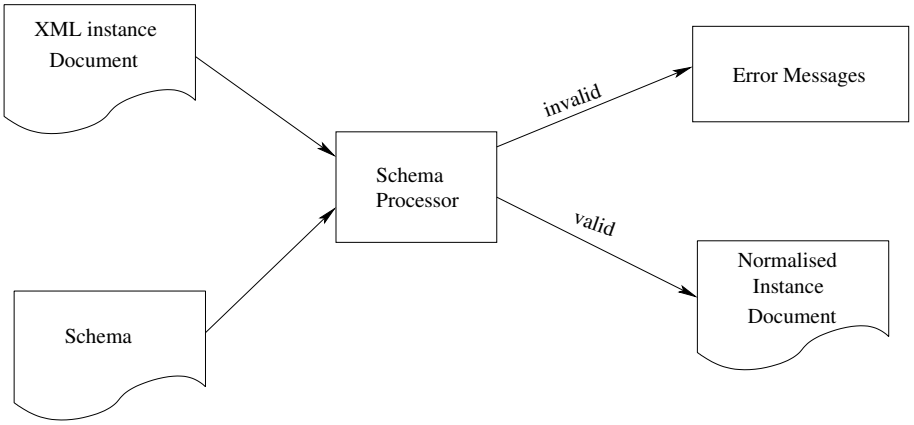

Figure 2.7 Block diagram illustrating the process to determine if an XML document is valid with respect to a schema (adapted from [12])

2. It should be possible to implement an efficient schema processor using the schema language; and

3. The schema language must be simple enough to be understood and used by non-experts.

Typically, most schema languages provide the following syntactic restrictions [3, 13, 14]:

- element names and attribute names which can appear in the document;
- which elements are child elements and their order;
- the number of children allowed;
- the data types for elements and attributes; and
- the default fixed values for elements and attributes;

The most popular schema languages for XML documents are the Document Type Definition (DTD) [1], XML Schema [3], Relax NG [15] and the lesser known Document Structure Declaration (DSD) [16].

2.2.2.1 Document Type Definition

The Document Type Definition (DTD) is a simple schema language and is part of the XML 1.0 [1] specification. An example DTD for the personnel record is shown in Figure 2.8.

A DTD element declaration is constructed as follows:

`<!ELEMENT` *element-name content-model*`>`

Where the element name is the name of the XML element to be defined, and the content-model is one of the following [12]:

- EMPTY - empty contents;
- ANY - any content;
- #PCDATA - character data;
- element name - a defined element elsewhere in the DTD;
- , - concatenation;
- | - union;
- ? - optional;
- * - zero of more repetitions; and
- + - one of more repetitions.

For example, to specify an Address element the declaration would be `<!ELEMENT Address (#PCDATA)>`.

To specify the allowed attributes for an XML element, the declaration is as follows:

`<!ATTLIST` *element-name attribute-name attribute-type default-declaration*`>`

where element-name is a previously defined element to which the attributes apply and the default-declaration is one of: #IMPLIED (optional or no default value), #REQUIRED (this

```
<!--doc:personnel  has 1 or more person elements. -->
<!ELEMENT Personnel (Person)+>
<!--doc:Specify information about a person. -->
<!ELEMENT Person (Name,Address+, DoB,Photo?)>
<!ATTLIST Person id CDATA #REQUIRED>

<!--doc:Specify the persons first and last name.-->
<!ELEMENT Name (First,Last)>
<!--doc:The persons last name.-->
<!ELEMENT Last (#PCDATA)>
<!--doc:The persons first name.-->
<!ELEMENT First (#PCDATA)>
<!--doc:address for this person.-->
<!ELEMENT Address (#PCDATA)>
<!--doc:Date of Birth for this person.-->
<!ELEMENT DoB (#PCDATA)>
<!--doc:Photo for this person.-->
<!ELEMENT Photo (#PCDATA)>
```

Figure 2.8 Example DTD to describe the personnel records

attribute is required to be present), 'value' (the default value) or #FIXED 'value' (can only be this value).

Whilst DTDs are sufficient for simple schema declarations, they do have a number of limitations. The main limitations are [12]:

1. DTDs cannot place constraints on character data. A DTD only allows the contents of an element to be either any character data or none at all;
2. DTD attributes are very limited, i.e., cannot define a type (such as integer);
3. DTDs do not support Namespaces; and
4. The DTD is not expressed as XML, but rather Extended BNF format [12].

2.2.2.2 Relax NG

The Relax NG schema language which is designed to only handle validation (i.e., there no is normalisation process) using a top-down traversal approach [12]. An example Relax NG schema of the personnel records is given in Figure 2.9.

```
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
    <start>
      <element name="Personnel">
        <zeroOrMore>
          <element name="Person">
            <attribute name="id"><text/></attribute>
            <element name="Name">
              <element name="First"><text/></element>
              <element name="Last"><text/></element>
            </element>
            <element name="Address"><text/></element>
            <element name="DoB"><text/></element>
            <optional>
              <element name="Photo"><text/></element>
            </optional>
          </element>
        </zeroOrMore>
      </element>
    </start>
</grammar>
```

Figure 2.9 Example Relax NG schema to describe personnel records

Relax NG relies on matching patterns to ensure that an XML instance is valid to a Relax NG schema. Patterns may match elements, attributes and character data [12]. For example, to match character data the element `text` is used. A Relax NG schema begins with the `grammar` element which contains one `start` element. The start element specifies the pattern to match the XML root node [12]. For example, Figure 2.9 contains a `<element name="Personnel">` after the start element which indicates that the root node for the XML instance is valid to that schema if the node name matches the pattern `Personnel`.

Element declarations in Relax NG may contain the following operators [12]:

- zeroOrMore - zero or any number of the child elements may occur;
- oneOrMore - one or more child nodes must occur;
- optional - zero or one occurrence;
- group - corresponds to concatenation;
- choice - corresponds to union;

- empty - the empty sequence;
- interleave - all possible children that match the subexpressions; and
- mixed - similar to the interleave (above) but additionally containing a text pattern.

2.2.2.3 XML Schema

XML Schema was developed by the W3C to cater for the problems that were present in DTDs. To avoid confusion between an XML schema and the W3C's XML Schema, the latter will be referred to as XML Schema (with the capital 'S').

Some of the goals for XML Schema were that is should be more expressive than DTD, expressed in XML, self describing and simple to implement [12]. An example XML Schema of the personnel record is illustrated in Figure 2.10.

XML Schema relies heavily on assigning all elements a 'type'; these types are as follows [12]:

- Simple type - a set of Unicode text strings [12];
- Complex type - a collection of attributes, sub-elements and character data [12];
- Element declaration - assign an element node of either Simple type or Complex type; and
- Attribute declaration - assign an attribute to a Simple type.

The example XML Schema for the personnel record in Figure 2.10 starts with the `<Schema>` element, as must all valid schema documents. The `xs:` denotes a prefix, and that it belongs to the namespace `http://www.w3.org/2001/XMLSchema` as defined by the `xmlns:xs` attribute. The first element tag defines a root level element with the name Personnel, which is followed by the complexType element (which must be present to be able to define nested children [3]). The next element defines that there is a sequence of the Name, Address, DoB and Photo elements. The `type` attribute on the child elements specifies the datatype of the element data.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
           targetNamespace='test:NS:2006'
           xmlns='test:NS:2006'>
  <xs:element name="Personnel">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="First" type="xs:string"
                          minOccurs="1" maxOccurs="1"/>
              <xs:element name="Last" type="xs:string"
                          minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Address" type="xs:string"
                    minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="DoB" type="xs:string"
                          minOccurs="1" maxOccurs="1"/>
        <xs:element name="Photo" type="xs:base64Binary"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2.10 Example of an XML Schema describing the Personnel Records

In XML Schema, a new element cannot be directly defined as a child of that element. Instead, the first child is a "model group" node which further contains the desired child nodes. Model group nodes [3] defined within XML Schema are as follows:

- Choice - provides a selection of a set of child nodes from which the XML instance can choose;
- Sequence - defines a set of possible child nodes and the order in which children must appear; and
- All - defines that all the child element must appear in this node, but unlike the sequence node, the order of the child nodes in the XML instance is irrelevant.

Model group elements may also be nested within other model group nodes. For example, Figure 2.11 shows a schema with root node F which contains a choice between G or a choice between H or the sequence of I and J.

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
  <xs:element name="F">
    <xs:complexType>
      <xs:choice>
       <xs:element name="G"/>
       <xs:choice>
        <xs:element name="H"/>
        <xs:sequence>
          <xs:element name="I"/>
          <xs:element name="J"/>
        </xs:sequence>
       </xs:choice>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2.11 Example XML Schema containing a nested choices and sequences

Each element described with XML Schema defines the minimum and maximum number of times that element may consecutively appear, usually referred to minOccurs and MaxOccurs respectively [3]. Should minOccurs and maxOccurs be absent in the schema element declaration, it is assumed that minOccurs is 1 and maxOccurs is 1.

Since XML Schemas may import part or all of other XML Schemas, there is the possibility that some elements may have the same name, but in reality have different meanings. To avoid this, a 'namespace' is attached to each schema to allow its definitions to be uniquely identifiable [17]. A namespace identifier [18] simply uses a URI as a unique identifier [18]. A prefix is used to specify the element and its namespace. The prefix name for a given namespace is specified by setting an attribute on an element, i.e., `<element xmlns:prefix="Namespace URI">` [17].

XML Schema is thought to suffer from the following problems [12]:

- XML Schema is regarded as a language which is too complicated; especially for users who are non-experts;
- a specific root element cannot be specified. For example, if there are many elements specified at the root level within the schema, they all can be a root element (i.e., you can not specify only one element to be the root);
- character data within mixed content (i.e., allowing an element as well as character data) cannot be constrained;
- XML Schema is not 100% self-describing, which makes it hard for non-experts; and
- default values for elements can only be character data, not allowing for additional markup.

2.2.2.4 Document Structure Description

Document Structure Description (DSD) 2.0 [16] is a schema language developed by the University of Aarhus and AT&T labs Research. The DSD schema language was created to be 100% self-describing, and when compared to XML Schema, the DSD specification is significantly smaller [12]. DSD was created to contain few and simple language constructs, easy to understand (especially by non-XML experts) and be expressive for practical purposes [12].

DSD is a rule based language, where for each element in the instance document a set of rules are processed [12]. The key goals of DSDs are as follows [12]:

```
<dsd xmlns="http://www.brics.dk/DSD/2.0"
         xmlns:t="http://example.uow.edu.au/personnel"
         root="t:Personnel">
        <if><element name="t:Personnel"/>
            <declare><contents>
              <repeat><element name="t:Person"/>
              </repeat>
            </contents></declare>
        </if>
       <if><element name="t:Person"/>
            <declare>
                <attribute name="id"/>
                <contents>
                    <sequence>
                        <element name="t:Name"/>
                        <element name="t:Address"/>
                        <element name="t:DoB"/>
                    </sequence>
                    <optional>
                        <element name="t:Photo"/>
                    </optional>
                </contents>
            </declare>
        </if>
       <if><element name="t:Name"/>
           <declare><contents>
                 <sequence>
                     <element name="t:First"/>
                     <element name="t:Last"/>
                 </sequence>
            </contents></declare>
       </if>
    <if>
        <or>
            <element name="t:First"/>
            <element name="t:Last"/>
            <element name="t:Address"/>
            <element name="t:DoB"/>
            <element name="t:Photo"/>
        </or>
        <declare><contents>
           <string/>
        </contents></declare>
    </if>
</dsd>
```

Figure 2.12 Example DSD Schema describing schema for the Personnel Records

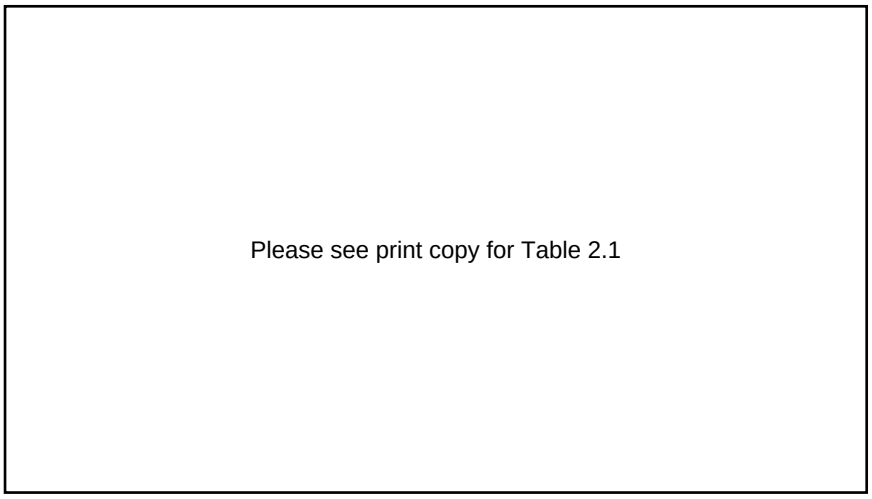

Table 2.1 Boolean expression constructs in DSD2 (From [12])

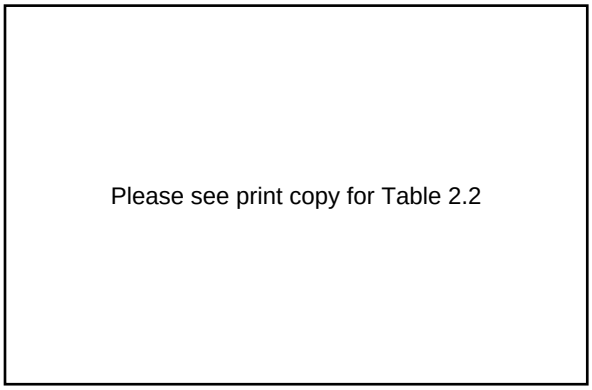

Table 2.2 Regular expression constructs in DSD2 (From [12])

- DSD rules contain *declare* and *require* sections; a *declare* specifies contents and attributes that are allowed for the specified element and a *require* denotes extra restrictions that are applied on contents, attributes and context [12];
- DSDs define attribute values and the contents of an element through the use of regular expressions; and
- Conditions of rules and additional restrictions are applied using boolean logic.

The boolean expressions available in the DSD Schema language are shown in Table 2.1 and the available regular expression constructs are shown in Table 2.2.

The DSD schema shown in Figure 2.12 starts with a `dsd` element to define that the XML is a DSD schema. This schema contains a number of conditional rules (`if` elements) which in this case, only checks the name of an element. The `declare` element declares attributes and contents for that particular node. The `t:Person` declaration defines an attribute `id` for the element and the contents of the element are contained between the `<content>` tags. For the `<t:Person>` the contents is a sequence of Name, Address and Do/b as well as an optional Photo element. The optional elements are contained within `<optional>` tags.

2.2.3 XML Parsing Techniques

XML parsing refers to the method by which the 'parser' application loads and traverses an XML document. An XML parser has many advantages such that it shields users from irrelevant details such as: document encoding (i.e., UTF8), how end of lines are terminated (i.e., line feed, carriage return, or both), byte ordering and how characters are escaped in the plain text [19].

In recent years, several XML parsing techniques have been introduced, each targeting a specific goal (i.e., speed, efficiency etc). These techniques are briefly described as follows:

2.2.3.1 Document Object Model

The Document Object Model (DOM) [5] is a W3C standard which provides a programming-like interface to allow access to objects within an XML document [5]. The DOM structure is a language-neutral interface that provides a tree-like structure [10] where all objects are loaded into memory, allowing fast searching and manipulation (i.e., add, delete nodes) of the document tree [5]. An example XML is given in Figure 2.13 and its corresponding DOM representation is illustrated in Figure 2.14.

DOM loads the XML into a tree of nodes. Most objects within DOM implement the Node interface, such as Element, Attribute and Text. Some of the more common functions from the DOM Node interface is given in Table 2.3. DOM specifies a `NodeList` which is used to hold a list of nodes [5]. For example, child nodes of a parent node would be given in a NodeList [5].

```
<BookRankings>
   <Book id="1">
     <Name>Book 1</Name>
     <Author>A. Author</Author>
     <Ranking>5</Ranking>
   </Book>
   <Book id="2">
     <Name>Book 2</Name>
     <Author>B. Author</Author>
     <Ranking>3</Ranking>
   </Book>
   <Book id="3">
     <Name>Book 3</Name>
     <Author>C. Author</Author>
     <Ranking>4.5</Ranking>
   </Book>
</BookRankings>
```

Figure 2.13 Example XML for three books with a name, author and rank

DOM is well suited for applications where random access on the XML document is required [19], or where many updates of data is required.

The major disadvantage of DOM is the need to store the entire document in memory, this can have adverse affects, especially with large XML documents (i.e., device memory requirements are higher and longer load times).

2.2.3.2 Simple API for XML

The Simple API for XML (SAX) [4] is another XML parsing technique whereby events (e.g., start tag, end tag) are pushed to the programming interface. An important difference to DOM is that the events are unidirectional (i.e., to access previous elements on the XML document requires restarting the parsing process from the beginning of the XML document) [19]. Each SAX event invokes a corresponding 'callback' method that the programmer has written [12]. Some of the SAX events are as follows [12]:

- start - the parser encountered the start of the document;
- end - the parser encountered the end of the document;
- start tag - the parser encountered the start tag of and element;

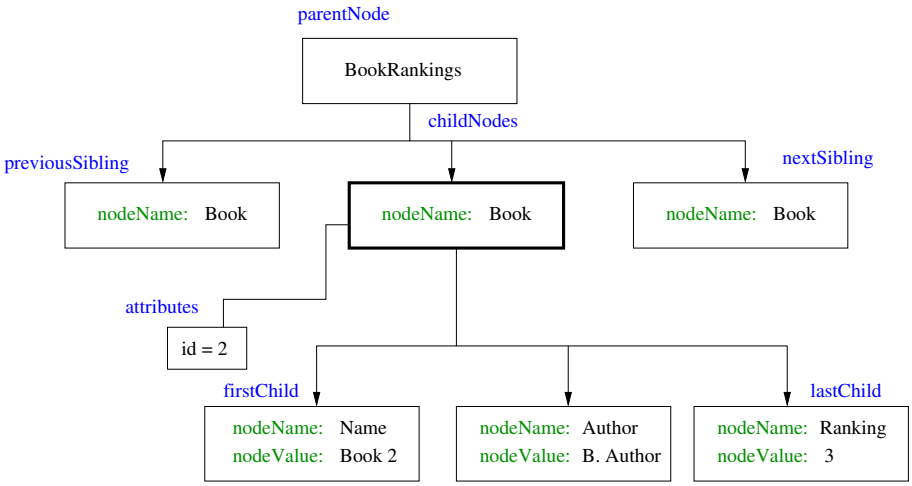

Figure 2.14 A DOM representation of the XML from Figure 2.13 illustrated graphically, where the current node is represented by a bold-lined box

Function	Meaning
`nodeName`	returns the node name as declared in the XML instance
`nodeValue`	returns the value of the node
`nodeType`	returns an unsigned short to indicate node type (i.e., element, text, etc)
`parentNode`	returns the parent node of the current node
`childNodes`	returns a NodeList of the child nodes
`firstChild`	returns the first child node of the current node
`lastChild`	returns the last child node of the current node
`previousSibling`	returns the previous sibling node
`nextSibling`	returns the next sibling node
`attributes`	returns the set of attributes for the current node
`insertBefore`	inserts a new node before the current node
`replaceChild`	replaces a child node with a new node
`removeChild`	removes a child node
`appendChild`	appends a new child node
`hasChildNodes`	returns true or false if the node has child nodes

Table 2.3 Commonly used functions from the DOM 3 Node Interface

```
start document
starting element: BookRankings
-character data, length 3
-starting element: Book
--character data, length 5
--starting element: Name
---character data, length 6
--end element: Name
--character data, length 5
--starting element: Author
---character data, length 9
--end element: Author
--character data, length 5
--starting element: Ranking
---character data, length 1
--end element: Ranking
--character data, length 3
-end element: Book
 ...
end element: BookRankings
end document
```

Figure 2.15 Example SAX events generated from parsing the XML in Figure 2.13, where ellipsis represents removed output

- end tag - the parser encountered the closing tag of an element;
- character data - the parser encountered character data; and
- processing instruction - the parser encountered a processing instruction.

Figure 2.15 demonstrates the SAX events generated from parsing the XML in Figure 2.13.

SAX is well suited for applications where memory is limited and speed is required, especially on very large XML documents. SAX is a good option for local processing where random access is not required.

2.2.3.3 XML-Pull API

The XML-Pull parsing API [20] was developed to combine the advantages of both the DOM and SAX techniques. The idea was to allow very basic parsing commands to 'pull' the data from the XML when the client is ready, opposed to events being 'pushed' to the client as is the case with SAX [20], but without the memory consumption of DOM.

XML-Pull API has one interface with only one key method; the `next()` method. The next method returns one of five events, these are (from [20]):

- Start Document - parser has not yet read any input;
- StartTag - parser is on start tag;
- Text - parser is on element content;
- EndTag - parser is on end tag; and
- EndDocument - document is finished and no more parsing is allowed.

An example output generated from the XML-Pull API (implemented by the XPP3 [21]) is shown in Figure 2.16 from parsing the XML in Figure 2.13.

```
Start document
next() => Start element: BookRankings
next() => Characters:    "\n  "
next() => Start element: Book
next() => Characters:    "\n   "
next() => Start element: Name
next() => Characters:    "Book 1"
next() => End element: Name
next() => Characters:    "\n    "
next() => Start element: Author
next() => Characters:    "A. Author"
next() => End element: Author
next() => Characters:    "\n    "
next() => Start element: Ranking
next() => Characters:    "5"
next() => End element: Ranking
next() => Characters:    "\n   "
next() => End element: Book
next() => Characters:    "\n"
  ...
next() => End element: BookRankings
next() => End Document
```

Figure 2.16 Example XML-Pull events generated from calling the next() method when parsing the XML in Figure 2.13, where ellipsis represents removed output

Popular implementations of XML Pull API are KXML2 [22] - which is effective for ma-

nipulating XML documents on mobile devices, and XPP3 [21] - which has good performance [20].

Unfortunately, current implementations of XMLPull (such as KXML2) require the XML documents to be available locally (requires the entire XML document to be downloaded), or applied to XML documents being sequentially streamed to the local device (if the document is not cached, and elements at the beginning of the document are required, the document needs to be re-streamed).

2.2.4 XPath

The XPath [23] language was developed to allow nodes within an XML document to be individually addressed [23]. XPath 2.0 is also a part of XQuery [24] which was developed in an attempt to provide some of the advantages which are available in the Structured Query Language (SQL) [25].

The XPath language defines a syntax to select nodes, or a set of nodes within an XML document. These nodes may be element, text or attribute nodes. For example, an XPath expression can be constructed to return all elements that match a given expression [26]. XPath contains over 100 built-in functions which are used for string, numeric, date, etc comparisons and conversions [26] (all the XPath functions can be found in [23]).

A sample XML file defining a collection of books is given in Figure 2.17 which will be used throughout this Section for the XPath examples.

There are two modes of addressing within XPath. The first method is absolute addressing (i.e., starting from the root node, which is the first topmost element in the XML document), indicated by starting the XPath expression with a forward slash (`/`) [23]. The second method of addressing is relative addressing, which assumes that the starting position is from the 'current' selected node as indicated by a period (`.`) [23]. For example, to select the `Collection` element from the XML in Figure 2.17 using an absolute address, the XPath expression would be `/Collection`. Furthermore, if the current position in the XML tree is pointing to the Book node, then selection of the Title node of that book can be done with a relative address of `./Title`.

```
<Collection>
  <Book id="B1">
    <Description>This is a Book
    </Description>
    <Title>myBook</Title>
    <Author>A. Author</Author>
  </Book>
  <Book id="B2">
    <Description>This is another book
    </Description>
    <Title>myBook2</Title>
    <Author>A. Author</Author>
  </Book>
</Collection>
```

Figure 2.17 Example XML document defining a collection of books

```
<Collection>
  <Book id="B1">
    <Description>This is a Book
    </Description>
    <Title>myBook</Title>
    <Author>A. Author</Author>
  </Book>
  <Book id="B2">
    <Description>This is another book
    </Description>
    <Title>myBook2</Title>
    <Author>A. Author</Author>
  </Book>
</Collection>
```

Figure 2.18 Nodes as indicated in **bold** are the same nodes selected using two different XPath expressions `//Book` and `/Collection//Book`

A double forward slash (`//`) is used when a selection searching all descendants from the current node is required. For example, the XPath expression `//Book` can be used to select all descendant Book elements from the root node as illustrated in Figure 2.18.

To select the *n*th element within an XML document, an index can be used. Indexing in XPath is done by surrounding the index number with square brackets (`[ ]`). For example, the XPath expression `/Collection/Book[2]` will select the second Book node of the Collection node. Indexing can also be performed by adding an expression inside the [ ], where the nodes matched will correspond to the nodes where the expression evaluates to 'true'. For example, the XPath expression:

```
/Collection/Book[Title="myBook"]
```

will select the Book node which has the value myBook for the Title node. From the XML in Figure 2.17, the following XML fragment evaluates as true for the previous XPath expression:

```
<Book id="b1">
  <Title>myBook</Title>
```

Attributes declared in XML elements can be referenced in XPath using the `@` symbol followed by the attribute name. For example, to select all Book nodes that have the attribute `id` equal to B2 (i.e., matching the XML tag `<Book id="B2">`, the corresponding XPath expression would be `//Book[@id="B2"]`.

2.2.5 MPEG-21

MPEG-21 is a ISO/IEC standard which relies heavily on XML and XML Schema. The main focus of MPEG-21 is to provide a solution to the 'big picture' of multimedia [27]. Prior to MPEG-21, there was no technology which integrated both the infrastructure for delivery and the consumption of multimedia content [27]. Thus, the role of MPEG-21 is to give users a framework within which they can create, consume and deliver multimedia content.

MPEG-21 part two focuses primarily on the Digital Item (DI) [28], which utilises XML and XML Schema. The Digital Item is the fundamental unit of distribution and transaction

within the MPEG-21 framework [28]. A Digital Item is a structured digital object, which is a collection of multimedia resources (content) held together according to relationships determined by the author of the DI [28]. The DI is content agnostic such that it can handle any type or format [29]. For example, a Digital Item of a music album may contain a number of video clips, music tracks, pictures and external links, accompanied with relevant metadata (e.g., MPEG-7).

Figure 2.19 shows part of a Digital Item Declaration representing a list of a classical music album[1], however due to the length of the DID, only the parts required to represent the first song are displayed. The partial DID in Figure 2.19 allows a user to choose between listening to a preview or the full version of each audio track.

For any given DI, the constituent resources may have accompanying descriptive metadata and rights expressions [29]. It is thus vital that the relationships and metadata in the DI can be consistently expressed. This is the role of the Digital Item Declaration (DID) [28], which describes a set of abstract terms and concepts that form a model for defining Digital Items [28]. The fundamental units of the model are: Containers, Items and Components [30]; an example is illustrated in Figure 2.20. Briefly, the core elements are [28]:

- Container - a structure that allows items and/or containers to be logically grouped together. Descriptors may be used in conjunction with a Container to provide descriptive information or a 'label';
- Items - a grouping of further sub-items and/or components that are bound to relevant descriptors [30]. Items can be customised (or configured) by the user through Choices contained within the Item; and
- Components - the binding of a resource to all of its relevant descriptors. Such descriptors will typically contain control or structural information about the resource.

In terms of implementation, the DID is defined in the form of an XML Schema, and additionally, also bound to the rules defined in the Digital Item Declaration Language (DIDL) [28].

[1]The full DID can be found at http://www.enikos.com/mpeg21_content.shtml

```
<DIDL xmlns="urn:mpeg:mpeg21:2002:01-DIDL-NS"
                              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:mpeg:mpeg21:2002:01-DIDL-NS didl.xsd">
    <Declarations>
        <Descriptor id="qlogo2">
            <Component>
                <Resource mimeType="image/gif" ref="images/qlogo2.gif"/>
            </Component>
        </Descriptor>
        <Descriptor id="nav1_1">
            <Component>
                <Resource mimeType="image/gif" ref="images/nav1_1.gif"/>
            </Component>
        </Descriptor>
        <Component id="full_at_qmusic">
            <Resource mimeType="text/html" ref="full/full.html"/>
        </Component>
    </Declarations>
    <Item>
        <Descriptor>
            <Statement mimeType="text/plain">Music from q-music</Statement>
        </Descriptor>
        <Descriptor>
            <Reference target="#nav1_1"/>
        </Descriptor>
        <Descriptor>
            <Statement mimeType="text/plain">
               A small selection of tracks available at http://www.q-music.co.uk
            </Statement>
        </Descriptor>
        <Descriptor>
            <Reference target="#qlogo2"/>
        </Descriptor>
        <Descriptor>
            <Statement mimeType="text/plain">(classical selection)</Statement>
        </Descriptor>
        <Choice choice_id="full_or_preview" default="play_preview"
                                              maxSelections="1" minSelections="1">
            <Descriptor>
               <Statement mimeType="text/plain">
                       Play full track or preview
               </Statement>
            </Descriptor>
            <Selection select_id="play_preview">
                <Descriptor>
                    <Statement mimeType="text/plain">Preview</Statement>
                </Descriptor>
            </Selection>
            <Selection select_id="play_full">
                <Descriptor>
                    <Statement mimeType="text/plain">Full</Statement>
                </Descriptor>
            </Selection>
        </Choice>
        <Item>
            <Descriptor>
                <Statement mimeType="text/plain">
                    Beethoven's Fifth Symphony - 1st Movement
                </Statement>
            </Descriptor>
            <Descriptor>
                <Condition require="play_preview"/>
                <Statement mimeType="text/plain">(preview)</Statement>
            </Descriptor>
```

```
            <Descriptor>
                <Reference target="#qlogo2"/>
            </Descriptor>
            <Component>
                <Condition require="play_preview"/>
                <Descriptor>
                    <Statement mimeType="text/plain">
                       Beethoven's 5th Symphony - 1st Movement
                    </Statement>
                </Descriptor>
                <Descriptor>
                    <Statement mimeType="text/plain">(preview)</Statement>
                </Descriptor>
                <Resource mimeType="audio/mpeg"
                          ref="preview/beethovens_5th_1st_mnt.mp3"/>
            </Component>
            <Component>
                <Condition require="play_full"/>
                <Descriptor>
                    <Statement mimeType="text/plain">
                         Beethoven's 5th Symphony - 1st Movement
                    </Statement>
                </Descriptor>
                <Reference target="#full_at_qmusic"/>
            </Component>
        </Item>
  ...
    </Item>
</DIDL>
```

Figure 2.19 Partial view of a Digital Item Declaration representing a selection of songs of a classical music album. The DID provides the user with the choice of listening to the preview track or the full track

The DIDL is required as XML Schema is not expressive enough to fully describe the DID (i.e., different validation rules depending on the value of attributes), and the DIDL provides additional rules after successful XML Schema validation (to check these rules, additional software (other than the XML parser) is required). For example, an XML document can be conformant to the DID XML Schema, but be invalid according to the rules of the DIDL.

The DID provides a versatile method of describing multimedia content and also different choices of content, thus to better match terminal capabilities. For example, a music DID may contain several audio tracks, with choices for different audio coding techniques and bitrates (i.e., choices for portable devices up to home entertainment systems).

A consequence of the DIDs versatility will likely be long DIDs. For example, from the DID in Figure 2.19, some additional MPEG-7 descriptors can be added to the Statement of a music track as illustrated in Figure 2.21. This illustrates the size increase for adding some simple MPEG-7 descriptors to just one track.

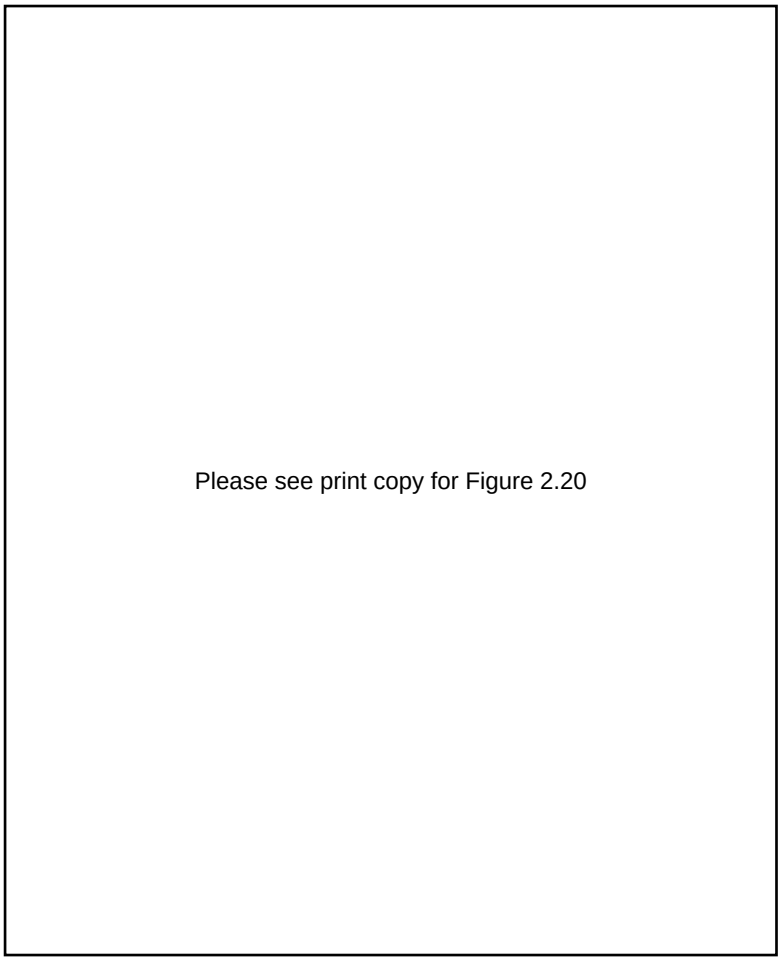

Figure 2.20 Relationship of the principle elements within the Digital Item Declaration Model [30]

```
...
   <Statement mimeType="text/xml">
    <mpeg7:Mpeg7>
     <mpeg7:Description xsi:type="CreationDescriptionType">
      <mpeg7:CreationInformation id="beethoven5th">
        <mpeg7:Creation>
         <mpeg7:Title type="songTitle">
            Beethoven's 5th Symphony - 1st Movement
         </mpeg7:Title>
         <mpeg7:Title type="albumTitle">Classical Favs
         </mpeg7:Title>
         <mpeg7:Creator>
          <mpeg7:Role href="urn:mpeg:RoleCS:2001:PERFORMER"/>
          <mpeg7:Agent xsi:type="PersonType">
            <mpeg7:Name>
              <mpeg7:FamilyName>Beethoven</mpeg7:FamilyName>
              <mpeg7:GivenName>Ludwig van</mpeg7:GivenName>
            </mpeg7:Name>
          </mpeg7:Agent>
          </mpeg7:Creator>
            <mpeg7:CreationCoordinates>
                <mpeg7:Date>
                  <mpeg7:TimePoint>1804</mpeg7:TimePoint>
                </mpeg7:Date>
              </mpeg7:CreationCoordinates>
            </mpeg7:Creation>
            <mpeg7:Classification>
              <mpeg7:Genre href="urn:id3:cs:ID3genreCS:v1:X">
               <mpeg7:Name>Classical</mpeg7:Name>
              </mpeg7:Genre>
            </mpeg7:Classification>
         </mpeg7:CreationInformation>
       </mpeg7:Description>
     </mpeg7:Mpeg7>
    </Statement>
...
```

Figure 2.21 Example of adding additional MPEG-7 descriptors to a music track from the classical music DID from Figure 2.19

Descriptors catering for different terminals may also be included in the same DID (i.e., specifying several formats and resolutions of the media files for different devices and screen sizes), further adding to the overall size of the DID.

These long DIDs may potentially affect performance within the MPEG-21 system, especially where users are in bandwidth limited environments. Furthermore, DIDs contain options providing choices to be made for specific applications (i.e., descriptors for an MP3 encoded at several bitrates), however, in a typical situation a user may only use one of these choices.

Thus, DIDs are a clear target for XML compression algorithms and efficient XML delivery mechanisms.

2.3 XML Transmission Techniques

Whilst XML and its self-describing textual nature has many benefits such as representing metadata in a structured tree-like format, it does however create long verbose documents [17]. These verbose documents are due to the numerous tags to create the structure and often additional descriptors are added to the raw data. Consequently, this verbosity results in larger file sizes as compared to the original raw data which additionally creates processing delays during parsing (i.e., more time is required to parse the extra tags), requires additional space and increases transmission times. However, there are solutions such as [31] that rewrite the XML or Schema before delivery in an attempt to reduce the overheads created by the XML itself. This thesis will investigate technologies and techniques pertaining to the delivery of required or relevant fragments of an XML document in an attempt to reduce the overheads of the XML document delivery without altering the structure or original data.

There are currently many protocols that can be used for transmitting XML data. Some of these are briefly investigated.

2.3.1 XML Fragment Interchange

XML Fragment Interchange [32] is a W3C candidate recommendation that was put forward in 2001. XML Fragment Interchange proposes a method for viewing or editing one or more entities of an XML document without the need to parse the entire document [32, 33]. XML Fragment Interchange defines that each fragment has “context information”, which is the set of information that allows it to be successfully parsed, edited or viewed [32]. The fragment context information also specifies the notation of how the content information should be described [32, 34].

An example XML document for a collection of books is in Figure 2.22 and an example XML Fragment Interchange document sending the second book item is given in Figure 2.23

```
<Collection>
  <Book id="B1">
    <Description>This is a Book
    </Description>
    <Title>myBook</Title>
    <Author>A. Author</Author>
  </Book>
  <Book id="B2">
    <Description>This is another book
    </Description>
    <Title>myBook2</Title>
    <Author>A. Author</Author>
  </Book>
</Collection>
```

Figure 2.22 Example XML document defining a collection of books

```
<p:package
     xmlns:p="http://www.w3.org/2001/02/xml-package">
  <f:fcs
     xmlns:f="http://www.w3.org/2001/02/xl-fragment">
    <Collection>
      <Book/>
    <p:fragbody/>
    </Collection>
   </f:fcs>

  <p:body>
    <Book id="B2">
      <Description>This is another book
      </Description>
      <Title>myBook2</Title>
      <Author>A. Author</Author>
    </Book>
  </p:body>
</p:package>
```

Figure 2.23 Example Fragment using XML Fragment Interchange

XML Fragment Interchange does not specify any method for defining or creating user determined fragments and there is no description for the packaging and delivery mechanisms required for successful interchange of XML fragments.

2.3.2 XML-Binary Optimized Packaging

XML-binary Optimized Packaging (XOP) [35] is another specification from the W3C and proposes a method for containment of serialised XML inside an extensible packaging format (such as MIME Multipart/Related [36]) and replacing selected portions of its content with links to locations within the package, re-encoding the selected removed portions where applicable [35].

Since XML is a text only format, to embed binary data into an XML document requires the binary data to be converted into its ASCII equivalent representation (such as base64 [11]). Base64 encoding represents every three bytes of data into four bytes of ASCII text. The base64 conversion thus results in a file size increase of 33% compared to the original representation.

After XOP removes the re-encoded sections from the XML document, these sections are then replaced within a special element which links to the packaged data using standard URIs [35]. An example of XOP architecture is illustrated in Figure 2.24.

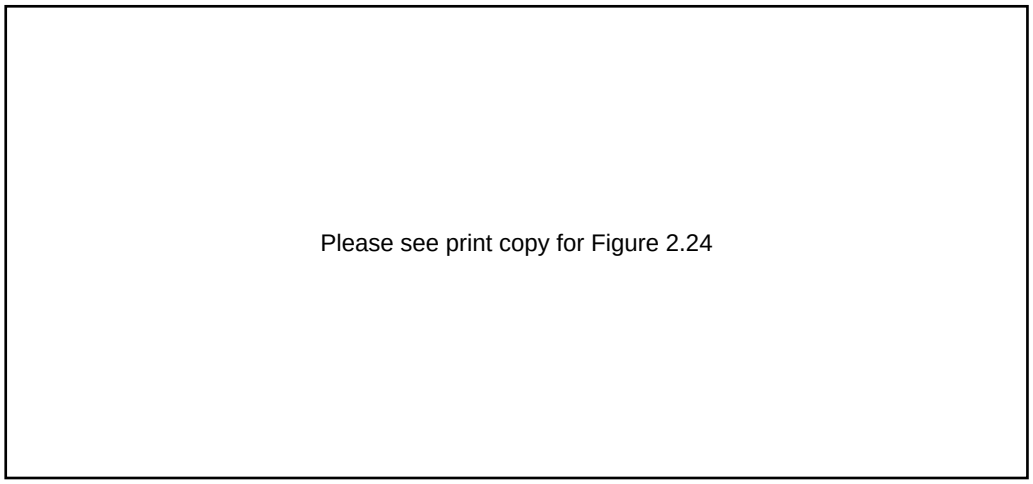

Figure 2.24 Architecture of the XOP framework [35]

The advantage of delivering XML using XOP is the possible reduction in transfer times since the base64 [11] data is now represented in its true binary form, and thus removing the 33% increase introduced by base64. For example, Figure 2.25 illustrates a very simple XML file, which contains an embedded photo contained within a Resource element (for readability the embedded photo is a string representing the base64 encoded photo).

```
<Album
 xmlns:xop="http://www.w3.org/2003/12/xop/include"
 xmlns:xop-mime="http://www.w3.org/2003/12/xop/mime"
 xmlns="myalbum:schema:2005">
  <Set id="2005">
    <Set id="Holiday">
       <Photo>
   <Description>A picture of the beach</Description>
   <Resource xop-mime:content-type="image/jpeg">
           /aWKKapGGyQ=
   </Resource>
       </Photo>
    </Set>
  </Set>
</Album>
```

Figure 2.25 Example XML document with embedded photo

After XOP processing has occurred on the XML in Figure 2.25, the base64 content is replaced with an xop:Include element and the base64 data is converted back into binary and stored within the XOP container. An example of the XML serialised as an XOP package is given in Figure 2.26

Whilst XOP provides reduced file size, and thus transmission times, unfortunately the optimisation of data types is limited to only elements being represented as base64 (as used by XML to represent embedded binary).

2.3.3 XStream

XStream [33, 34] is another XML delivery technique which differs to XOP in that is delivers an XML document in fragments rather than the entire XML document at once. XStream creates the XML fragments based on the XML document semantics, utilising the XMLs structure and organisation [33] (determined from the corresponding DTD). After the fragmenta-

```
MIME-Version: 1.0
Content-Type: Multipart/Related;boundary=MIME_boundary;
              type=text/xml;start=<example.xml>
Content-Description: XML with jpeg image

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <example.xml>

<Album
 xmlns:xop="http://www.w3.org/2003/12/xop/include"
 xmlns:xop-mime="http://www.w3.org/2003/12/xop/mime"
 xmlns="myalbum:schema:2005">
  <Set id="2005">
    <Set id="Holiday">
      <Photo>
   <Description>A picture of the beach</Description>
   <Resource xop-mime:content-type="image/jpeg">
    <xop:Include href="http://myserver.org/beach.jpg"/>
   </Resource>
      </Photo>
    </Set>
  </Set>
</Album>

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <http://myserver.org/beach.jpg>

<< binary data for the jpeg image here>>

--MIME_boundary
```

Figure 2.26 Example XML serialised as an XOP Package

tion phase, XStream packages these XML fragments into Xstream Data Units (XDU) [34] which are then delivered to the client using the XStream Transfer Protocol (XSTP) [34].

The client receives the XDUs and reconstructs the XML by assembling the fragments back together. Any XDUs that are received out-of-order and additionally can not be appended to the XML document (i.e., the parent fragment is missing), are cached until there is enough information to add them to the XML document [33]. The architecture of XStream is further illustrated in Figure 2.27

Although XStream defines a fragmentation and delivery strategy independent of the DTD (i.e., the DTD is not required to re-assemble the fragments on the receiver), a disadvantage is that XStream still streams the entire document to the receiver. XStream does not provide the receiver with the ability to request or dynamically tailor the fragments (XDUs), or specify the order of transmission.

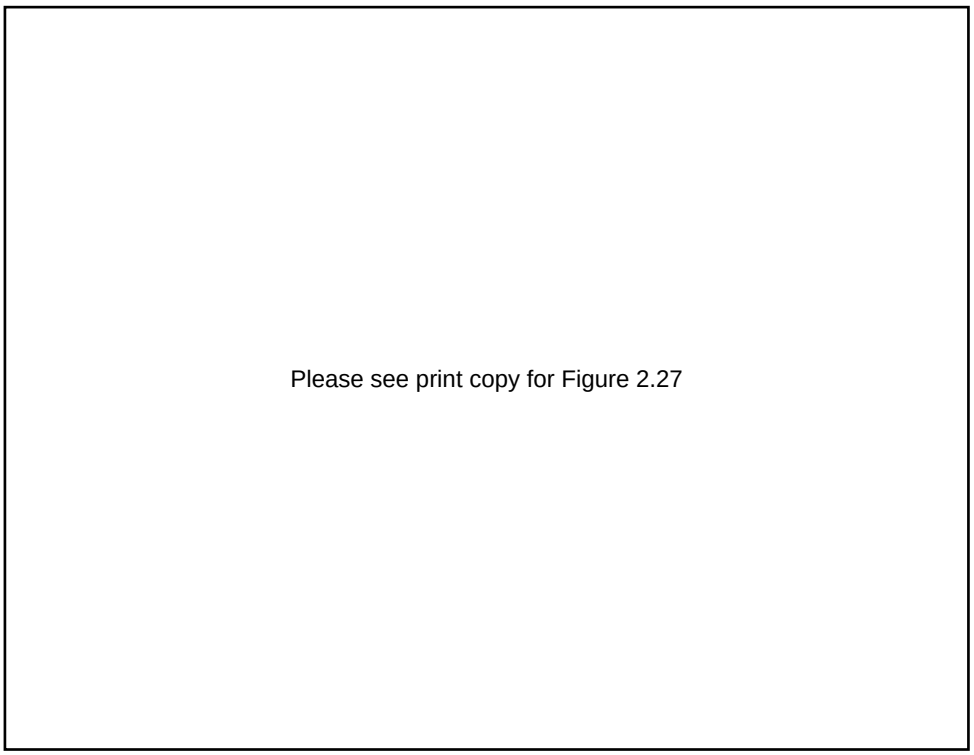

Figure 2.27 Interaction diagram showing the operational flow of XStream (from [33]

2.3.4 HyperText Transfer Protocol

The HyperText Transfer Protocol (HTTP) [37] is probably the most popular protocol used for the transmission of documents on the Internet. HTTP is a stateless protocol, which is an approach whereby each command is executed independently, without any knowledge of commands previously issued [37]. HTTP is a simple, oneway protocol that can only be used to retrieve files but the protocol itself is not designed to transfer files to the server [37]. HTTP requests and responses are specified as a textual format and are composed as specified in Figure 2.28 and Figure 2.29 respectively.

For example, to request the file /test.txt from the server tone.elec.uow.edu.au, a connection would be established on port 80, and the request as illustrated in Figure 2.30 would be transmitted. The response of the request in Figure 2.30 as illustrated in Figure 2.31, is transmitted to the client.

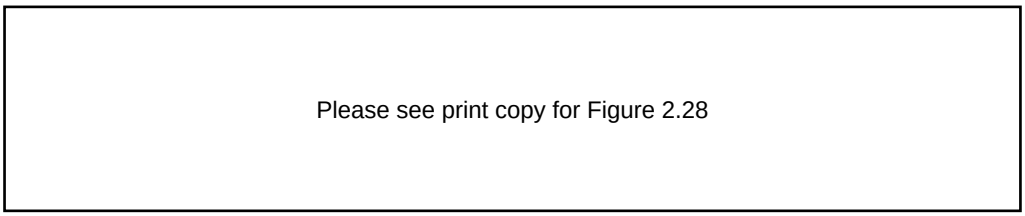

Figure 2.28 Format of the HTTP 1.1 Request format (from [37])

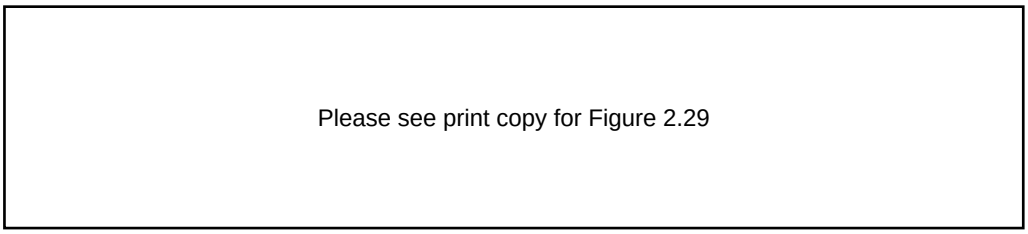

Figure 2.29 Format of an HTTP 1.1 Response (from [37])

```
GET /test.txt HTTP/1.1
Host: tone.elec.uow.edu.au
```

Figure 2.30 Example of an HTTP 1.1 Request

```
HTTP/1.1 200 OK
Date: Wed, 19 Jul 2006 13:17:35 GMT
Server: Apache/2.0.55 (Debian)...
Last-Modified: Wed, 19 Jul 2006 13:16:15 GMT
ETag: "c00001-e-116625c0"
Accept-Ranges: bytes
Content-Length: 14
Content-Type: text/plain

This is a Test
```

Figure 2.31 Example of an HTTP 1.1 Response to the Request in Figure 2.30

Although HTTP is capable of transmitting XML documents, HTTP does not understand the structure of the XML documents and they are simply delivered in its entirety. For example, a user might only wanted one small part of a potentially large XML document but had to receive the entire document.

2.3.5 File Transfer Protocol

The File Transfer Protocol (FTP) [38] is a two-way protocol where clients can upload and download files from the server, as long as the user has appropriate permissions. FTP defines a number of commands which can be issued by the client, some of the most commonly used commands include [38]: Get (to retrieve a file), Put (to upload a file), CD (change directory) and Dir (obtain a directory listing). FTP can transfer both text and binary files using the ASCII and binary ftp modes, respectively.

FTP requires at least two connections for successful transmission: one for sending the control information, and one for sending the actual data. The FTP model is illustrated in Figure 2.32.

Although FTP is capable of transmitting XML documents it, like HTTP, just delivers the entire file as is. For example, a user might only wanted one small part of a potentially large XML document but had to receive the entire document. FTP may transmit XML document slightly faster than HTTP since it has the ASCII mode of transfer.

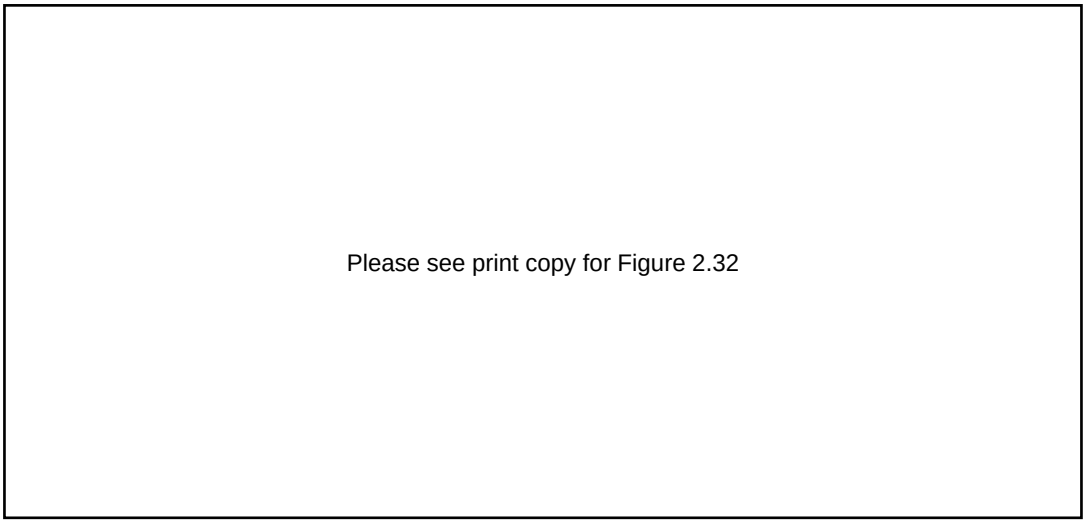

Figure 2.32 Model for FTP Use (from [38])

2.3.6 MPEG-7 TeM

MPEG-7 [7] is an ISO/IEC standard developed to describe multimedia resources such as audio, video and images. The MPEG-7 Systems specification [7] also defines a textual delivery method for delivering these multimedia descriptors that are described in XML. This delivery technique is known as the Textual Encoding format for MPEG-7 (TeM).

TeM relies on the principle that an XML document can be divided into smaller XML fragments, which can be reassembled at the client. The local version of the XML document on the client side is manipulated through TeM specific commands [39] sent by the server. The TeM commands are:

- AddNode - informs the client to 'append' the node the current list of children of the target node [39];
- ReplaceNode - informs the client to replace the specified node with the new fragment; and
- DeleteNode - informs the client to delete the specified node.

TeM does not offer the ability to insert a node between two, already received, consecutive

nodes [39]. However, a 'deferred node' can be added before adding other nodes. This deferred node notifies the client a child node exists, but it currently is not received yet. To add the content of a deferred node, the node is simply replaced with the new fragment [39].

TeM delivers the XML fragments via MPEG-7 Access Units (AUs); an AU is the smallest unit within MPEG-7 which contains relevant timing information. TeM supports two modes for delivery:

1. Synchronous - each Access Unit is assigned a unique time to indicate when the fragment (contained in the Access Unit) is becomes available to the client [39]; and
2. Asynchronous - the timing information required to notify when an AU is sent to a client is unknown to the producer or this information is not relevant for reconstruction [39]. If the producer of the stream chooses to define an order of delivery of the AUs, then the AU order of reconstruction is guaranteed at the client [39].

TeM is capable of fragmenting an XML document and delivering these XML fragments to a client to recreate the original XML document structure. However, the fragmentation process is determined purely by the author who created the fragments, without providing the receiver with the ability to specify fragment sizes and/or order. For example, if a user only wanted the last fragment at the end of the document, the user would have to wait until it is delivered, receiving many unwanted fragments.

2.3.7 Fragment Caching for Mobile Devices

A technique described in the literature [40] defines a method for caching fragments on a server and keeping in synchronisation with what is loaded on the client. The goal of this technique is to optimise the memory management of DOM applications running on mobile devices, as well as providing an efficient server-side caching technique to keep a copy of the clients cache on the server [40].

The paper [40] proposes a technique where a mobile device may choose between two methods for retrieving XML fragments. The first method for requesting XML fragments uses 'direct' requests; these requests use XPath for fragment selection [40]. Direct requests pro-

vide the ability to read, update, insert and delete fragments on a remote server. The second method uses indirect requests using what is called the MDOM (Mobile DOM) interface [40]; which uses the standard DOM API. Indirect requests only support read and delete operations on the remote XML document.

Although indirect requests allow modification of the remote document, unfortunately, direct requests only support read and delete operations which does not give the users the ability to modify or insert new data.

The paper [40] mentions a protocol to transmit these direct and indirect requests to the server, however, there is no mention to the underlying functionality of this protocol, nor are there any specifications of the protocol. Furthermore, this technique requires DOM API to be used on the client if modifications to the remote document are required.

2.3.8 Delivery in Mobile Environments

Mobile environments provide wireless connectivity to mobile devices (such as PDAs and mobile phones). Within mobile environments, retrieving vast amounts of information causes unnecessary delays due to the low bandwidth and shared nature of mobile networks; data rates using GPRS range from 56 kilobits per second to 150 kilobits per second and 3G data rates are generally between 128 kilobits per second (for high speeds) to 2 megabits per second (for fixed stations) [41]. Furthermore, transferring vast amounts of information costs users additional money; many mobile networks often charge users on a per kilobyte downloaded basis [33].

For example, in Australia as of September 2006 the following charges for mobile data access applied:

- Telstra [2] GPRS - Pay-as-you-go users are charged at 2.2¢ per kilobyte and pre-paid plans ranging from $5 for 0.25 megabytes (1.9¢ per kilobyte) up to $85 for 10 megabytes (0.83¢ per kilobyte);
- Optus [3] GPRS - pay-as-you-go users are charged 1.5¢ per kilobyte or a pre-paid plan

[2] http://www.telstra.com.au/
[3] http://www.optus.com.au/

can be purchased providing $4.95 for 4 megabytes (0.12¢ per kilobyte); and

- Three Mobile [4] broadband using 3G - users are charged $29 for 100 megabytes (0.03¢ per kilobyte) or pay-as-you-go at $4 per megabyte (0.4¢ per kilobyte).

Mobile devices are generally small and portable which have limited processing power and tight memory constraints when compared to laptops and desktop PCs. For example, many mobile phones currently on the market come with built-in memory ranging from 16 megabytes to 64 megabytes RAM. Furthermore, due to the size constraints, mobile screen sizes are often 100x100 pixels for small devices, 240x320 pixels for medium devices and up to 640x480 pixels for larger devices [42, 43].

Wireless environments have an added problem over traditional wired networks, in that noise can also affect wireless transmissions. Noisy environments can cause errors or even dropped packets, requiring re-transmission [33, 44]. Furthermore, a mobile battery life is reduced as it performs process intensive tasks, transmitting data over the air and uses more memory [45]. Thus, it is beneficial that delivering data to mobile devices in mobile environments are efficient in terms of both delivery and processing.

2.4 XML Compression Techniques

XML by nature has the consequence of increasing the overall document file size as compared to the original raw representation [46]; this increase can be significant depending on the original dataset, the level of descriptors added and semantics. There are two main contributing factors which account for this increase [47]:

1. XML is represented utilising a textual representation that can often be larger than the equivalent binary representation [47]. For example, an integer within the range -127 to 128 would require eight bits represented as two's compliment binary, whereas in textual form it would require four characters for negative values and three characters for positive values.

[4] http://www.three.com.au/

2. One of XML's main goals is a format specifically designed to be human readable and additionally provide interoperability amongst many devices and platforms [1,47]. Consequently, XML documents carry numerous whitespaces which is purely for formatting purposes [47], where each single tab/space consumes an additional character. Additionally, when surrounding raw text with 'tags', well-formed XML requires start tags to match end tags. For example, if long tag names are chosen, then this can greatly increase overall document size [47].

Due to the extra structural information added by surrounding data with XML tags, this produces XML documents that tend to be verbose. These verbose XML documents consequently increase the file size dues to the extra text which consumes additional diskspace and increases transmissions times. In an attempt to conserve diskspace and reduce bandwidth, several techniques have been created to convert an XML document into a binary equivalent format, which if needed, can be converted back into its original form. The authors of [2] suggest that there are two main reasons for users to represent XML in a binary format:

1. Size - Construct the binary XML such that the size overhead of the XML data is reduced; generally through XML-specific compression techniques [2]; and
2. Performance - Attempt to improve XML parsing performance by reducing the complexity of the implementation [2].

Binary XML formats generally incorporate some kind of compression to further reduce overall filesize. Occasionally some literature interchanges the terms 'binary' and 'compression'. Currently, there exist several XML compression techniques which have been developed in an attempt to reduce the overall size of the XML document. XML compression techniques can be classified into two main categories [48]:

1. Lossy XML compression techniques such as those proposed in the literature [48–50] do not create an exact replica of the original file when uncompressed. During the compression process lossy algorithms may make some approximations or reduce the

resolution of original data utilising semantic information (either in the Schema or deduced from the XML). For example, this process may round numbers to the nearest decimal point, or may simply drop elements which it deems may not be relevant.

2. Lossless XML compression techniques are capable of recreating an exact copy of the original document when uncompressed, or in some cases, the canonical equivalent document form [51] is created (i.e., may not be bit-to-bit exact, but still has the same logically equivalent context). Although some techniques do discard XML comments (i.e., text enclosed within the tags <!-- -->), processing directives etc, these techniques are still often considered as lossless.

An overview of all the various compression categories in terms of compression ratios is illustrated in Figure 2.33. The compression ratio increases from left to right in Figure 2.33.

Although lossy XML compression techniques exist (and can obtain good compression ratios), however data is lost or degraded through this process and usually requires user input or prior knowledge of the contained data to obtain good compression ratios. Thus, this thesis focuses primarily on lossless techniques since the objective is to deliver only user requested fragments of XML thus reducing the amount of data send.

Lossless XML compression techniques can further be broken down into three categories: Redundancy based compression, Schema based compression and Hybrid [52] compression. Many XML compression techniques (such as [53–55]) attempt to compress the XML so as to maintain the original XML document structure such that the entire document does not need to be decompressed in order to access small portions of the document which increases efficiency of add/remove operations.

2.4.1 Redundancy Techniques

Redundancy compression algorithms operate on the principle that there is repetition of data (strings or characters) within the document. Redundancy methods are used in widely known compression programs such as WinZip [56], and gzip [57].

Lossless redundancy techniques usually fall into two groups [58]:

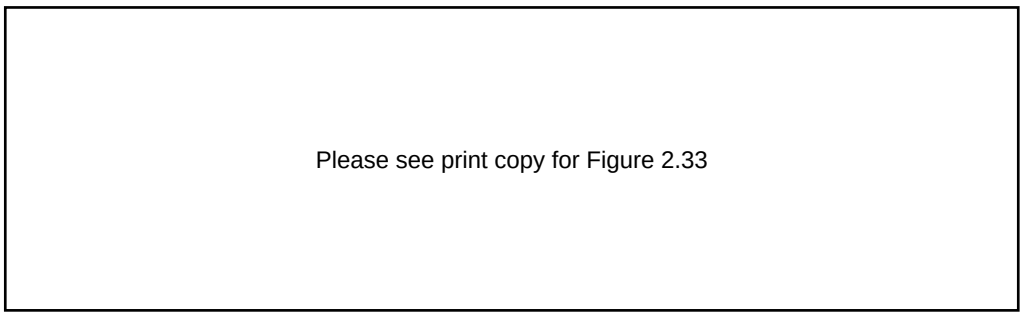

Figure 2.33 Different Kinds of compression and compression level [49]

- **Statistical** - these techniques generate a unique code for each character based on the probability of the character's occurrence within the document. The more frequent codes are assigned short codes while less probable codes are assigned longer codes [59]. An example of statistical encoding is Huffman [59] coding technique. For example, in the string `bbbbabcdbabab` the character b is most frequent and is assigned the shortest code, followed by a, then c and d would have the longest codes.

- **Pattern** - these techniques scan for duplicate strings within the data. Duplicate strings may be replaced with a "pointer" or reference to the first occurrence, or by an index value into a dictionary which contains mappings of strings to index codes [60]. An example of pattern redundancy compression is the well known Lempel-ziv [61] technique. For example, in the string `abcdefabcabc` the substring `abc` is duplicated three times, and the second and third occurrence would just contain a pointer to the first occurrence.

2.4.2 XML Conscious / Schema Based Compression Techniques

XML conscious compression techniques utilise the fact that the resulting XML file has a standard tag-based format and structure and exploits this knowledge to achieve better compression ratios (i.e., it is known that for a valid XML document a start tag must be closed with a "closing" tag of the same name). Schema based compression techniques take advantage of the fact that the XML schema is external and can be:

- assumed to be common to both encoder and decoder; and

XML Schema Text	Binary Code
<sequence> <element name="childa" minOccurs="1"/> <element name="childb" minOccurs="1"/> <element name="childc" minOccurs="1"/> </sequence>	 00 01 10

Figure 2.34 Example of XML Schema fragment and assigning binary codes based on the element position

- the schema contains important structure information about the XML document being compressed.

The schema provides encoders and decoders useful information such as specifying the number of possible children of an element and enumerated datatypes. Thus, an XML conscious coder only needs to encode the position and does not need to encode the total number of children as it is predetermined by the schema. For example, the XML Schema fragment in Figure 2.34 shows a sequence of three child elements. All these element have the restriction of minOccurs equal to one, thus the encoder and decoder knows that all the elements need to be present. Since there are three nodes only two bits are needed to represent all three elements. To represent the node `childa` in XML (i.e., `<childa>` would require 8 characters (64 bits), whereas as binary it can be represented with 00 which is only two bits.

It is important note, that in order to apply schema based compression techniques, the XML must be valid to a schema in order to utilise information in the schema (such as in Figure 2.34). Furthermore, schema based methods require the schema to be available at both the encoder and decoder.

Current XML conscious and schema based compression techniques will be further investigated.

2.4.2.1 WAP Binary XML

The WAP Binary XML (WBXML) [62, 63] attempts to represent an XML document as a compact binary representation. WBXML is designed to increase transmission times without loss of functionality or semantic information. The first stage of WBXML requires to the

XML document to be tokenised, which is a process whereby all markup and XML syntax (such as tags, attributes and entities) are converted into their corresponding tokenised format [62]. During the tokenisation process, all comments are removed and processing directives (only for the purpose of aiding the tokeniser) may be removed [62]. Tokens are split into two categories, either 'global' or 'application', and are chosen such that they are comprised into a set of overlapping code spaces [64]. Inline data (such as strings) and miscellaneous control functions are encoded into global tokens. Application tokens, which have a context-dependant meaning, are divided into two overlapping code spaces: "tag code space" for representing specific tag names, and "attribute code space" for attribute names [64]. An example XML and assigned tokens are illustrated in Figure 2.35, and the resulting output illustrated in Figure 2.36.

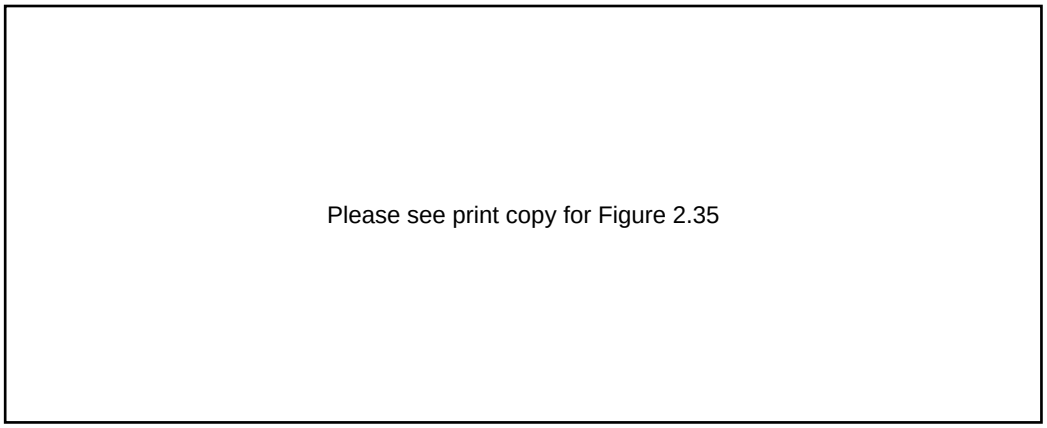

Figure 2.35 Simple XML document and corresponding tokenisation assignment (from [62])

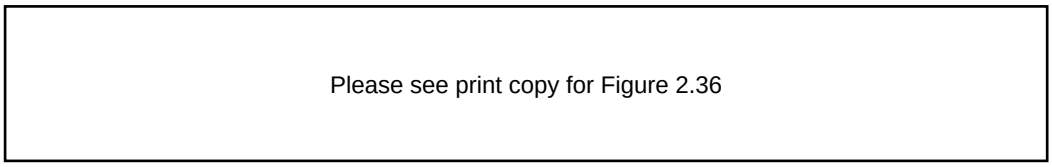

Figure 2.36 Example of the output stream from the Example XML in Figure 2.35 (from [62])

2.4.2.2 Millau

Millau [64] is an extension of the WAP Binary XML Format focusing on serialisation and streaming of large XML documents. In the WBXML format, character data is not compressed; it is sent as inline strings (i.e., using base64) or as a reference [6]. Millau operates slightly different to WBXML such that it can separate the content and character data, and

deliver each on separate streams [6]. An advantage of sending the character data on a separate stream is that it can additionally compress this data by using traditional redundancy compression techniques [64, 65].

Millau functions very similar to WBXML by encoding all the elements and attributes of the XML document as tokens. After the XML document is tokenised, Millau reorders XML such that the most important information is delivered first [6].

To cater for the fact the Millau separates the character data, additional global tokens [6] are required. These additional tokens are either STR or STR_ZIP to represent the uncompressed character data and compressed character data, respectively. An advantage of transmitting the character data on a separate stream is to avoid the necessary overhead on embedded binary formats by storing the binary data in its textual representation (i.e., this can save approximately 33% overhead when compared to using the base64 [11] technique). The architecture of the Millau compression/decompression system is illustrated in Figure 2.37.

Millau obtains a linear compression rate on the average length of the node names since XML nodes and attributes are replaced with tokens [64]. The compression ratio is further increased by the benefits provided by the algorithm used to compress the character data [6]. Millau typically achieves good compression ratios on documents with file sizes between 0 - 5 Kilo-Bytes [6].

Although Millau reorders the fragments to deliver more important information first, there is no mechanism to do this on a per user basis.

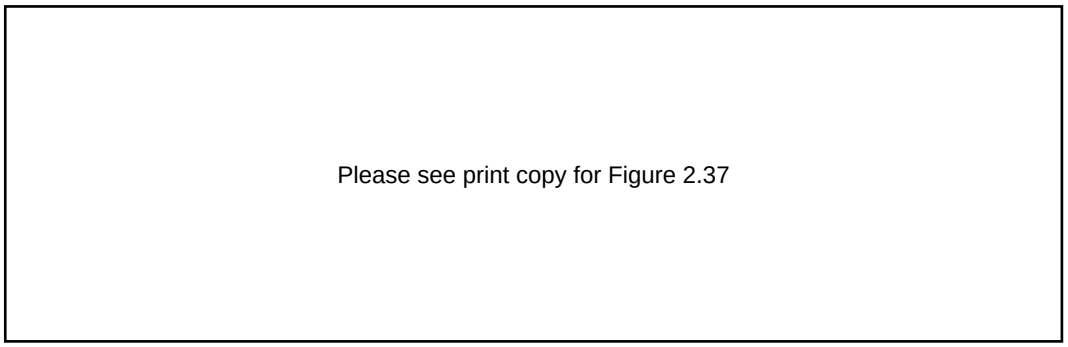

Figure 2.37 Architecture of the Millau Compression - Decompression System [65]

2.4.2.3 XMLPPM

XMLPPM [10] employs a modeling technique which is referred to as Multiplexed Hierarchical Modeling (MHM) and further incorporates a Prediction by Partial Match (PPM) [66] algorithm. XMLPPM utilises two techniques: multiplexing several text compression models based on XML's syntactic structure, and injecting hierarchical element structure symbols into the multiplexed models [10]. Within PPM, models are created to maintain statistics about which symbols have been seen and the context of preceding symbols [10]. The PPM model is used to to estimate a probability range of a symbol which is used to transmit the symbol using arithmetic coding [10]. XMLPPM multiplexes several models switching between them based on a syntactic context supplied by the parser [10]. XMLPPM utilises four models, a separate model is used for : 1) structure, 2) element and attribute names, 3) attributes and 4) strings [10].

2.4.3 Hybrid Techniques

Hybrid techniques attempt to further increase XML compression by employing a combination of both schema and redundancy based methods. This may be as simple as applying a redundancy compression technique on top of a schema based compression. Redundancy methods could also be applied within parts of the XML data whilst maintaining the schema structure information. In the literature, [52] demonstrates that hybrid techniques generally yield better compression ratios on average than either the redundancy or schema based techniques alone. Some of the more popular hybrid techniques are further investigated.

2.4.3.1 XMill

XMill [46] is an XML-conscious compression technique which claims to achieve around twice the compression ratio when compared to traditional compressors (such as gzip) [46]. XMill takes advantage of already well known data compression techniques (such as zlib [67]) as well as other specific datatype compressors [46]. XMill also has the ability to include user-defined compression techniques for application specific datatypes [46].

XMill uses three main techniques for compressing XML documents [46]:

1. Separate structure from data - Within an XML document, the structure is typically the XML tags and its attributes. XMill separates the structure and data from the document and compresses the structure separately [46]. The separated structure consists of a sequence of strings which represents the element and attribute values [46];

2. Group related data items - All similar data items within the document are grouped together into containers, and each container is then compressed separately [46]; and

3. Apply semantic compressors - XMill applies different compression techniques to each container, depending on the datatype [46]. For example, a container holding text could use a different compression technique to a container holding numbers.

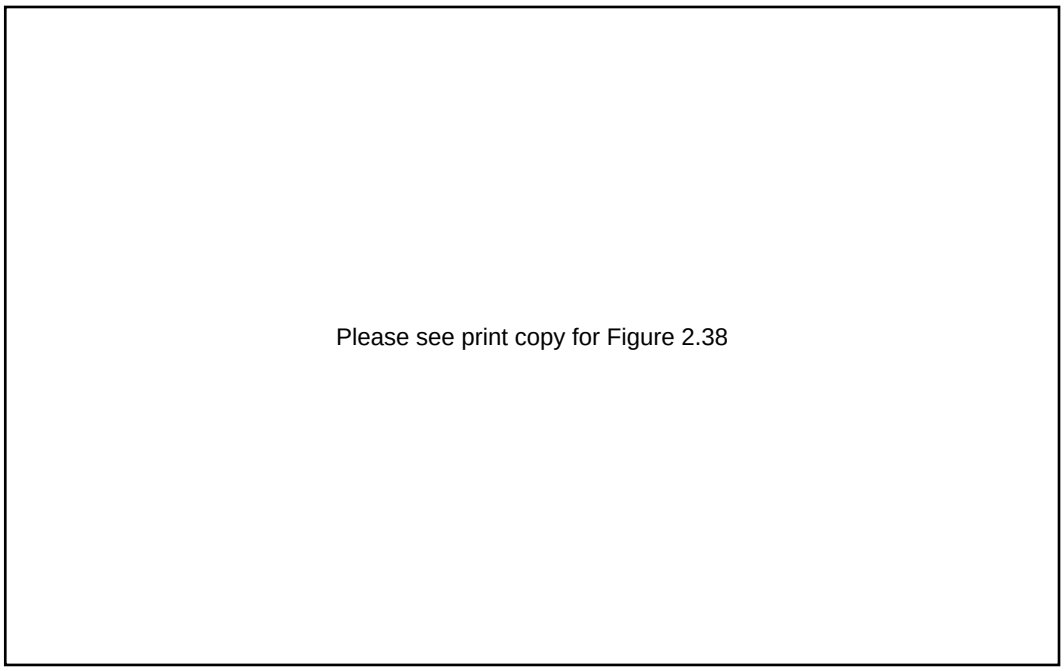

Figure 2.38 Architecture of XMill Compressor [46]

The XMill architecture is illustrated in Figure 2.38 which shows how the techniques fit together. XMill has speeds comparable to gzip [46], however the tradeoffs are: it does not support direct querying on the compressed format [46, 58], and only achieves good compression ratios when the dataset is large, typically over 20 kilobytes [46]. Furthermore, user assistance is required to achieve the best compression [10], which requires the user to have knowledge of the structure and data.

2.4.3.2 MPEG-7 BiM

MPEG-7 [7] adopted the Binary Format for MPEG-7 Metadata (BiM) as the technique to provide compression of MPEG-7 XML files. MPEG-7, unlike earlier MPEG standards, is an extensive metadata standard which covers the description of video, audio and general multimedia content. Although BiM was developed within MPEG-7, it is still applicable to most schema-valid XML documents.

BiM [7] is a tree-based compression technique that utilises knowledge that schema information should be common to all users and this information can thus be regarded as a priori information [68]. The consequence is that all XML files conformant to that schema can be significantly compressed through binary encoding of the tags [52]. BiM operates on the principle that every child node of the schema tree can be assigned a binary code [7]. This allows the binary code to be represented in binary rather then the text of the XML tag. BiM applies datatype specific compression based on the information provided by the XML Schema [7].

Whilst MPEG-7 considers BiM a lossless compression technique, it does not replicate the exact XML file after reconstruction due to some parts of the original XML is lost. Parts which BiM does not process include nodes containing comments, pre-processing information etc [7]. BiM is engineered to ensure that the structure and data remain intact and correct in XML files reconstructed from the binary stream (given the availability of the schema file).

2.4.3.3 XGrind

XGrind [58] is another XML compression technique that supports queries which can be directly applied within the compressed domain [58] (i.e., the data can directly be queried without the need to decompress the data first). The capability to directly query compressed documents is especially useful on resource-limited devices which have limited CPU processing and memory/storage (i.e., mobile devices) [58]. In many cases, these resource-limited devices may be unable (or would be too time consuming) to fully decompress the compressed XML document in order to perform a query on the extracted XML document.

XGrind provides the ability to query the compressed format since it maintains the same struc-

ture from the original XML document [58]. Furthermore, maintaining the structure allows the same parsing techniques as currently available for text XML documents [58], since the compressed document is still an XML document [69]. XGrind compresses at the level of individual elements or attributes within the XML document, utilising a context-free compression scheme which is based on Huffman coding [58]. Further efficiency in compression is achieved by exploiting information present in the DTD. XGrind uses different techniques for compression, these are described below [58]:

1. Meta-data Compression - All tags are encoded by a 't' followed by a unique element-ID. with all end-tags are encoded by a '/'. Attributes are encoded by a 'A', followed by a unique attribute-ID [58];

2. Enumerated-type Attribute Value Compression - These types are encoded by using knowledge within the DTD. To represent all enumerated K values, a $log_2 K$ encoding scheme is used [58]; and

3. General Element/Attribute Value Compression - A two pass encoding scheme is used to encode these datatypes. The first stage collects statistics about the document In the second stage, strings are encoded by assigning a code which is independent of its location within the document. This has the advantage that given any arbitrary string, further occurrences within the compressed document can easily be found without any decompression.

The architecture and compression process of the XGrind system is illustrated in Figure 2.39.

2.5 Conclusion

This Chapter investigated XML schema languages, XML Parsing techniques and delivery techniques applicable for transport of and compression of XML documents. There are many XML schema languages each with their own advantages and disadvantages. In particular, XML Schema utilising typing of elements, Relax-NG using pattern matching and DSD using a rule based approach. XML Schema is currently the most popular XML schema language

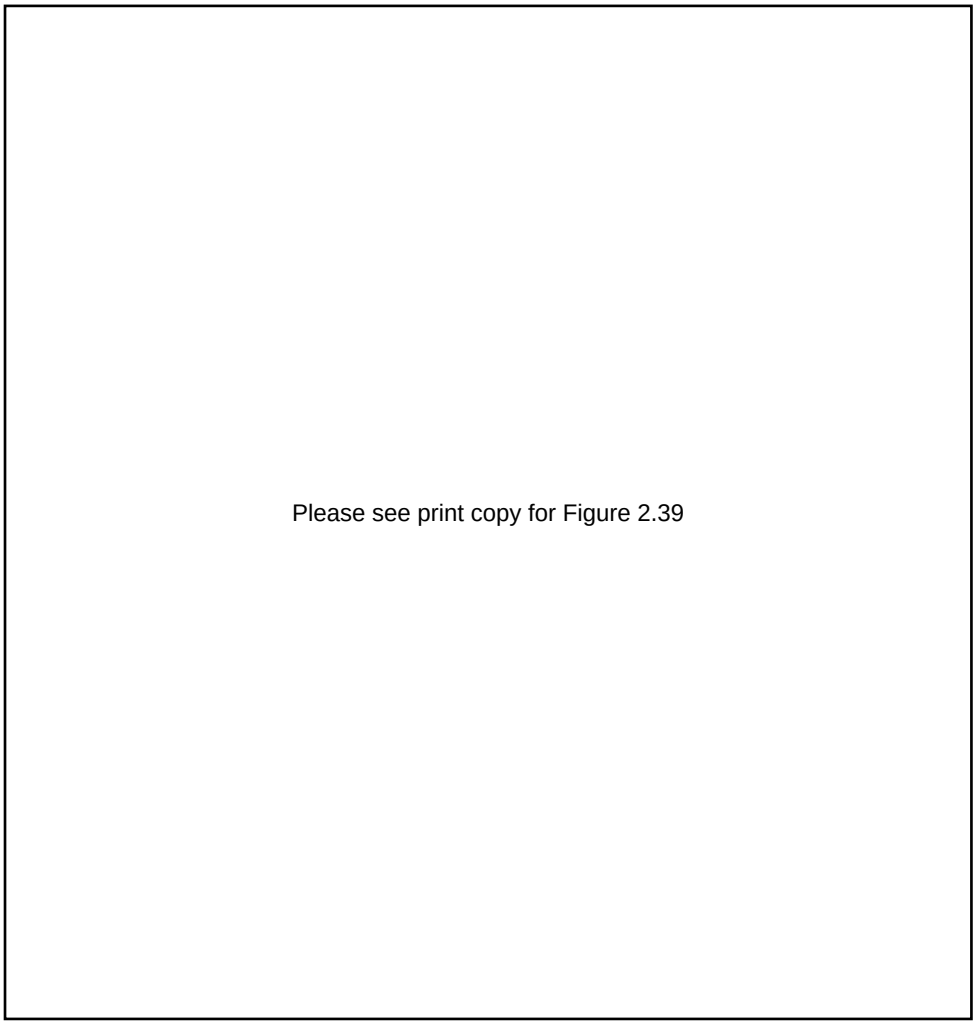

Figure 2.39 Architecture of the XGrind Compressor [58]

due to the number of tools which build upon XML Schema typing ability, such as XQuery [12]. Due to XML Schema's popularity and number of schemas available, XML Schema will be the main focus as the schema language throughout this thesis.

This Chapter then investigated the popular XML Parsing techniques and the shortcomings of current XML delivery techniques. Generally, these delivery techniques do not provide the user with the ability to select their own fragments and fragment size, resulting in the user receiving unwanted data.

Lack of research contributions into the two-way delivery of XML to mobile devices is possibly due to recent mobile devices becoming more capable in terms of memory and processing. The lack of research into the area of XML Schema negotiation techniques is possibly due to the assumption that the entire XML Schema is always required.

This Chapter then summarised current XML compression techniques and their suitability to be adapted into a technique driven by user requests.

Analysis of the literature within this Chapter raises some research problems, these are:

- XML documents, such as those used within MPEG-21, have the potential to become very large in terms of filesize. These large filesizes may adversely affect transmission of XML documents, especially within mobile environments;
- Current XML delivery techniques generally deliver the entire document to the client. Unfortunately, these methods may consequently waste bandwidth depending on use of the XML document. In many cases, XML documents provide more information, or duplicate information formatted in a different way, which may not be desired in a given application or by some users. In these situations it makes sense to be able to retrieve only parts of the XML document which are relevant;
- Generally, most XML compression techniques compress the XML document as a whole which needs to be decompressed before it can be fragmented. Current techniques which allow compressed fragmentation also do not allow the user to request compressed fragments. Users should be able to use a request technique to deliver com-

pressed fragments. Ideally, the delivery technique should be compressable too; and

- For clients to perform XML validation or manipulation keeping the document valid with respect to a schema, the entire schema is also delivered to the client. There are many cases where only a subset of both the XML document and corresponding schema declarations are required, especially with the growing popularity to use XML as a container format (such as MPEG-21).

Chapter 3

A New XML Delivery Protocol

3.1 Introduction

Techniques for the transmission of XML documents were reviewed in Section 2.3. It was shown that current techniques do not provide mechanisms to allow users to request and retrieve only relevant portions of a remote XML document.

This Chapter thus introduces the development of a novel protocol to deliver user requested XML fragments from one location to another. This protocol effectively provides clients the ability to 'pull' relevant parts of a remote XML document through navigation techniques. Use of navigation techniques on remote XML document allows the user to skip over unwanted data, retrieving only the parts of the XML document which are of interest. This is especially beneficial for mobile users, operating in slow and restrictive environments; many mobile providers often charge users on a per kilobyte download charge (see Section 2.3.8).

3.2 A Remote XML Pull Protocol

There are several techniques that are available to users to retrieve XML documents from a remote source (see Chapter 2.3). Many of these techniques blindly deliver the entire contents of the XML document, regardless of what data within that file the client may actually require.

XML has obtained widespread adoption for the storage of data as it has several advantages over binary only formats (see Chapter 2.2). One such advantage is the added structural infor-

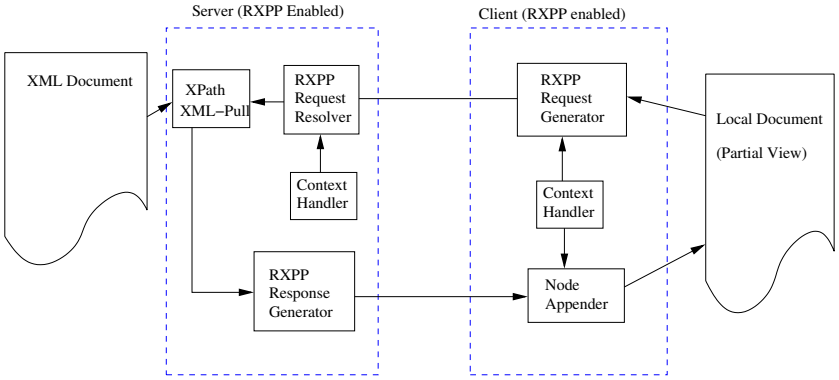

Figure 3.1 Example block diagram of the proposed system, illustrating the communication between the server and the client to exchange XML documents

mation to the raw data; this structure provides a 'tree' view, where each branch and subbranch are clearly visible. Furthermore, the structural information provides users with a selection of only the branches that may pertain relevant information. However, most transport protocols (such as HTTP) are unable to give users direct control over the transmission (i.e., they do not allow a user to retrieve the data from a selected subbranch).

XML Pull Parsing [20] provides the client with an API which gives direct control over how a locally stored XML document is parsed and traversed (i.e., moving through the structure of the XML document). XML pull parsing would be advantageous if it were extended to provide remote clients the same document access (i.e., provide true remote pull parsing).

This Chapter details the development of a new protocol (or messaging system) to effectively relay the exchange of client requests and relevant responses between the two devices.

Throughout this Chapter, the device containing the XML document will be referred to as the server, and the client will refer to the the device requesting the remote XML document. An example block diagram of the protocol is illustrated in Figure 3.1

3.2.1 Requirements

The commands that make up the protocol should effectively allow a client to ask for small pieces (or fragments) of XML from the original remote XML document. These small pieces of XML require additional information to be transmitted such that the client can precisely ap-

pend the XML fragment to its local version of the XML document, whilst keeping the local XML document structure synchronised with the original remote XML document. By keeping the document synchronised with the original remote XML document, the client can progressively replicate the original XML document, including its structure (minus the information that is not needed). This ensures that any hardcoded XPath expressions in applications, will still return the correct results.

To assist in developing the new protocol, some requirements / use cases have been developed; they are as follows:

1. A client must be able to navigate though the remote XML structure:
    - One node at a time (i.e., go to the next sibling node). This may be useful for users on mobile devices who may wish to move trough the document node-by-node, choosing only the desired nodes to retrieve; and
    - Expanding a branch at a time (i.e., retrieving all the child nodes of a selected node). This allows a user to navigate through the structure (viewing the children of each selected node) to make decisions on which nodes to retrieve next;
2. A client must be able to navigate backwards though the document without needing to re-transmit the document from the beginning. This is advantageous where the client had not buffered (or cached) the previous nodes on navigation;
3. The protocol must be small, simple and lightweight, such that implementation is possible on mobile devices without performance penalties;
4. A server implementing the protocol should support one or both of the following modes of operation:
    - Maintain State - a server will remember the current context of a connected client. The server will remember where each connected clients' position is within the XML document; and
    - Stateless - a server need not remember the current context of a client. In this case, the client will need to specify the XML document and full path of the nodes to

RXPP Request := | Command | Parameters |

Figure 3.2 RXPP field definition for and RXPP request

retrieve on every connection.

5. A client must be able to specify the number of descendant child nodes to be transmitted upon selection of a node.

These requirements have been generated to cater for standard devices (such as laptops and PCs) but with an additional focus on mobile devices (such PDAs and mobile phones) operating in restricted environments (i.e., limited/shared bandwidth).

For example, a mobile device with limited memory and screen size could simply request the correct number of XML nodes (created for PC usage) to fill the screen and only request the rest if/when the next page is called. This reduces the need for mobiles to cache and parse large documents locally, additionally saving time waiting and also reduces bandwidth requirements.

3.2.2 RXPP Requests

RXPP requests are created by one device (i.e., client) and delivered to a RXPP capable receiving device (i.e., server). An RXPP request instructs the server on what the client wants to do (i.e., request a node or select an XML document). There are a number of RXPP commands (see Section 3.3.2), where each command specifies different arguments. Thus, an RXPP request packet will be structured into Command and Parameters fields (as illustrated in Figure 3.2).

The RXPP fields are further described as follows:

- **Command** - The command field defines what method is required to be performed on the server side. RXPP accepts the following commands: `Src` (see Section 3.3.2.1), `XPath` (see Section 3.3.2.2) and `XML-Pull` (see Section 3.3.2.3). After a `Src` command is received, the server will keep the connection open awaiting the next command to allow an `XPath` or `XML-Pull` command; and

- **Parameters** - The format of the parameters field depends on the preceding command field (i.e., each command has different values for the Parameters field).

  `Src` Parameters

  - mandatory parameter: file location (i.e., /files/test.xml); and
  - optional parameter: connection mode which has a value of ether Open or Closed.

  `XPath` Parameters

  - mandatory parameter: XPath locator (i.e., /book[1]/chapter[2]); and
  - optional parameter: level depth, which has a value of either a positive integer or -1.

  `XML-Pull` Parameters

  - mandatory parameter: One of the following XML-Pull commands: Next, Up, Expand or Back; and
  - optional parameter: level depth, which has a value of either a positive integer or -1.

3.2.3 XML Fragmentation

To provide clients with the ability to receive selectable parts of an XML document, the original XML document needs to be broken up into smaller, and uniquely identifiable fragments. An XML fragment is defined as a small piece of the original XML document. An XML fragment may contain structure: such as branches, subbranches or parts of subbranches of the original XML document. An XML fragment may additionally contain an entire subtree (node plus all its descendants), a partial subtree or simply a single node.

To further illustrate this concept, an example XML document (illustrated as a tree) is shown in Figure 3.3. Figure 3.3 additionally illustrates one of many ways in which a large tree can be divided into smaller XML fragments (in this example, the tree is divided into two fragments). When these two fragments are merged back together at the correct node, they form the original XML document.

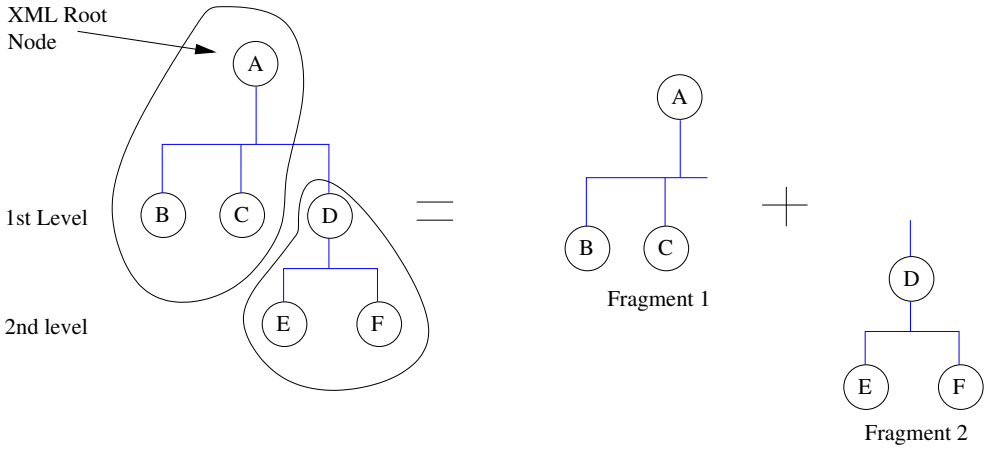

Figure 3.3 Example of breaking a tree-like structure into two smaller fragments, which when added together, form the original structure

3.2.4 Node Identification in the Remote XML Document

Section 3.2.3 demonstrated a simple example to break an XML document (viewed as a tree) into smaller pieces. In order to recreate the original document, these fragments need to be appended to the correct node, which requires the 'path' to that node to be recorded and transmitted to the receiving device. XPath [23] is the most common technology which is used to describe the path to an element within XML documents. The expression in XPath is traced from an existing known node (in many cases, this is the root node of the XML document) to the desired node (for more information see Section 2.2.4). Furthermore, with the ability to describe the path to any node within an XML document, XPath effectively provides random access into the XML document. The 'path' part is only a small portion of the XPath language, which will be referred to as XPath Locators throughout this thesis (i.e., XPath locators do not allow any of the querying methods of XPath).

3.2.5 Remote XML Navigation using RXPP

The RXPP commands (see Section 3.3.2), provides commands to allow a client to effectively 'navigate' or 'browse' though a remote XML document. This navigation is possible due to XML's tree-like structure, where a user can look at the XML structure and data by 'stepping through' the XML document either by the nodes or branches of the tree. Navigation thus allows a user or application to literally 'browse' the XML document, making decisions based

on the structure of the tree. This is analogous to looking at the contents from a book and selecting a chapter to read.

Navigation is desirable when the structure and/or contents of the remote XML document is unknown (e.g., there is no schema) to the client. A client can move through the structure of the document, making decisions based on the XML node names and attributes. There are two techniques to perform navigation: stepping through the document, node by node or opening a branch of XML at a time. These two techniques are now described in detail.

3.2.6 Node by Node Navigation

Fragmentation of XML using navigation on a node by node basis is achieved via the XML-Pull commands (see Section 3.3.2). The XML-Pull commands allow the client to navigate through the remote XML document by retrieving one XML node at a time. This technique provides the client with the ability to:

- receive the next sibling with respect to the current node;
- receive the previous sibling with respect to the current node;
- receive the parent node with respect to the current node; and
- receive the first child of the current node.

Considering the example in Figure 3.4, a connection to the server has already been established, using the open connection mode, and has already navigated to node `b` (the current pointer on the server is set to `b`). The client decides to issue the 'next' command, receiving node `c` at time t3 as illustrated in Figure 3.4, and moving the current pointer on the server to node `c`. Following on from Figure 3.4, the client now issues the `expand` command, as illustrated in Figure 3.5, and receives the node `d`, which is the first child node of `c`.

3.2.7 Level by Level Navigation

When navigation is performed on a level by level basis, the XML fragments received are the direct children of the requested node. When a level depth is provided, the specified number

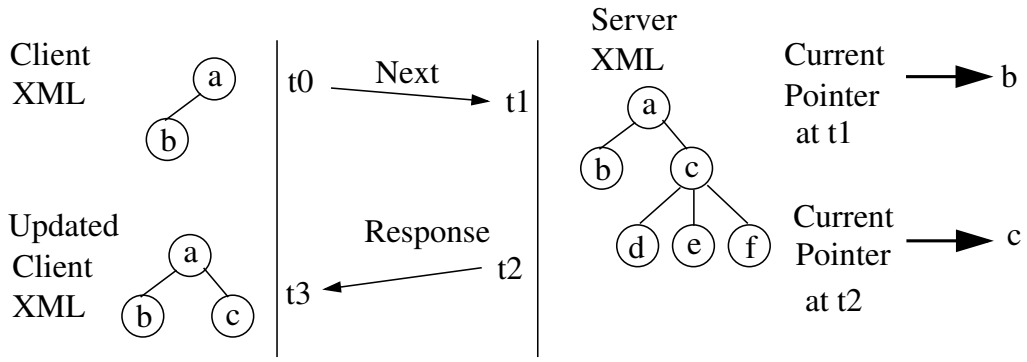

Figure 3.4 Example of a Client beginning at t0 requesting the 'next' node. The server moves the current pointer to the Node c and delivers this node to the client. At time t3 the client adds this to its local version.

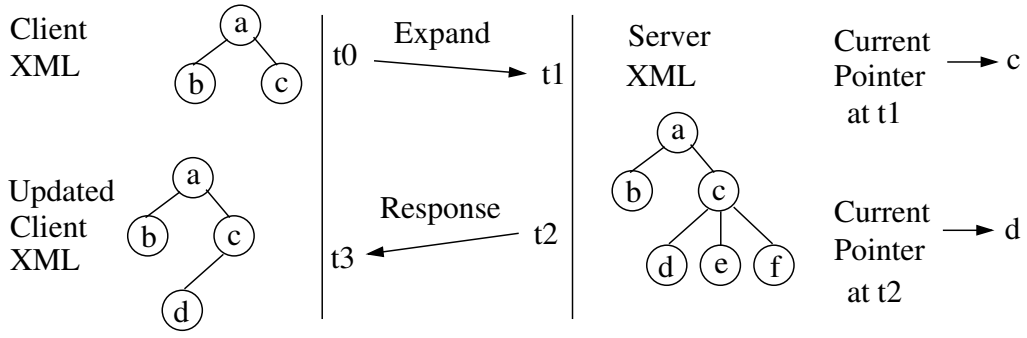

Figure 3.5 Example of a Client beginning at t0 issuing an expand command on the 'current' node c. The server moves the pointer to d, the first child of c, and sends these to the client. At t3 the client adds the node to its local version.

of descendant children is additionally retrieved. Level by level navigation provides more information as compared to node by node navigation (see Section 3.2.6), as it reveals more information by retrieving all the direct children.

For example, Figure 3.6 illustrates that a client has already received the children (nodes `b` and `c`) of node `a`. If a client wishes to receive the next level (i.e., all child nodes) of `c`, the XPath locator `/a/c` is sent and all first level descendants of node `c` are received, as illustrated at time t3 in Figure 3.6. If the client was interested in a node within these children, it could request to retrieve the next level, thus skipping the download of the subtrees of unwanted child nodes.

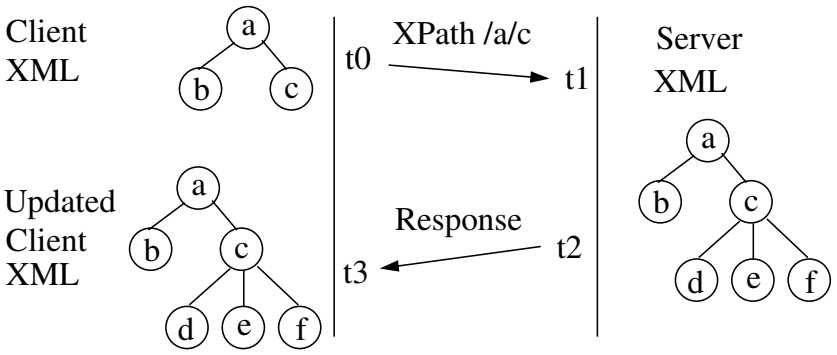

Figure 3.6 Example of a Client beginning at t0 requesting all children of the node c using an XPath locator. The server locates all children of c and sends these to the client. At t3 the client adds the children to its local version.

3.3 The Remote XML Pull Protocol

Considering the requirements in Section 3.2.1, a set of commands were developed in this thesis. These commands form the basis for the Remote XML Pull Protocol (RXPP) which can query an XML file in a remote location, and transmit only the requested XML data. Efficient exchange of remote XML documents can be achieved through combination of XPath methods and XML Pull techniques.

3.3.1 Invoking an RXPP Connection

Internet applications currently use URLs to identify a protocol, a remote device and the path to a given file. For example, `http://www.example.com/test.xml` defines that

`http` is the selected protocol, the remote device address is `www.example.com` and the file requested is `/test.xml`.

Utilising current standards, RXPP connections can be defined using a standard URL[1], where the protocol name is `rxpp`. Thus, to intiate an RXPP session on an XML document from remote server, the following URL is constructed:

```
rxpp://server.address:port/path/to/file.xml
```

For example, to initiate an RXPP request to the server with the address of `www.elec.uow.edu.au` and specifying an XML document `test.xml` in the `samples` directory, the following URL is constructed:
`rxpp://www.elec.uow.edu.au/samples/test.xml`.

3.3.2 RXPP Commands

Section 3.2.1 detailed a number of requirements for RXPP. By grouping the common requirements, there are three main RXPP commands: `Src`, `XPath` and `XML-Pull`. These commands are further detailed.

3.3.2.1 XML Source

The XML Source (abbreviated as `Src`) command allows a client to notify the server which XML file the client wishes to remotely access. The server will proceed to locate and load the requested XML document. To give the client a starting point for navigation, the RXPP server will respond to the `Src` command with the root element of the requested XML document. This provides the client with the root node and any attributes on that node (i.e., schema declarations).

The `Src` command has an optional second parameter field to specify the 'mode' of the connection. The mode defines how the server should maintain connection(s) to the client as specified in Section 3.3.2.4.

[1] For widespread adoption of RXPP, the protocol would need to be registered with the IETF

3.3.2.2 Select Node

Since XPath already satisfies the requirement for the selection of a node within an XML document, it will be used as the Select Node (`XPath`) command. The `XPath` command notifies the server that an XPath locator is being received. Evaluation of the XPath is done on the server and is executed on the XML document which has already been opened with the `Src` command. The appropriate resulting XML fragment from the XPath locator is then transmitted back to the client.

The requirements in Section 3.2.1 require that when a node it selected, a client may specify the number of descendant children (of the selected node) to be received. This will be referred to as "level depth", and thus the `XPath` command may contain a second optional parameter that represents a level depth.

The level depth parameter is defined as either a positive integer or -1. A level depth of zero indicates that no descendant child nodes are to be retrieved (only the selected node and its attributes are sent). A value of -1 specifies that all descendant children of the select node(s) are to be retrieved (i.e., the entire subtree specified from the selected node). For any other positive value for level depth that number of descendant children are retrieved, if the requested depth is greater than the available depth, the server will just send all the available descendant nodes.

An example XML document (shown in a tree view, where nodes are represented by circles) stored on a remote server is illustrated in Figure 3.7. Suppose a client has already navigated to node `C`, and requests the XPath expression of `/A/C`, with a level depth of of -1 (i.e., retrieve the entire subtree), the fragment returned is illustrated in Figure 3.8.

As another example, a client establishes an RXPP connection to the document as illustrated in Figure 3.7, and receives the first node (`A`). If the client requests node `A`, with a leveldepth of 1 (i.e., retrieve the direct child nodes), the fragment received is illustrated in Figure 3.9.

Another example, a client establishes an RXPP connection to the document as illustrated in Figure 3.7, and receives the first node (`A`). If the client requests node `A`, with a leveldepth of

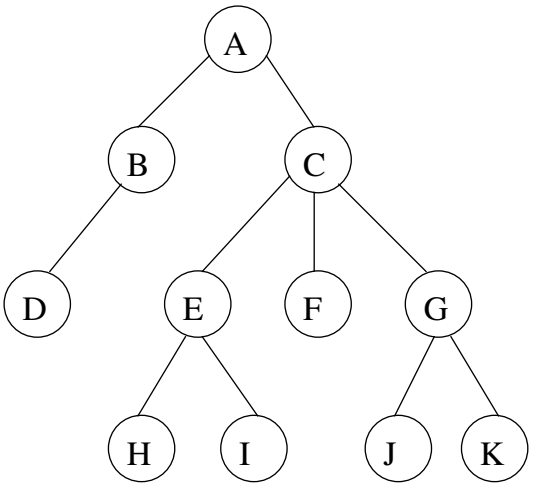

Figure 3.7 Example XML Document represented in a tree structure where the XML nodes represented as circles with the node name inside it

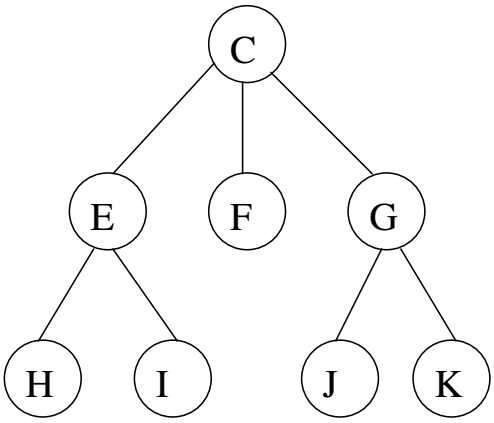

Figure 3.8 Example fragment received by the the client after issuing a level depth of -1 on node `C` from Figure 3.7

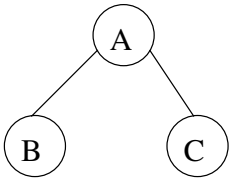

Figure 3.9 Example fragment received by the client after issuing a level depth of 1 on node `A` from Figure 3.7

2 (i.e., retrieve two levels of descendant nodes), the fragment received is illustrated in Figure 3.10.

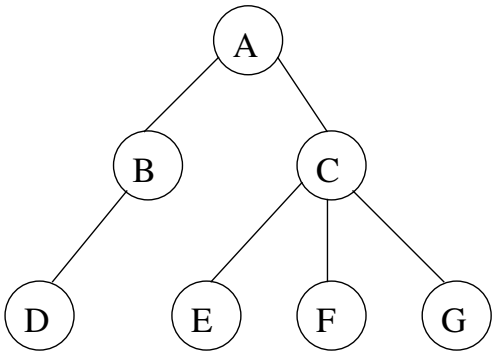

Figure 3.10 Example fragment received by the client after issuing a level depth of 2 on the root node `A` from Figure 3.7

3.3.2.3 `XML-Pull` Command

The `XML-Pull` command provides the client with the ability to 'pull' nodes from the remote XML document. Observing the requirements in Section 3.2.1, there are four commands that clients may use to pull XML nodes: Next, Up, Back and Expand.

These `XML-Pull` commands are only available when the RXPP connection is created in open mode since the server needs to be instructed to to remember the current context for the connected client. This scenario can be thought of as a client controlling a pointer into the XML document on the server side, and the client RXPP commands instruct the pointer to move one node at a time. The possible `XML-Pull` commands available are:

- **Expand** - This command can be thought of as 'expanding' the branch of the current node, revealing access to the child nodes of the selected node. The expand command moves the current pointer on the server to the first child node and transmits the first child node to the client;
- **Up** - This command moves the pointer up a level in the tree, which is the parent node. The Up command provides a simple methods to go backwards though the structure;
- **Next** - This command retrieves the 'next' sibling node after the current node, in the order that the nodes appears in the XML document; and

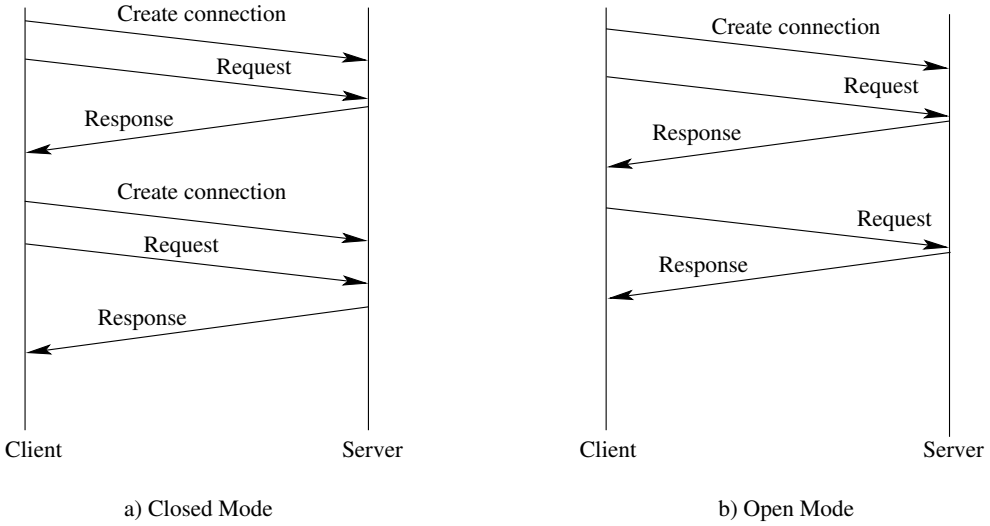

Figure 3.11 Example timing diagram illustrating the difference between Open and Closed modes of maintaining a server connection. a) illustrates the requests and responses when in closed mode. b) illustrates the requests and responses when in open mode

- **Back** - This has the reverse effect to the Next command, it moves the current pointer back to the previous sibling, transmitting this node to the client.

The XML-Pull commands provide users with the ability and methods for the creation of XML fragments whilst navigating remote XML documents. Examples and usage of XML-Pull is further described in Section 3.2.6.

3.3.2.4 Maintaining Server Connections

From the requirements in Section 3.2.1, a server should support both state and stateless modes of operation. These two modes of operation specify how the connection is maintained between the client and the server. These two modes of operation will be referred as: Open (maintain state information) and Closed (stateless mode of operation). An example of the differences between open and closed modes of operation are illustrated in Figure 3.11, and each mode is described below:

Open Mode Connection

In RXPP, an open mode connection indicates to the server that the client wishes to keep the connection open for future messaging; this eliminates the need to establish a new connection

each time a new message is to be delivered. This is advantageous as it reduces the overheads (for example, opening a new TCP socket) needed to create a new connection with each request as well as the reduction in time required to establish each new connection. The period in which the client is connected to the server with a single connection will be referred to as an ‘ RXPP session’. For example, Figure 3.11b illustrates a timing diagram for an open mode connection.

When a client requests an open mode connection, this indicates to the server that it should additionally keep track of its position locally on the serverside XML file (i.e., to store context information).

For example, if a connected client had previously navigated to node `b`, the server would remember that the current node is `b`. If the client requested to navigate to the ‘next’ sibling node, the server would know that the client requires the next sibling node after node `b`.

In open mode, the XML pull commands (see Section 3.3.2.3) become available for use within RXPP, and additionally, can accept XPath expressions using the `./` notation (which indicates the ‘current node’).

Closed Mode Connection

A closed mode connection within RXPP indicates to the server that it need not remember its current location within the XML. In this case, XML Pull commands (see Section 3.3.2.3) are unavailable and each request will be treated independently, thus requiring full XPath locators. For example, the `./` notation no longer has context as the server does not remember the current node. Figure 3.11a illustrates an example timing diagram when using RXPP closed mode.

A closed connection is useful where single requests are desired. Single requests can be used in cases where the XML Schema or XML conformant to a schema, may be known by the client or application in advance, and the client can then request the data directly without the need to navigate through the XML document.

For example, Figure 3.12 shows a simple XML Schema describing a book. If the client was

only interested in the title of the book and has the XML Schema in Figure 3.12, then it could use the XPath expression of `/Book/Title/text()` to request the title data, as it knows from the XML Schema the structure of the XML document (where the XML is valid with respect to the XML Schema).

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
  <xs:element name="Book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Title" type="xs:string"/>
...
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 3.12 An example book XML Schema, where the ellipsis represents removed XML Schema

Where the client is able to request the data directly, the application running on the client need not implement an XML parser as the data can be directly requested. This is ideal for mobile applications where devices wish to use standardised metadata structures, and thus know in advance where their data is going to be stored. XPath expressions can be pre-programmed such that no XML needs to be implemented on the client side.

3.3.3 RXPP Scalability

In order to test the scalability of RXPP some simple tests were performed using JAVA 6 and Saxon 8. The time and memory required to parse each RXPP request was measured to determine the overheard of just the RXPP request. The time taken to parse the XML document and locate the node using XPath was measured. Finally, the total memory required for total operation was recorded. The tests as shown in Table 3.1 were generated on a Pentium M 1300MHz with 512Mb RAM.

From the results in Table 3.1, the average parsing time and memory usage for the RXPP request is 7.4 milliseconds and 30,107 bytes respectively. The average time required for parsing the requested XML file and executing the XPath expression is 321 milliseconds. These results demonstrate that the overhead of the RXPP is minimal even on a low end

Table 3.1 RXPP Parsing Times and Memory Requirements

XML File	File Size (bytes)	Request PT (ms)	Request Memory (bytes)	Nodes Selected	XML and XPath PT (ms)	Total Memory (bytes)
m3uOut	44,110	13.9	34,776	1	1771	720,344
m3uOut	44,110	20.6	30,608	1	103	309,448
choicetest	3,297	0.7	29,072	1	28	269,048
bugs	11,148	14.9	29,072	0	25	257,336
classical	8,859	0.7	29,072	1	15	250,496
EDGI911DI	29,438	0.7	29,072	1	40	289,544
m3uOut2	602,641	0.8	29,072	1	270	1,250,728
Average		**7.4**	**30,107**		**321.7**	**478,134**

PT - Parsing Time

Pentium M class machine. Results could be significantly improved by utilising XML/data caching techniques and executed on high-end server machines.

3.4 Usage Scenarios

To generate some results for RXPP, a client and server application was written in JAVA (Appendix A provides implementation details). The client sends RXPP requests and adds resulting XML fragments (into its appropriate place) in a JTree for visual inspection. The client application simulates the user navigation though the XML, retrieving only the nodes which are selected by a mouse click which results in an RXPP request being sent for that element. A screenshot of the demonstration JAVA application is given in Figure 3.13.

Due to the nature of RXPP whereby results vary from user to user and application to application, results will be given by way of scenarios.

3.4.1 Scenario One: Navigation

Navigation is an important feature, especially in upcoming standards such as MPEG-21, where a ‘menu’ is delivered to the user via the Digital Item. This is advantageous as a Digital Item may contain many versions (i.e., different languages, device descriptors) and choices of descriptors, all within a single document.

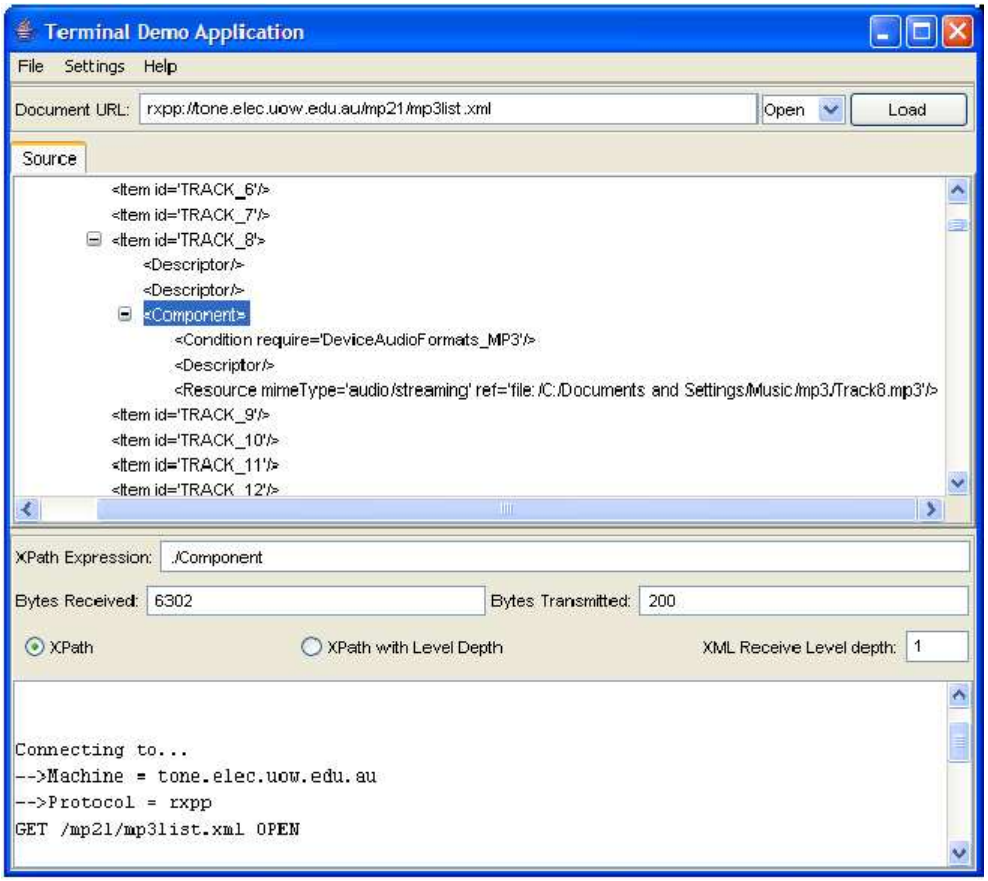

Figure 3.13 Example Screenshot of client application

In this scenario, a user has a large collection of MP3s and keeps a catalog of them within a DID. A sample of the format used for the MP3 items within this DID is illustrated in Figure 3.14. The user queries the DID when searching for a particular song. The DID also contains additional information linked to each song, providing additional information such as lyrics, images, or links to other resources on the Internet.

The example XML file selected represents a typical MPEG-21 [30] Digital Item Declaration (DID). An MPEG-21 DID is an XML file capable of incorporating large amounts of meta-data. The DID in this case is a user's MP3 playlist, which describes all the MP3 tracks, including MPEG-7 descriptors of each track, possible bitrate variations (if available), and the MP3 file location. The XML playlist is originally 613,777 bytes in size and contains descriptions of 214 MP3 tracks.

The diagram in Figure 3.15 gives a visual example of the RXPP protocol messaging and how

```
<Item id="TRACK_4">
  <Descriptor>
    <Statement mimeType="text/plain">
               The Best Song</Statement>
  </Descriptor>
  <Descriptor>
   <Statement mimeType="text/xml">
     <mpeg7:mpeg7>
      <mpeg7:Description xsi:type="CreationDescriptionType">
        <mpeg7:CreationInformation id="track-4">
          <mpeg7:Creation>
             <mpeg7:Title
                 type="songTitle">The Best Song</mpeg7:Title>
             <mpeg7:Title type="albumTitle">Group</mpeg7:Title>
             <mpeg7:Creator>
              <mpeg7:Role
                  href="urn:mpeg:mpeg7:RoleCS:2001:PERFORMER"/>
              <mpeg7:Agent xsi:type="PersonType">
               <mpeg7:Name>
                 <mpeg7:FamilyName>Last</mpeg7:FamilyName>
                  <mpeg7:GivenName>First</mpeg7:GivenName>
               </mpeg7:Name>
              </mpeg7:Agent>
             </mpeg7:Creator>
             <mpeg7:CreationCoordinates>
              <mpeg7:Date>
               <mpeg7:TimePoint>1994</mpeg7:TimePoint>
              </mpeg7:Date>
             </mpeg7:CreationCoordinates>
          </mpeg7:Creation>
          <mpeg7:Classification>
           <mpeg7:Genre href="urn:id3:cs:ID3genreCS:v1:X">
             <mpeg7:Name>Rock</mpeg7:Name>
           </mpeg7:Genre>
          </mpeg7:Classification>
        </mpeg7:CreationInformation>
      </mpeg7:Description>
    </mpeg7:mpeg7>
   </Statement>
 </Descriptor>
 <Component>
  <Condition require="DeviceAudioFormats_MP3"/>
   <Descriptor>
    <Statement mimeType="text/plain">The Best Song</Statement>
   </Descriptor>
   <Resource mimeType="audio/streaming"
             ref="song.mp3"/>
 </Component>
</Item>
```

Figure 3.14 A sample portion of a single MP3 entry as used in the catalog when stored in the DID

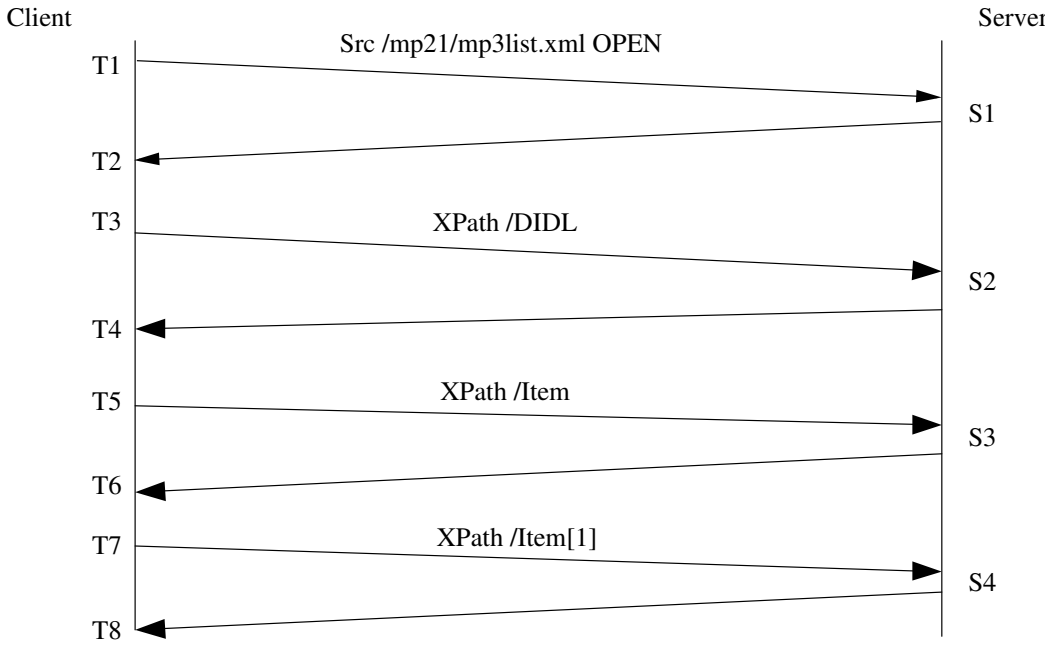

Figure 3.15 Protocol Messaging Example

the XML fragments are delivered over time. At time T1 (at the terminal) the user requests the XML file and at S1, the server processes this information. The XML file is opened and the XML root node followed by its child nodes are sent. The client receives this XML at time T2. In this example there was only one element, `<Item>`, at this level in the XML. The user selects `<Item>` (represented at time T3) and the next level of XML is received at time T4. Now the user has enough information to see the list of music tracks, as defined in the XML, and chooses the Item, at T5, which contains the information about the first track. This process continues until the user has received the desired information.

This process illustrates that a user can navigate through a document, selecting which items are to be retrieved. Furthermore, within each item (i.e., music track), the user may decide to skip the unwanted MPEG7 descriptors, and only retrieving the song name and file location, thus saving the download of unwanted data (Figure 3.14 illustrates the amount of MPEG7 data for each track).

3.4.2 Scenario Two: Random Access

Following on from the previous example (see Section 3.4), now consider that the user is playing these tracks on a mobile device with limited bandwith and local storage space. In this case, downloading the entire XML document is not feasible. The user decides to request random tracks to play for one hour, including the descriptors and MP3 locations. Initially the user obtains the entire list (partial view shown in Figure 3.16) of the tracks using the same process as in Figure 3.15. With this information, the software on the mobile device can use RXPP to randomly request the descriptors for the next random track.

```
<DIDL>
 <Item>
   <Item id="Media">
     <Item id="test.m3u">
       <Descriptor>
         <Statement mimeType="text/plain">
            DI Media from M3U Playlist=test.m3u
         </Statement>
       </Descriptor>
       <Item id="TRACK_1"/>
       <Item id="TRACK_2"/>
       <Item id="TRACK_3"/>
       <Item id="TRACK_4"/>
       <Item id="TRACK_5"/>
       <Item id="TRACK_6"/>
...
```

Figure 3.16 A sample portion of a MP3 list from the DID

The average download and upload per track in this example is 450 bytes and 55 bytes respectively. In this scenario, over the period of one hour, the terminal has uploaded a total of 1,100 bytes and only downloaded 11,737 bytes. If this was all the user intended on doing, and considering that the original document size is 613,777 bytes, this is a significant saving of downloaded data. This demonstrates the effectiveness of RXPP especially when random access is required.

3.4.3 Scenario Three: Every Element is Retrieved

In the event that every element within the remote XML document is selected and retrieved, the total downloaded and uploaded data exceeds that of traditional file transmission tech-

niques. The extra upload and downloaded data is a result of the protocol, where small overheads are added to the original XML. For example, each RXPP XPath request contains the text 'XPATH' followed by the XPath expression (which can range from one to many characters).

However, a user may prefer the additional overhead of RXPP to gain the advantage of retrieving data progressively instead of waiting for the entire XML document to be retrieved. Devices in bandwidth limited environments benefit from progressive download as it provides 'instant access' to data contained within the remote XML document, with less waiting times. For example, the user may decide progressive download is advantageous as they could have access to parts of the document immediately rather than wait for the entire initial download. Furthermore, some mobile devices may not have the capacity to hold the entire document.

If a user did want to retrieve the entire document, then RXPP can still be used. By setting the level depth to -1 for the XPath expression of '/', the client can retrieve all the descendant nodes of the XML root node (i.e., the entire document).

3.5 Conclusion

This Chapter introduced a novel protocol, called RXPP, to provide users with the ability to navigate through a remote XML document, only retrieving the relevant parts of the document, rather than retrieve the entire XML document. This technique is advantageous for many XML documents that may contain irrelevant or redundant information, since this irrelevant information does not need to be retrieved, saving both time and bandwidth. This Chapter demonstrated that using RXPP allows users to randomly access data (using XPath to select nodes) within the remote XML document, without the need to receive the information up to that position in the XML document. RXPP has advantages over existing XML delivery techniques as shown in Table 3.2.

RXPP is beneficial for mobile devices where processing power is limited and are restricted to low speed networks. Since the XPath expressions are executed server side, this also saves XML processing on the mobile client and eliminates the need for the client to possess the

Table 3.2 Comparison of RXPP with other technologies

	RXPP	Millau	XMill	XFI[1]	XOP	XStream	TeM	FCMD[2]
User Defined Queries	Y	N	N	N	N	N	N	Y
Retrieve up to n branch levels	Y	N	N	N	N	N	N	N
XML Fragments	Y	N	N	Y	Y	Y	Y	N
Random Access	Y	N	N	N	N	N	N	Y
Supports Modify	Y	N	N	N	N	N	Y	P
Supports Delete	Y	N	N	N	N	N	Y	P
Supports Insert	Y	N	N	N	N	N	N	P
Partial Downloads	Y	N	N	N	N	N	N	Y

1. XML Fragment Interchange
2. Fragment Caching for Mobile Devices
P - Partial Support

entire XML document.

Where structure of the remote document is unknown (e.g., not valid to a standard), then RXPP navigation provides an effective method to browse through the structure of the document, whilst skipping sub-branches of unwanted nodes.

Chapter 4

Remote Exchange of XML Documents

4.1 Introduction

Chapter 3 described a technique to provide a simple and lightweight method to allow a mobile client to pull fragments of an XML document from a remote device. However, there can be often many users wishing to use a query to request fragments. Alternatively, users may wish to collaboratively edit documents with many other users. In a collaborative editing situation, this requires a full two-way protocol which can update versions of a remote document as well as receive updates. Lack of research contributions into the two-way delivery of XML to mobile devices is possibly due to recent mobile devices becoming more capable in terms of memory and processing. Furthermore, most users possibly assume that the entire XML document is required before it can be opened.

This Chapter presents a novel two-way exchange protocol, defined wholly in XML, used for the exchange of XML fragments. The advantages of using an XML-based protocol allows applications to utilise existing XML tools. This Chapter then demonstrates the effectiveness of the protocol when used within collaborative editing environments (e.g., when many users contribute to the same document).

Finally, this Chapter details a version of RXEP which has been accepted into MPEG-B [8], to compliment the already standardised Fragment Update Units already defined in MPEG-7 [7].

4.2 Remote XML Exchange Protocol

Chapter 3 described a simple technique (called RXPP) which allow devices to pull only the required fragments of XML documents from a remote source. However, there are many cases where users may wish to go beyond retrieval and edit or update remote XML documents; this type of exchange requires a two-way protocol providing client to server and server to client data exchange. To be capable of a full two-way exchange, additional commands need to be added to RXPP which provide users with operations crucial to editing, such as the ability to add, update, modify and delete XML fragments. This new protocol provides a remote exchange of XML data (i.e., both devices may send request and receive responses) and will be referred to as the Remote XML Exchange Protocol (RXEP). Throughout this Chapter, the sender or remote device refers to the device which contains the XML document, and the client or receiver refers to the device which is retrieving parts or all of the remote XML document.

Extending upon the requirements for RXPP in Section 3.2.1, the following additional requirements for RXEP are as follows:

1. The protocol must be easily extendable to allow other applications to build upon the protocol (i.e., allow third parties to add extra commands to the protocol);
2. The protocol must allow queries to be sent to the remote device to determine the fragment(s) of XML to be delivered back to the device initiating the request;
3. The protocol must provide the ability to deliver one or more relevant fragments within a single response to an RXEP request. For example, a simple query may match many nodes, and all the matched nodes should be packaged together within a single response;
4. The protocol must provide support for the editing of remote XML documents. For example, providing clients with the ability to manipulate the document with commands such as add, update, insert and delete (providing the client has appropriate permissions to do so);

5. The protocol must provide support for a remote device to deliver (push) updated fragments to all connected clients. This is required where an XML document may be changing data and/or structure over time; the sender must be able to notify clients of these changes by sending the changes as appropriate XML fragments to update the clients;

6. The protocol must provide the ability to enable efficient compression of the protocol and its contained data where possible; and

7. The protocol must provide the ability to send a single request to receive a 'stream' of incoming fragments. The fragmentation method and timing of fragment delivery may be requested by the client, or the decision left solely up to the sender.

From the above RXEP requirements, it was necessary to create two RXEP packet types for transmission: RXEP packets sent from the client to the remote device, and RXEP packets delivered by the remote device to the client, referred to as RXEP request and RXEP response packets, respectively.

The literature reviewed in Chapter 2 showed that XML compression is very efficient for small XML documents and that XML Schemas provide easy methods for extending existing schemas. For example, in XML Schema an existing schema is extended by importing the schema and adding new declarations. An example XML Schema `testSchema1` defining a type called `Test` which contains a sequence of two children is shown in Figure 4.1. Figure 4.2 then shows a new XML Schema which imports the schema from Figure 4.1 using the import command. A new element, `myNewTest`, is thus created which extends the `Test` type from Figure 4.1 with the extension command to add an additional element to the sequence. This new element, `myNewTest` now has a sequence of three child elements.

Given the advantages of XML and XML Schema, such as platform independence, human readability and easy extension of existing XML Schemas, the RXEP protocol will thus be described using XML Schema, where the instantiation of the protocol is written in XML (to satisfy requirements 1 and 6). Furthermore, the implementation of RXEP in XML and XML Schema allows applications to use already existing XML tools (such as DOM and

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="testSchema1"
            xmlns="testSchema1">
     <xsd:complexType name="Test">
        <xsd:sequence>
             <xsd:element name="A" type="xsd:integer"/>
             <xsd:element name="B" type="xsd:integer"/>
         </xsd:sequence>
     </xsd:complexType>
</xsd:schema>
```

Figure 4.1 An example XML Schema which defines the Test type which contains a sequence of two elements

SAX parsers) and powerful XML query languages such as XQuery and XPath, to create and manipulate RXEP packets.

Fragment Caching for Mobile Devices [40] defines a technique to provide mobile DOM access to XML documents. However, this technique does not offer users the ability to modify or insert new data which is vital for use in a collaborative environment. The mobile DOM concepts from [40] are advantageous and will be used in RXEP as the basis for its open mode operation.

Although MPEG-7 TeM/BiM [7] provides the ability to send fragments to a client, it is incapable of inserting an XML node between already existing nodes (see Chapter 2.3.6). MPEG-7 TeM/BiM also utilises the concept of a 'deferred node', which instructs the receiver that there is a node, but its contents are unavailable and will be sent at a later time. This type of approach could be used to provide the receiving side of navigation through a remote XML document, but it is inefficient when dealing with XML documents which contain a flat structure. For example, the simple XML fragment in Figure 4.3 shows an example of using MPEG-7 with the deferred nodes concept. In this case, the `A` through to `F` node names have been transmitted, even though only the `G` node required. Although this was a simple example, if the node names were lengthy, and if there were many nodes, this transmits unnecessary information.

Since MPEG-7 TeM/BiM is incapable of inserting a node between two nodes (which is a vital operation for editing XML documents), and with the additional overhead of deferred nodes,

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
           xmlns:test="testSchema1">
  <xs:import namespace="testSchema1"
                  schemaLocation="testSchema1.xsd"/>
  <xs:element name="myNewTest">
   <xs:complexType>
      <xs:complexContent>
        <xs:extension base="test:Test">
          <xs:sequence>
             <xs:element name="C" type="xs:integer"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 4.2 An example XML Schema creating a new element `myNewTest` extending the Test type from the XML Schema in Figure 4.1 by adding an additional element to the sequence

```
<Alphabetic>
  <A> (Deferred)
  <B> (Deferred)
  <D> (Deferred)
  <E> (Deferred)
  <F> (Deferred)
  <G>
     ...
```

Figure 4.3 An example XML fragment illustrating the nodes required in MPEG-7 TeM to receive only the `G` element

RXEP instead extends upon the MPEG-7 TeM [7] concepts and provide its own protocol commands.

The following Scenario will be used throughout this Chapter:

Scenario 4.2.1 *An RXEP enabled server contains several books (book1.xml, book2.xml, etc), each represented as an XML document. An example of* `book1.xml` *is illustrated in Figure 4.4. All the images in the book are embedded within the XML document (encoded as base64). A user may wish to perform the following actions without receiving the entire XML document:*

1. *Browse the structure of the XML document to reveal the number and titles of the chapters;*
2. *Select a chapter to read, only downloading the material within that chapter;*
3. *Select a chapter but only retrieve the major sections of that chapter, not receiving the subsections unless selected;*
4. *Query the entire document to receive only parts of the book, i.e., receive the introduction of every chapter;*
5. *The user is the author of the book, and discovers a mistake in one of the sections and corrects this mistake by uploading the correction, and not the entire document;*
6. *Request the entire book to be sent, but wishes to segment the book into small pieces and delivered; and*
7. *Navigate through the various chapters of a book, skipping unwanted chapters.*

```
<Book name="book1">
  <Chapter title="Chapter1">
    <Section title="Section1">
      <Para>This is some text.</Para>
      <Para>Some more text.</Para>
      <Figure name="fig1"> ... </Figure>
      ...
    </Section>
    ...
  </Chapter>
  <Chapter title="Chapter2">
     ...
  </Chapter>
</Book>
```

Figure 4.4 Example XML format of book1.xml located on an RXEP enabled server

4.3 RXEP Packets

Since RXEP uses XML and XML Schema, a schema syntax which will encapsulate all the RXEP commands is required. As mentioned in Section 4.2, RXEP will be composed of

request and response packets. Figure 4.5 shows the RXEP root XML Schema syntax which defines a choice between either an RXEP response or an RXEP request packet.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="RXEP">
   <xs:complexType>
     <xs:choice>
      <xs:element name="Request" type="requestType"/>
      <xs:element name="Response" type="responseType"/>
     </xs:choice>
   </xs:complexType>
 </xs:element>
...
</xs:schema>
```

Figure 4.5 Root XML Schema syntax for used for creating RXEP packets

4.3.1 RXEP Request Packets

RXEP request packets are created by the client in order to request one or more XML fragments to be delivered back to the client from the remote device. The RXEP request reuses the RXPP commands as described earlier in Chapter 3, such as Src and XML-Pull, but replaces the XPath command by a more generic 'Query' command. Using a generic query command has the advantage of not locking RXEP into only one query language, but instead act as a carrier for any existing (or future) query languages (e.g., XPath and Xquery). RXEP further adds a new 'stream' command to satisfy requirement number 7. The XML Schema syntax for the RXEP request commands are shown in Figure 4.6. Each RXEP Request is only allowed to specify one `Src` command to ensure that all commands within a single RXEP request apply to one XML document source. This restriction is achieved by using a single sequence of the Src element (which can only occur once) and a Choice which may allow many occurrences (to allow the other commands to appear many times). For example, it is possible to construct an RXEP request specifying one Src command, and multiple XML-Pull commands if the user knows the structure (e.g., from previous received fragments) to save sending multiple requests.

```
<xs:complexType name="requestType">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:element name="Src" type="srcType" minOccurs="0"
                maxOccurs="1"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Query" type="queryType"/>
      <xs:element name="XMLPull" type="xmlpullType"/>
      <xs:element name="Stream" type="streamType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

Figure 4.6 RXEP request syntax for the RXEP schema used for creating RXEP request packets

4.3.1.1 RXEP `Src` Command

Before RXEP requests can be made on a remote document, the specific remote XML document needs to be specified; this will be referred to as the source (`Src`) command, which is used to select an XML document when initiating an RXEP connection to a remote device. The XML Schema syntax of the source command is shown in Figure 4.7. The Query, XMLPull and Stream commands (see later) are only available after a connection is created. As with RXPP, RXEP also defines Open and Closed modes to specify how the connection between the sender and receiver is maintained (see Section 3.3.2.4).

```
<xs:complexType name="srcType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="OpenMode" type="xs:boolean"
                    use="optional"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Figure 4.7 RXEP XML Schema syntax for the Src type

A source command may only appear once within a single RXEP packet to ensure that a single request operates only on one XML document. An example of a source command requesting the file `book1.xml` from the location `/examples/xml/` is shown in Figure 4.8. Once a `Src` command is sent, it is assumed that all proceeding RXEP packets with the source command absent (only applicable in open mode) apply to the last defined source command.

If the connection type is in 'closed' mode, then the source command is required on all RXEP requests.

```
<RXEP>
 <Request>
   <Src openMode="true">/examples/xml/book1.xml</Src>
 </Request>
</RXEP>
```

Figure 4.8 Example RXEP request packet specifying to open an RXEP connection in open mode and specifying the source of `/examples/xml/book1.xml`

4.3.1.2 RXEP `Query` Command

An RXEP `Query` command is used to perform queries upon the remote XML document as specified by the source command. The XML Schema syntax for an RXEP query command is shown in Figure 4.9.

Referring back to Scenario 4.2.1, there are two possible ways that a user may wish to use a query: 1) to receive nodes and contents of its descendant nodes; or 2) request nodes (and its immediate child nodes) for later navigation. In order to distinguish between the two modes, the RXEP query command will specify a `navMode` boolean attribute; 'true' for navigation mode where the specified node and is its immediate child nodes are received, and 'false' where the `levelDepth` attribute specifies the number of descendant children. RXEP query allows any query language to be used, specified by the `qLang` attribute, provided that the remote device supports that query language. By default, the RXEP query specifies XPath due to its ability to specify a 'path' in XML documents as well as its simple query functions.

Additionally, an optional `levelDepth` attribute may be used to specify the number of levels of descendants which are also to be returned. RXEP uses the same notation for `levelDepth` as in RXPP such as: -1 specifies the retrieval of the entire subtree, 0 specifies the retrieval of only the specified node, and a positive integer *I* to specify the retrieval of *I* levels of descendant nodes (see Section 3.3.2.2 for more information).

An RXEP request may contain several `query` commands to perform multiple queries on the remote XML document, and receive the responses within a single RXEP response packet. For

```
<xs:complexType name="queryType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute ref="levelDepth" use="optional"/>
      <xs:attribute name="navMode" type="xs:boolean"/>
      <xs:attribute name="qLang" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:attribute name="levelDepth"
              type="xs:nonNegativeInteger"
              default="1"/>
```

Figure 4.9 XML Schema syntax for the RXEP Query type

example, referring to Scenario 4.2.1, to request all Introduction sections and corresponding text and images throughout the entire book (i.e., `//Section[@title="Introduction"]`) as well as all the major section nodes (and not their descendants) for all chapters which have an introduction (i.e., `//Chapter[Section/@title="Introduction"]`), the request as shown in Figure 4.10 can be constructed.

An RXEP query can offer significant savings in terms of the size of the data transmitted, since only the XML fragments matching the given query are delivered, rather than the entire document.

```
<RXEP>
  <Request>
   <Query navMode="false" levelDepth="-1">
        //Section[@title="Introduction"]
   </Query>
   <Query navMode="true">
       //Chapter[Section/@title="Introduction"]
   </Query>
  </Request>
</RXEP>
```

Figure 4.10 Example RXEP request packet specifying two RXEP query commands to request all Section nodes and its contents (i.e. levelDepth of -1) where the title is 'Introduction' as well as all the `Section` nodes for all Chapters containing an Introduction section, where results from both commands will be contained within a single RXEP response

4.3.1.3 RXEP `XMLPull` Command

The `XMLPull` command provides navigation through a remote XML document on a node-by-node basis. Using XML-Pull commands, a user can view the structure and data of an XML document, one node at a time. The RXEP XML-Pull commands are the same as RXPP (see Section 3.3.2.3):

- Next - requests the next sibling node from the current node;
- Expand - requests to 'expand' or 'open' the current node, which returns the first child node;
- Back - requests to return the previous sibling node from the current node. This may be useful if the previous sibling has not been cached (i.e., on a limited mobile device); and
- Up - requests to return the parent node of the current node.

RXEP XML-Pull is only available in open mode since the server remembers the 'current' node in order to apply further XML-Pull commands. The corresponding XML Schema syntax for the RXEP XML-Pull command is shown in Figure 4.11. The XML-Pull defines a string which is restricted by an enumeration to only allow the Next, Up, Expand and Back commands. The XML-Pull command, like the query command, may optionally allow the `levelDepth` attribute to specify the level of descendant nodes to be retrieved.

For example, and referring to Scenario 4.2.1, if the user is at the first `Chapter` node and decides that the they do not want to read any of chapter 1. The user may issue an XML-Pull next command (as shown in Figure 4.12) to request the next sibling node (chapter 2) without downloading any of the data contained within chapter 1.

As with the query command, several XML-Pull commands may be present within a single RXEP request, as shown in Figure 4.13. When multiple XML-Pull commands are present, the results from each command are all packaged together in order within a single RXEP response.

```
<xs:complexType name="xmlpullType">
 <xs:attribute name="Command" type="xmlPullCommandType"
               use="required"/>
 <xs:attribute ref="LevelDepth" use="optional"/>
</xs:complexType>

<xs:simpleType name="xmlPullCommandType">
 <xs:restriction base="xs:string">
   <xs:enumeration value="Next"></xs:enumeration>
   <xs:enumeration value="Up"></xs:enumeration>
   <xs:enumeration value="Expand"></xs:enumeration>
   <xs:enumeration value="Back"></xs:enumeration>
 </xs:restriction>
</xs:simpleType>
```

Figure 4.11 XML Schema syntax for the RXEP XML-Pull type

The XML-Pull technique can offer advantages, especially on mobile devices, where there may be limitations on the processing capabilities or the bandwidth of the network connection. If a device had limited memory or small screen size, it could simply request one node at a time. By using the `next` command, a user can avoid downloading potentially large, unwanted portions of the XML document, saving time (less time spent waiting for download) and money (some carriers charging on a per kilobyte basis).

```
<RXEP>
  <Request>
   <XMLPull>Next</XMLPull>
  </Request>
</RXEP>
```

Figure 4.12 Example RXEP request packet specifying an RXEP XMLPull command to return the 'next' node from the current node

4.3.1.4 RXEP `Stream` Command

The final technique for requesting fragments of an XML document is the RXEP `Stream` method. The stream command provides the user with the ability select a method for streaming (or delivering) a remote XML document. Streaming an XML document refers to how the sender continuously sends XML fragments without requiring continuous RXEP requests for each fragment. The stream command may detail the streaming of an entire document or

```
<RXEP>
  <Request>
   <XMLPull>Next</XMLPull>
   <XMLPull>Next</XMLPull>
   <XMLPull>Expand</XMLPull>
  </Request>
</RXEP>
```

Figure 4.13 Example RXEP request packet specifying multiple RXEP XMLPull commands, specifying the nodes to be returned are the 'next' node, followed by the new 'next' node, and then followed by the 'expand' command

simply just branches or sub-branches of the remote XML document. Figure 4.14 shows the XML Schema syntax for representing an RXEP stream command.

```
<xs:complexType name="streamType">
  <xs:sequence>
    <xs:any namespace="##any" processContents="lax"/>
  </xs:sequence>
  <xs:attribute name="location" type="xs:string"
                use="optional"/>
  <xs:attribute name="method" type="xs:string"
                use="optional"/>
</xs:complexType>
```

Figure 4.14 XML Schema syntax for the RXEP Stream type

The stream command specifies an optional `location` attribute which is used to select the node(s) which will be the starting point(s) for streaming XML fragments. If the location attribute is absent, then it is assumed that the root node of the document is selected as the starting point for the streaming.

The `method` attribute of the stream command specifies which XML fragmentation technique, and the timing for the delivery of the specified XML fragments. If there is no method selected for the stream command, then the remote peer will determine the best method to fragment the subtree and stream the XML fragments to the client. For example, this may be the technique which is currently used in MPEG-B, where the author has already predetermined the XML fragments and the timing for the delivery of the XML fragments.

If the requesting peer requires additional control over how the remote XML document is to

```
<RXEP>
  <Request>
   <Stream location="/Book/Chapter[1]"/>
  </Request>
</RXEP>
```

Figure 4.15 Example RXEP request packet specifying a RXEP Stream command, using the location attribute to instruct the sender that the starting point for streaming the XML fragments is the first chapter of a book specified by the XPath `/Book/Chapter[1]`

be fragmented and the order and timing of the XML fragments, the `method` attribute can select an appropriate technique, provided it is supported by the sender.

For example, if the user in Scenario 4.2.1 wishes to stream all of chapter one, an RXEP stream request is sent using an XPath expression to specify the starting position for streaming as shown in Figure 4.15.

The stream command can save bandwidth by streaming fragments from only the requested branches of an XML document, rather than sending the entire document.

4.3.2 RXEP Response Packets

When an RXEP request is sent to the device containing the remote XML document, it processes the request and generates an RXEP response packet. This RXEP response packet is delivered back to the requesting device, where the contained XML fragment(s) are added to the requesting device's local version of the XML document. The requesting device's local version of the XML document is only a partial view of the remote XML document.

The XML Schema syntax for the RXEP responseType is shown in Figure 4.16. The RXEP response defines a choice of the possible RXEP response operations, where each RXEP response will contain at least one one response operation, but may contain several operations if required. The defined RXEP operations are Add, Delete, Update and Insert.

Throughout this Section, the partial view (after some navigation) of the remote XML document from Figure 4.4, as shown in Figure 4.17, will be used as the starting point for the client before the RXEP responses are applied.

```
<xs:complexType name="responseType">
 <xs:sequence>
   <xs:element name="RXEPConfig" type="configType"
               minOccurs="0"/>
   <xs:choice minOccurs="1" maxOccurs="unbounded">
     <xs:element name="Add" type="addType"/>
     <xs:element name="Delete" type="deleteType"/>
     <xs:element name="Update" type="updateType"/>
     <xs:element name="Insert" type="insertType"/>
 </xs:choice>
 </xs:sequence>
</xs:complexType>
```

Figure 4.16 RXEP XML Schema syntax for the RXEP responseType used for creating RXEP response packets

```
<Book name="book1">
  <Chapter title="Chapter1"/>
  <Chapter title="Chapter2"/>
</Book>
```

Figure 4.17 Example client side XML document after navigation of the remote XML document in Figure 4.4

4.3.2.1 RXEP `Add` Command

The RXEP `Add` command is used by the sender to instruct the receiver to add (or place) the contained XML fragment to the location specified on the receiver's local XML document. Figure 4.18 shows the XML Schema syntax for the `addType` which is used for the add command.

```
<xs:complexType name="addType">
   <xs:sequence minOccurs="0">
      <xs:any namespace="##any" processContents="lax"
              minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="location" type="xs:string"/>
</xs:complexType>
```

Figure 4.18 RXEP Add response declaration for the RXEP schema used to notify to the receiver that the contained XML fragment is to be added to the location as defined in the location attribute

For example, and referring to Scenario 4.2.1, Figure 4.19 shows an RXEP add response from an RXEP request on `/Book/Chapter[1]` with a level depth of -1. The RXEP response

informs the receiver to 'add' the contained fragment to the location `/Book/Chapter[1]`. The new XML document after the RXEP add command is shown in Figure 4.20.

```
<RXEP>
  <Response>
    <Add location="/Book/Chapter[1]">
     <Section title="Section1">
       <Para>This is some text.</Para>
       <Para>Some more text.</Para>
       <Figure name="fig1"> ... </Figure>
      ...
    </Section>
  </Response>
</RXEP>
```

Figure 4.19 Example of RXEP Add response instructing the client to add the contents of Chapter 1 to the `/Book/Chapter[1]`

```
<Book name="book1">
  <Chapter title="Chapter1">
      <Section title="Section1">
       <Para>This is some text.</Para>
       <Para>Some more text.</Para>
       <Figure name="fig1"> ... </Figure>
      ...
    </Section>
  </Chapter>
  <Chapter title="Chapter2"/>
</Book>
```

Figure 4.20 Updated XML document after the RXEP Add response from Figure 4.19

4.3.2.2 RXEP `Delete` Command

The RXEP `Delete` command is used by the sender to instruct the receiver to delete (or remove) the node at the specified location on the receiver's local XML version. When a node is deleted, all its descendant nodes are also deleted. Figure 4.21 shows the XML Schema syntax for the `deleteType`, which is used for the delete command.

The delete response is used where the server has removed a node, and informs the client to delete the corresponding node in its local version to keep the clients document synchronised with the senders version.

```
<xs:complexType name="deleteType">
    <xs:attribute name="location" type="xs:string"/>
</xs:complexType>
```

Figure 4.21 RXEP Delete response declaration for the RXEP schema used to notify to the receiver that the XML node at the location as defined in the location attribute is to be deleted

For example, and referring to Scenario 4.2.1, Figure 4.22 shows an RXEP delete response that informs the receiver to delete the node at the location `/Book/Chapter[2]` on the local XML document. The updated XML document is shown in Figure 4.23.

```
<RXEP>
  <Response>
    <Delete location="/Book/Chapter[2]"/>
  </Response>
</RXEP>
```

Figure 4.22 Example of RXEP Delete response instructing the client to delete the second chapter

```
<Book name="book1">
  <Chapter title="Chapter1"/>
</Book>
```

Figure 4.23 Updated XML document after the RXEP Delete response from Figure 4.22

4.3.2.3 RXEP `Update` Command

The RXEP `Update` command is used by the sender to instruct the receiver to update the node at the specified location on the receiver's local XML document. Figure 4.24 shows the XML Schema syntax for the `updateType`, which is used for the `update` command.

For example, Figure 4.25 shows an RXEP update response that requests the receiver to update the node at the location `/Book/Chapter[1]` on the local XML document. The updated document is shown in Figure 4.26.

4.3.2.4 RXEP `Insert` Command

The RXEP `Insert` command is used by the sender to instruct the receiver to insert the new node at the specified location on the receiver's local XML document. The insert command defaults to inserting the node after the specified node, unless the insertBefore attribute is set

```
<xs:complexType name="updateType">
  <xs:sequence minOccurs="0">
    <xs:any namespace="##any" processContents="lax"
            minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="location" type="xs:string"/>
</xs:complexType>
```

Figure 4.24 RXEP Update response declaration for the RXEP schema used to notify to the receiver that the location as defined in the location attribute is to be updated with the XML fragment contained within the RXEP response

```
<RXEP>
  <Response>
    <Update location="/Book/Chapter[1]">
      <Chapter title="newChapter1"/>
    </Update>
  </Response>
</RXEP>
```

Figure 4.25 Example of RXEP Update response instructing the client to update the first chapter

```
<Book name="book1">
  <Chapter title="newChapter1"/>
  <Chapter title="Chapter2"/>
</Book>
```

Figure 4.26 Updated XML document after the RXEP Update response from Figure 4.25

to true, then it is inserted before the specified node. Figure 4.27 shows the XML Schema syntax for the `insertType`, which is used for the `insert` command.

```
<xs:complexType name="insertType">
  <xs:sequence minOccurs="0">
    <xs:any namespace="##any" processContents="lax"
            minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="location" type="xs:string"/>
  <xs:attribute name="insertBefore"
                type="xs:boolean" use="optional"/>
</xs:complexType>
```

Figure 4.27 RXEP Insert response declaration for the RXEP schema used to notify to the receiver that the contained XML fragment is to be inserted as defined in the location and insertBefore attributes

For example, Figure 4.28 shows an RXEP insert response to request the receiver to insert the node contained within the response packet between the nodes `/Book/Chapter[1]` and `/Book/Chapter[2]` on the local XML document. The updated XML document is shown in Figure 4.29.

The `Insert` command introduces significant savings in terms of data transmitted as compared to TeM, which does not include insert functionality. To achieve an 'insert' in TeM, the entire subbranch would need to retransmitted.

```
<RXEP>
  <Response>
    <Insert location="/Book/Chapter[1]"
            insertBefore="false">
      <Chapter title="Chapter1.5"/>
    </Update>
  </Response>
</RXEP>
```

Figure 4.28 Example of RXEP Insert response, instructing the client to insert the new node after the node specified by the XPath `/Book/Chapter[1]`

4.3.2.5 RXEP `RXEPConfig` Command

The `RXEPConfig` command allows the server to send configuration information to the client. Configuration information includes the URIs that are used throughout the document and the element name defining the global element of the XML document. Figure 4.30 shows

```
<Book name="book1">
  <Chapter title="Chapter1"/>
  <Chapter title="Chapter1.5"/>
  <Chapter title="Chapter2"/>
</Book>
```

Figure 4.29 Updated XML document after the RXEP Update response from Figure 4.25

the XML syntax for the `RXEPConfig` command.

```
<xs:complexType name="configType">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="uri">
      <xs:complexType>
        <xs:attribute name="nameSpace"
                      type="xs:string"/>
        <xs:attribute name="prefix"
                      type="xs:string"/>
        <xs:attribute name="xsdLocation"
                      type="xs:string"/>
      </xs:complexType>
    </xs:element>
   </xs:choice>
 </xs:complexType>
```

Figure 4.30 XML Schema syntax for the RXEPConfig type

Each `uri` element contains three attributes and their definitions are as follows:

1. nameSpace - defines the Namespace for the defined URI;
2. prefix - defines the prefix associated with the Namespace; and
3. xsdLocation - defines the location for the XML Schema document.

Although this definition is mainly used for the binarised RXEP (see Chapter 5) to define all the XML Schemas used with the XML document, an RXEP server may decide to use it where the namespace declarations are defined on nodes, other than the root node such that the client can locate the XML Schema for a given prefix. For example, Figure 4.31 illustrates a simple XML document which defines the definition for the `ns2` prefix on the root's first child node. In this example, if the node `ns2:node1` was selected as the result of a query, the server

could construct the RXEP Response as shown in Figure 4.32. This response now gives the client information about the `ns2` namespace.

```
<ns1:top xmlns:ns1="ns1:2006"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="ns1:2006 ns1_def.xsd" >
    <ns2:child xmlns:ns2="ns2:2006"
               xsi:schemaLocation="ns2:2006 ns2_def.xsd">
        <ns2:node1>some data</ns2:node1>
    </ns2:child>
</ns1:top>
```

Figure 4.31 Example XML document containing a prefix definition on a node other than the root node

```
<RXEP>
  <Response>
    <RXEPConfig>
      <uri nameSpace="ns2:2006" prefix="ns2"
       xsdLocation="http://some.remote.server/ns2_def.xsd"/>
    </RXEPConfig>
    <Add location="/ns1:top/ns2:child">
      <ns2:node1>some data</ns2:node1>
    </Add>
  </Response>
</RXEP>
```

Figure 4.32 Example RXEP response with defining an RXEPConfig

4.3.3 RXEP XPath Locators

RXEP uses XPath locators to determine where within the client's local version of the document the RXEP response should be applied. XPath locators are used to define precisely which nodes are used for RXEP, however, XPath relies on the fact that for the current XPath locator the nodes specified do not alter position. This is acceptable for RXPP and RXEP when navigation is used to retrieve nodes up to a given node, since the children are retrieved before continuing through the next level of the XML structure. However, since RXEP allows users to query the remote XML document and receive various fragments from multiple locations throughout the document, there is the possibility that the exact placement of the node, with respect to the whole document, cannot be determined.

For example, consider the remote XML document as shown in Figure 4.33. If a user sent a

```
<a>
  <f/>
  <p>
     <t>
        ...
     </t>
     <h>
        ...
     </h>
     <t>
        ...
     </t>
  </p>
</a>
```

Figure 4.33 An example XML document

server an XPath query of `/a/p/t[2]`, Figure 4.34 shows the corresponding RXEP response using a standard XPath locator. If this was the first received RXEP packet (i.e., no previous navigation) or no information is known about its structure, the only information that can be determined from the RXEP response is that there is a root node `a`, with a child of `p`, with a child of `t`. From the XPath locator, it is also determined that `t` is the second occurrence, however, there is no way to know if there are other nodes between the two `t` nodes. Figures 4.35 (a) and (b) both demonstrate two possible XML documents that could both satisfy the XPath expression `/a/p/t[2]`, but the order is different. Thus, just using the XPath locator does not provide enough information to allow exact placement of the XML fragment. If an XML Schema is provided, there may be enough information for exact placement (i.e., using minOccurs and sequences), but lack of placement information remains a problem for documents without a schema, or for documents linking to schemas where this information is not apparent.

```
<RXEP>
  <Response>
    <Add location="/a/p/t[2]">
     ...
    </Add>
  </Response>
<RXEP>
```

Figure 4.34 An Example RXEP response resulting from a query `/a/p/t[2]`

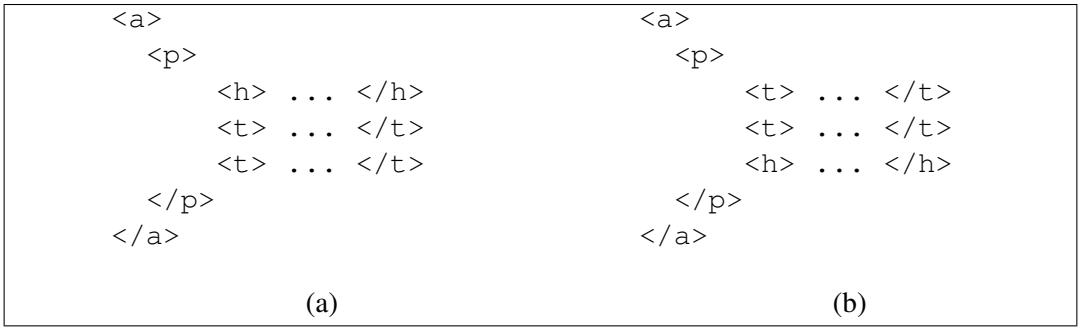

Figure 4.35 Two examples of valid XML documents which could be interpreted from the XPath expression /a/p/t[2] even though they have a different order

If the client was only interested in the data returned from the query and not its position with respect to the entire document, then exact placement of the XML fragments do not pose a problem. However, if a client requires the exact structure of the document to be replicated; two possible methods will now be discussed.

4.3.3.1 Client Requests More Information

If a client received a node and did not have any information about the surrounding nodes, it could request an XPath in navigation mode on each parent node. For example, a user opens an RXEP connection to a remote device, and requests an RXEP query which returns the response as in Figure 4.34. The client thus far only knows about the root node `a`, and requests the children for the root node `a` and its child node `p`, by constructing an RXEP query set in navigation mode, as shown in Figure 4.36. The RXEP response from the request in Figure 4.36 is shown in Figure 4.37. Now, the client has enough information to add the response from Figure 4.34.

```
<RXEP>
  <Request>
    <Query location="/a" navMode="true"/>
    <Query location="/a/p" navMode="true"/>
  </Request>
<RXEP>
```

Figure 4.36 An Example RXEP request to retrieve the child nodes of `/a` and `/a/p`

Unfortunately, this approach is not efficient if the remote XML document has a structure which contains many child nodes and multiple descendants also with many children. This

```
<RXEP>
  <Response>
    <Add location="/a">
      <f/>
      <p/>
    </Add>
    <Add location="/a/p">
      <t/>
      <h/>
      <t/>
    </Add>
  </Response>
<RXEP>
```

Figure 4.37 An Example RXEP response to the request in Figure 4.36, instructing placement of the XML child nodes to the parent nodes `/a` and `/a/p`

would result in many (possibly unwanted) nodes, just for the exact placement of a node.

4.3.3.2 RXEP XPath Locators

An alternate approach for exact placement of a node, as compared to requesting all children of the parent back to the root node (as in Section 4.3.3.1), is to provide additional placement information along with the fragment. This requires information such as the child location, with respect to its sibling nodes, as well as the number of sibling nodes. Any nodes not retrieved could be marked as 'not retrieved', creating a document with the structure as shown in Figure 4.38. If the client was then interested in the nodes that were not retrieved, it could just issue a request only on the parent node using RXEP query, configured to use navigation mode.

In order to recreate the exact structure, and reusing the XPath concept, a slight modification to XPath can be incorporated, which will be referred to as an RXEP XPath Locator. To precisely add the fragment in the exact location the number of nodes needs to be known to insert placeholders for nodes which exist but are not received, as well as the current node location (represented as `[node position, total nodes]`).

When using an RXEP XPath locator in an RXEP response, the node name as well as the extension is used, for example, the RXEP XPath locator of `/a/p[2,2]` instructs the client

to create a `p` node as the second child of node `a`.

To illustrate the use of RXEP XPath locators, imagine a client has received the RXEP response as shown in Figure 4.39, and the document as shown in Figure 4.38 would be created utilising the information provided by the RXEP Xpath Locator (`/a/p[2,2]/t[3,3]`). If the client then wanted the previous sibling node, it could create an RXEP XPath locator of `/a/[2,2]/[2,3]` when constructing an RXEP query request.

```
<a>
  [not retrieved]
  <p>
     [not retrieved]
     [not retrieved]
     <t> ... </t>
  <p>
</a>
```

Figure 4.38 Example of a local version of an XML document, resulting from the query `/a/p/t[2]`

```
<RXEP>
  <Response>
    <Add location="/a/p[2,2]/t[3,3]">
     ...
    </Add>
  </Response>
<RXEP>
```

Figure 4.39 Example RXEP response using the RXEP XPath locator of `/a/p[2,2] /[3,3]`, to provide the client with precise node placement

4.4 RXEP XML Fragmentation Strategies

RXEP provides methods for requesting and delivering XML fragments from a remote XML document. RXEP provides the user with three methods of accessing XML fragments: navigation, queries and a combination of navigation and queries.

4.4.1 Fragmentation by Navigation

Navigation is especially advantageous when there is no knowledge of the incoming XML document ahead of time; this may be the case if there is no available schema describing the

current XML, or the XML document does not implement a known standard (such as MPEG-21). Fragmentation by navigation is the same as described in RXPP (see Section 3.2.5), except RXEP requests and responses are used.

4.4.2 Fragmentation by Queries

In addition to navigation, another technique for XML fragmentation is to dynamically create fragments in response to a query. Within RXEP, the client issues the query via an RXEP request and receives the result via an RXEP response. Creation of a query is possible when a client has knowledge on the format or structure of the XML document; such information may be retrieved from the following:

- An XML document may be valid to an XML Schema. For example, an XML Schema contains information about the structure and data types which are possible in an XML document valid to that schema;
- The structure and format may be inferred utilising information as defined in a standard (i.e. MPEG); and
- Guessing the structure and datatypes from already retrieved data. For example, a client may have performed navigation through part of the XML document and might assume that further nodes of the document follow the same format.

XML fragmentation using the query approach requires a client to construct a query, utilising whatever information is available, and constructs an RXEP request. For example, Figure 4.40 shows an example remote XML document. Figure 4.41 shows an RXEP query request created knowing that there are `Book` nodes, where each book node has a `Name` node.

After the server receives the query from the client (shown in Figure 4.41), the server executes the query locally on the specified XML document, sending this result back to the client as one or many fragments depending on the query and data queried. Figure 4.42 shows the RXEP response to the RXEP request from Figure 4.41.

```
<Library>
 <Book>
   <Name>Book1</Name>
      ...
 </Book>
 <Book>
   <Name>Book2</Name>
      ...
 </Book>
 <Book>
   <Name>Magazine1</Name>
     ...
 </Book>
</Library>
```

Figure 4.40 An example of a remote XML document containing a library of books (only the Name elements are shown in this example, missing data is represented by ellipsis)

```
<RXEP>
 <Query>
     //Book[contains(Name, 'Book')]
 </Query>
</RXEP>
```

Figure 4.41 Example RXEP request looking for all Book nodes where the Name contains the string 'book'

```
<RXEP>
  <Add location="/Library/Book[1,3]">
    <Name>Book1</Name>
  </Add>
  <Add location="/Library/Book[2,3]">
    <Name>Book2</Name>
  </Add>
</RXEP>
```

Figure 4.42 Example RXEP response from the RXEP request in Figure 4.41

The result from a query provides enough information for exact placement of the fragment(s) within the client's local version of the XML document, as the query may return many fragments from the original document.

Server side processing of queries are advantageous to some mobile clients, where they may not have enough memory and processing power to execute complex queries, and the entire XML document is not needed locally on the mobile device to perform the query. Since the client is receiving fragments based on a query, there is a reduction on the amount of irrelevant XML received which saves both time, bandwidth and resources. For example, and referring to Scenario 4.2.1, suppose the book contained five chapters, where each chapter was one megabyte in size (including images). If the query returned all of chapters three and four, then the mobile client would have only received approximately two megabytes of data from a total of five megabytes, which saved downloading three megabytes of unwanted data.

4.4.3 Fragmentation by Navigation And Queries

The final fragmentation strategy involves a mixture of both the navigation and query techniques previously mentioned in Sections 4.4.1 and 4.4.2. Utilising a mixture of navigation and queries is beneficial in situations where the XML or the structure of the XML may not be known upfront. In this case, a client can first navigate through the first part of the XML structure inferring how the rest of the document is possibly structured. The client can then construct a query after navigation to attempt to retrieve any other relevant data.

For example, Figure 4.43 illustrates a client after having navigated the first level of the `music` node, and through the first `track` node. Since the root level was retrieved, it can be seen that there are three child nodes (music, photos and movies), however, in this example, the user is only interested in music. Thus the XML for the videos and movies are not retrieved. For example, if the photos and movies portion of the document was 60 percent of the total size, the client has instantly saved downloading 60 percent of unwanted data.

The client might assume that the rest of the remote XML document has a similar structure for all its music tracks, and the client can now ask for all `track` nodes where the artist is 'A. Artist', using the following XPath query:

```
<album>
  <music>
     <track>
         <songid>5</songid>
         <artist>A. Artist</artist>
         <title>Music track 1</title>
         <link>/Artist_track1.mp3<link>
     </track>
     <track /> (not received)
     <track /> (not received)
       ...
  </music>
  <photos /> (not received)
  <movies /> (not received)
</album>
```

Figure 4.43 An example of a local version of an XML document after navigation has revealed the format of the first track node

`/album/music//track/artist[text()='A. Artist'].`

For example, if all tracks by 'A. Artist' represented only 14 out of 100 tracks, it has further saved downloading 86 percent of the music section.

4.5 Delivery of RXEP Messages

Since RXEP is defined in XML (which is a text only format), it has the advantage that RXEP messages can be encapsulated within standard transport mechanisms (i.e., HTTP or SOAP). A block diagram of two devices capable of RXEP is illustrated in Figure 4.44.

For example, an RXEP request and its corresponding RXEP response, delivered using a HTTP message are illustrated in Figure 4.45 and Figure 4.46, respectively. Examples with SOAP messages are shown in Figures 4.47 and 4.48.

4.6 RXEP Experimental Results

Due to the reliance of user selections or application specific selections, results will be given by way of a few simple scenarios to demonstrate the effectiveness and advantages of RXEP. Results have been generated using an RXEP implementation using JAVA (for more informa-

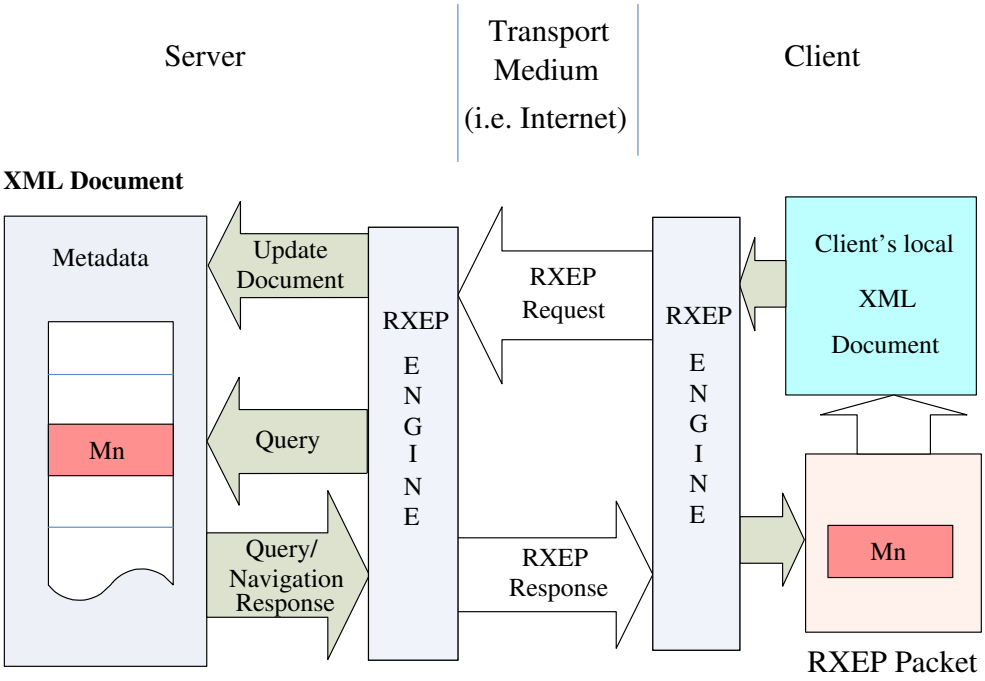

Figure 4.44 Block diagram illustrating communication between two RXEP enabled devices

```
POST /rxep HTTP/1.1
Host: localhost:81
Content-Type: text/xml; charset=utf-8
Content-length: 118

  <?xml version="1.0"?>
  <RXEP>
     <Src>/FavouriteMusic.xml</Src>
     <Query>/Album/Music//Track</Query>
  </RXEP>
```

Figure 4.45 Example RXEP request asking for all music tracks using HTTP as the transport mechanism

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 180

  <?xml version="1.0"?>
 <RXEP>
  <Add location="/Library/Book[1,3]">
    <Name>Book1</Name>
  </Add>
  <Add location="/Library/Book[2,3]">
    <Name>Book2</Name>
  </Add>
</RXEP>
```

Figure 4.46 Example RXEP response to the RXEP request from Figure 4.45 using HTTP as the transport mechanism

```
POST /examples HTTP/1.1
Host: localhost:81
Content-Type: application/soap+xml; charset=utf-8
Content-length: 381

<?xml version="1.0"?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <RXEP>
     <Src>/FavouriteMusic.xml</Src>
     <Query>/Album/Music//Track</Query>
  </RXEP>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.47 Example RXEP request encapsulated within a SOAP message delivered via HTTP

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: 466

<?xml version="1.0"?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
   <RXEP>
     <Add location="/Library/Book[1,3]">
       <Name>Book1</Name>
     </Add>
     <Add location="/Library/Book[2,3]">
       <Name>Book2</Name>
     </Add>
   </RXEP>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.48 Example RXEP response to the RXEP request in Figure 4.47, encapsulated within a SOAP message delivered via HTTP

tion see Appendix A). The RXEP software implementation consists of two separate applications, a server and a client. The client provides the user with a simple interface to select remote XML documents (using a URL) and the XML is presented to the user as a JTree. As a user selects a node, the client constructs the appropriate RXEP request and transmists the request to the server. As the client receives the RXEP responses, it updates the JTree to reflect the newly updated local XML instance.

4.6.1 Scenario One: RXEP on a Mobile Device

In this scenario, a baseball fan is at a live baseball match and wishes to settle a discussion with his friend about some players and teams from 1998. The baseball fan uses his mobile device to connect to the baseball statistics which are stored in an XML document. Having previously searched the baseball statistics, the baseball fan has knowledge of the structure and layout of the statistics.

The baseball fan and his friend want to know which players in the national league have scored more than 120 runs and played more than 150 games. The baseball fan uses RXEP

to connect to the 1998 baseball statistics XML document and constructs an RXEP Query as illustrated in Figure 4.49. To ensure that all the details of each returned player is retrieved, the `levelDepth` attribute is set to `-1`.

```
<RXEP>
 <Request>
   <Src>/1998bbstats.xml</Src>
   <Query levelDepth="-1">
//bb:PLAYER[compare(ancestor::LEAGUE/LEAGUE_NAME,"Nati
onal League")=0 and GAMES > 150 and RUNS > 120]
   </Query>
 </Request>
</RXEP>
```

Figure 4.49 Example RXEP Query request to query a remote baseball statistics XML document

The server matched four players from the RXEP Query, and constructs an RXEP response which is delivered back to the baseball fan's mobile device. This RXEP response is illustrated in Figure 4.50.

The size of the 1998 baseball statistics is 763,831 bytes. The RXEP Query in Figure 4.49 is only 222 bytes which represents data uploaded by the baseball fan's mobile device. The total data download (i.e., from the RXEP response) to retrieve the desired information is 3,135 bytes.

The overheads added by the RXEP protocol for the upload and download in this example are 105 bytes and 512 bytes, respectively. The total overheads of the protocol are therefore 617 bytes. Thus, for the small overheads added by RXEP, the user has saved 760,474 bytes of downloaded data.

4.6.2 Scenario Two: Navigation

In this scenario a user is interested in determining which countries have a total GDP greater than 900,000. The user uses RXEP to connect to an XML document (mondial-3.0.xml) containing information from CIA World Factbook, the International Atlas, and others. The user constructs an RXEP Query with an XPath expression of `//country[@gdp_total > 900000]`. The server matched nine countries, and sends back an RXEP Response as

```
<RXEP>
  <Response>
    <Add location="/SEASON/LEAGUE[1]/DIVISION[2]/TEAM[6]/
PLAYER[8]">
        <SURNAME>Mcgwire</SURNAME>
        ...
    </Add>
    <Add location="/SEASON/LEAGUE[1]/DIVISION[1]/TEAM[1]/
PLAYER[16]">
        <SURNAME>Jones</SURNAME>
        ...
    </Add>
    <Add location="/SEASON/LEAGUE[1]/DIVISION[2]/TEAM[3]/
PLAYER[17]">
        <SURNAME>Biggio</SURNAME>
        ...
    </Add>
    <Add location="/SEASON/LEAGUE[1]/DIVISION[2]/TEAM[1]/
PLAYER[23]">
        <SURNAME>Sosa</SURNAME>
        ...
    </Add>
  </Response>
</RXEP>
```

Figure 4.50 Example RXEP Response from the RXEP Query in Figure 4.49. For brevity, removed data is represented by ellipsis

illustrated in Figure 4.51. In this example, only the first level of each country is retrieved, where further elements will be downloaded via navigation.

```
<RXEP>
 <Response>
   <Add location="/mondial/country[14]">
    <country id='f0_213' name='France' capital='f0_1510'
     population='58317448' datacode='FR' total_area='547030'
     population_growth='0.34' infant_mortality='5.3'
     gdp_agri='2.4' gdp_total='1173000' inflation='1.7'
     government='republic' gdp_ind='26.5' gdp_serv='71.1'
     car_code='F'/>
    <Name/>
    <province id='f0_17485'/>
...
    <religions  percentage='1'/>
...
    <languages  percentage='100'/>
    <ethnicgroups  percentage='44'/>
    <religions  percentage='70'/>
    <encompassed continent='f0_119' percentage='100'/>
    <border  length='60' country='f0_144'/>
...
   </Add>
   <Add
....
 </Response>
</RXEP>
```

Figure 4.51 Example RXEP Response containing results from a query. Repeated element names are indicated by ellipsis

The initial RXEP response downloaded is 3,094 bytes and the RXEP request is 47 bytes. This now provides the user with a list of country elements and their corresponding attributes. This download is large to do the structure of the XML; there are 16 attributes for each country element.

From this list the user decides to further navigate through the USA's statistics. This now downloads a further 6,248 bytes and uploads a further 50 bytes.

The user now looks at the first level elements of the USA country element. The user comes across the religion elements, and decides it might be interesting to look at all the religions in the USA, and sends another RXEP request. The RXEP request is 50 bytes and the RXEP

Table 4.1 RXEP Parsing Times and Memory Requirements

XML File	File Size (bytes)	Request PT (ms)	Request Memory (bytes)	Nodes Selected	XML and XPath PT (ms)	Total Memory (bytes)
m3uOut	44,110	4.4	528,552	11	601	1,144,992
bugs	11,148	7.0	185,512	0	22	266,672
choicetest	3,297	5.0	172,552	68	19	472,936
classical	8,859	4.8	182,800	4	45	518,592
EDGI911DI	29,438	2.6	181,280	11	17	488,176
m3uOut2	602,641	4.6	181,592	1	222	1,661,272
m3uOut	44,110	4.9	179,224	1	24	633,248
Average		**4.7**	**230,216**		**135**	**740,841**

PT - Parsing Time

response is 438 bytes. The user then decides to view the religions in France to compare with USA, and sends a navigation query on the France element. The RXEP request is 60 bytes and the RXEP response is 580 bytes.

This simple example demonstrates the RXEP navigation combined with RXEP querying. In this case, the user does not need to download the entire document, and can navigate through the result set, downloading only selected elements on-demand. In total, the user has only uploaded 207 bytes and downloaded 10,360 bytes, where the total data transferred is 10,567 bytes. The total data in this case is significantly less than the remote XML file size of 1,784,877 bytes.

4.6.3 RXEP Scalability

In order to test the scalability of RXEP some simple tests were performed using JAVA 6 and Saxon 8. The time and memory required to parse each RXEP request was measured to determine the overheard of just the RXEP request. The time taken to parse the XML document and locate the node using XPath was measured. Finally, the total memory required for total operation was recorded. The tests as shown in Table 4.1 were generated on a Pentium M 1300MHz with 512Mb RAM.

From the results in Table 4.1, the average parsing time and memory usage for the RXEP request is 4.7 milliseconds and 230,261 bytes respectively. The average time required for

parsing the requested XML file and executing the XPath expression is 135 milliseconds. These results demonstrate that the overhead of the RXEP is minimal even on a low end Pentium M class machine. Results could be significantly improved by utilising XML/data caching techniques and executed on high-end server machines. Additionally, since RXEP can be contained in other formats (such as SOAP) additional parsing times would be required to parse the parent container format (e.g. decode the SOAP message).

4.7 Collaborative Editing Using RXEP

Collaborative editing is a process that occurs on a day-to-day basis and increasingly, there are multiple authors editing a single document. One example of collaborative editing is the gaining popularity of 'Wikis' [70], which have become accepted for collaborative editing on the Internet. Applications of Wikis range from software documentation/howtos to online encyclopedias (such as Wikipedia [70]). Programmers also regularly collaboratively edit software code using versioning software (such as CVS [71]). This allows an author to check for updates and compare against their local version before making modifications, to ensure they have the latest version of the file.

Office document formats, such as OASIS OpenDocument [72] and Microsoft Office documents [73] have moved away from the proprietary binary format and begun using XML as the container to store document data. As these document formats are described in XML, it is thus possible to utilise standard XML tools and collaborative editing techniques on these office documents.

RXEP is an ideal candidate for managing the delivery and request of XML documents. This is beneficial since often the entire document is not desired. Additionally, features of RXEP such as add, delete, update and insert provide collaborative editing functionality.

Using RXEP, a client may download parts (or all) of a document, make changes locally, and only upload the parts that have been changed (upload entire fragment or just the differences). For example, a user who wishes to revise a document can create an RXEP request and navigate through the document. On finding an error in the document the user can make changes

and only those changed fragments of XML need be relayed back to the server as an RXEP response to update the original document.

RXEP also provides additional functionality for users on portable devices. For example, users may wish to only retrieve one paragraph at a time to reduce storage requirements, bandwidth usage, or download waiting times. RXEP can thus be used to make browsing of large documents convenient and practical. Strict collaborative editing rules may utilise the tree-based structure of the XML. Techniques such as [74] describe lock-based protocols designed specifically for collaborative editing of XML documents. Utilising these lock-based protocols, branches of the tree can be locked (so no other authors can edit), whilst allowing the other branches to be edited; this allows separate portions of the document to be concurrently edited, while preventing others from updating the sections that are being accessed. Furthermore, some users may not have privileges to access certain parts of a document, and methods such as [75, 76] may be used to restrict views on the server side.

4.7.1 Experimental Results

To demonstrate the effectiveness of RXEP for collaborative editing and delivery, some practical examples are considered. These examples were evaluated using an implementation of RXEP in JAVA. These examples are aimed at investigating the amount of data transferred in the process of viewing/editing particular office documents.

4.7.1.1 Receive XML Documents with RXEP

The first set of results examine RXEP's efficiency when downloading only sections of data. Here, textual representations are compared to the original document size and results are taken from the upload and download traffic totals from the client's device. These results are illustrated in Figure 4.52.

The first text file for comparison is an XML version of Shakespeare's Othello[1]. This file contains little structure and mostly string text, with an original file size of 248,777 bytes. The test was performed by navigating to Act 8->Scene1->1st speaker, changing the speaker name, and uploading the alteration. The text mode resulted in a 1,176 bytes upload and 1,598 bytes

[1]Downloaded from `http://www.ibiblio.org/xml/examples/`

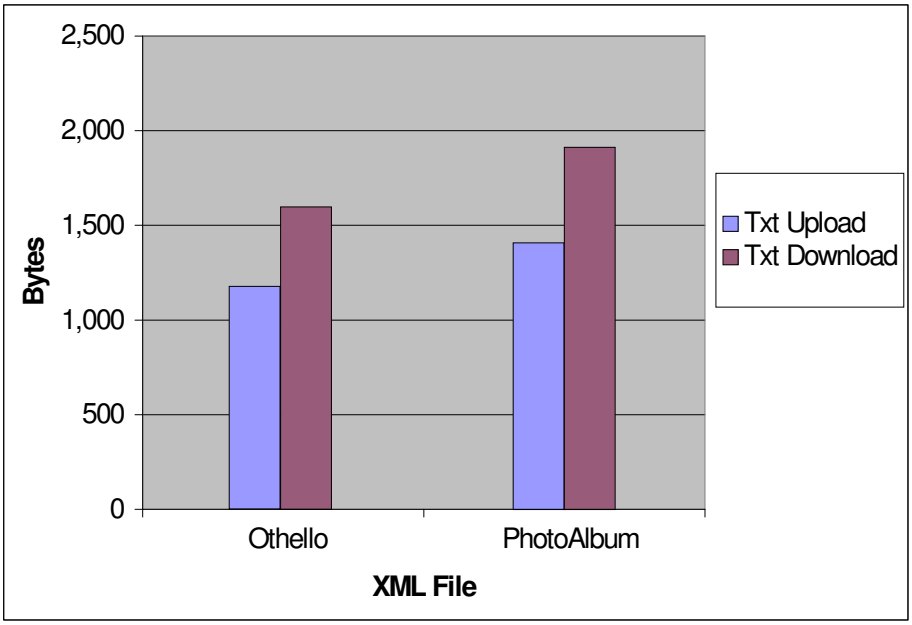

Figure 4.52 Comparison of data uploaded and downloaded for the two tests

download. To progressively receive the text to that position (i.e., downloading the whole file to that point) would have required 207,030 bytes; this demonstrates the bandwidth savings achieved by skipping the unwanted parts of a file.

The second example consists of a photo album represented by a MPEG-21 Digital Item[2]. Initially, this XML file is of length 21,040 bytes. The test navigated to the level of photo descriptors and altered the description of a photo. Using RXEP resulted in a 1,409 bytes upload and 1,914 bytes download, compared to if the client had streamed to the chosen location, which would have required 3,997 bytes.

4.7.1.2 Comparison of Office XML Documents with RXEP

Currently, the two most popular office XML documents are OpenOffice and Microsoft Word which use the opendocument and WordML formats, respectively. This Section compares the advantages of using RXEP with these two document formats.

To obtain results for these tests, both office packages were used to author the same (visually on screen) document and saved as its corresponding XML format. These results are shown

[2]Photo album DID can be found in Appendix C

in Figure 4.53.

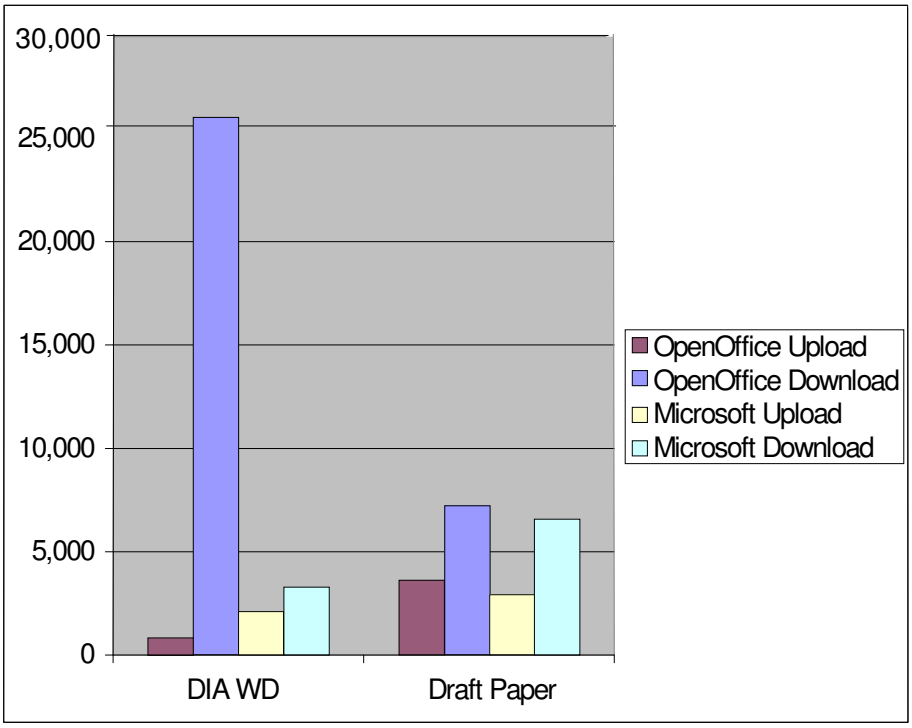

Figure 4.53 RXEP comparison between the collaborative editing tests on two documents using OpenOffice and Microsoft XML documents

The first test document (DIA WD[3]) consists of a MPEG-21 Digital Item Adaptation (DIA) working draft document. The document was converted into XML using both software packages. The test consisted of navigation to the third Section of the document, then navigation to a paragraph, where a mistake is present. The correction was made and the new fragment was then uploaded to the original remote version.

OpenOffice XML required 27,742 bytes download and 836 bytes upload. The MS Word XML required 3,282 bytes download and 2,108 bytes upload.

The second file (Draft Paper[4]) consisted of the first draft of a conference paper, converted using both packages. This test involved navigating to the section describing RXEP and modifying the second paragraph. OpenOffice XML required 7,529 bytes download and 3,723 bytes upload. The Microsoft (MS) Word XML required 6,787 bytes download and 2,910

[3]DIA WD can be found in Appendix C
[4]Draft conference paper can be found in Appendix C

bytes uploaded.

In the first example, it was found that the OpenOffice XML used a much flatter document structure as compared to the MS Word XML format. This flatter structure results in many elements occurring at the body node level, which consequently requires significantly more download as compared to the MS Word XML.

In the second example, the file had fewer major sections than the first file, and the differences were not as significant. Furthermore, the OpenOffice XML was much harder to navigate due to the flatter structure than the MS Word XML. Overall, this demonstrates that document structure can adversely affect RXEP's efficiency.

4.8 Creating a Standardised FRU Solution of RXEP

MPEG-7 standardised a method for the delivery of XML fragments of MPEG-7 XML descriptors known as TeM (for more information see Section 2.3.6). TeM offers a method for delivering XML fragments contained within Fragment Update Units (FUUs), where these FUUs instruct the receiver where to insert/update the fragment of XML or delete the node (and thus descendant nodes). Within MPEG-7, XML fragments are predetermined by the author as well as the timing information. This technique is adequate within streaming environments (i.e., delivered via MPEG-2 Transport Streams), however, with new standards such as MPEG-21, the need for dynamic creation of XML fragments based on user requests has become apparent.

MPEG-21 aims to provide interoperability for a large range of devices (small mobile devices on limited bandwidths to powerful machines with fast connectivity). MPEG-21's interoperability is obtained by way of the flexibility of the Digital Item and the ability to link in the vast number of descriptors available. Consequently, many MPEG-21 Digital Items contain superfluous metadata, which is more than is required for each individual client. For example, consider a Digital Item containing video, audio and appropriate descriptors. If a client chooses to only receive audio at a given bitrate; all the video metadata is irrelevant, as is the metadata for the other audio bitrates.

This is a problem with a general container such as a Digital item and while compression can help reduce the file size (and thus transmission time), it is still inefficient to deliver this redundant data. MPEG-B BiM [77] currently provides a method to deliver compressed fragments (small sections of the XML document), however, there are no dynamic mechanisms to allow only selected fragments of the XML to be sent. This is vital, as clients may need to make changes to selections midstream in the consumption of a Digital Item. For example, a client may acquire access to an external screen and thus need video data to be sent as well as the corresponding video metadata.

Although compression attempts to reduce the amount of data transmitted, the fundamental issue with MPEG-21 is that it is locked into one-way content delivery. Through integration of the concepts of RXEP requests to the already existing FUUs, a standardised flexible solution for the user to possess more control over the delivery of XML documents is facilitated.

4.8.1 MPEG-B Requirements

To introduce the concept of client side control of XML delivery within MPEG, client generated requests must be sent to the sending peer(s). TeM defines a method for delivery of a fragmented XML documents through Fragment Update Units (FUU). It is possible to use these mechanisms for delivery but it is necessary to create a new upstream messaging system. The new, upstream fragment requests will be referred to as Fragment Request Units (FRU). Through the process of defining FRUs, the following requirements [78] were identified:

1. Remote fragment request - FRUs shall provide mechanisms for a user to request fragments of a remotely located XML document. It is important that the user should have granularity in the ability to specify the contents of the fragment to be returned. A user should be able to specify [78]:
    - a particular node to be returned;
    - that a node plus a certain number of its descendants be returned; or
    - that all nodes matching a certain pattern be returned (e.g., All choices within a DID).

2. Remote fragment navigation - FRUs shall provide mechanisms for a user to navigate through a remotely located XML document receiving one or more nodes at a time. For example, a user could request the next sibling, parent, or children of the current node [78].

3. Remote fragment streaming request - FRUs shall provide mechanisms for a user to request that a particular subset of an XML document be streamed in multiple fragments. For example, a user may request a certain subtree to be delivered, with the XML fragmented according to a maximum size (in bytes) per fragment [78].

4. Fragment Request Encoding - FRUs shall provide fragment request mechanisms which may be transmitted in an encoded format. For example, a user could transmit the fragment request in a BiM encoded format [78].

4.8.2 MPEG-B Fragment Request Units

Since the requirements (see Section 4.8.1) are a simplified set of the RXEP requirements, the RXEP requests have been modified to fit within MPEG-B.

MPEG-7 has already defined Fragment Update Units (FUUs) which are used to transport fragments of XML to the receiver. Thus, to complement FUUs, the upstream requests will be called Fragment Request Units (FRUs).

4.8.3 Fragment Request Unit Syntax

As with RXEP, the FRU structure and set of commands is fully described in XML Schema as shown in Figure 4.55.

The XML Schema syntax for the `Src`, `Query`, `XMLPull` and `Stream` are the same as for RXEP (see Section 4.3.1 for further information).

For example, a user wishes to query a remote DI XML document, and the FRU as shown in Figure 4.56 is constructed. The server executes the query on the specified XML document, and constructs the FUU as shown in Figure 4.57.

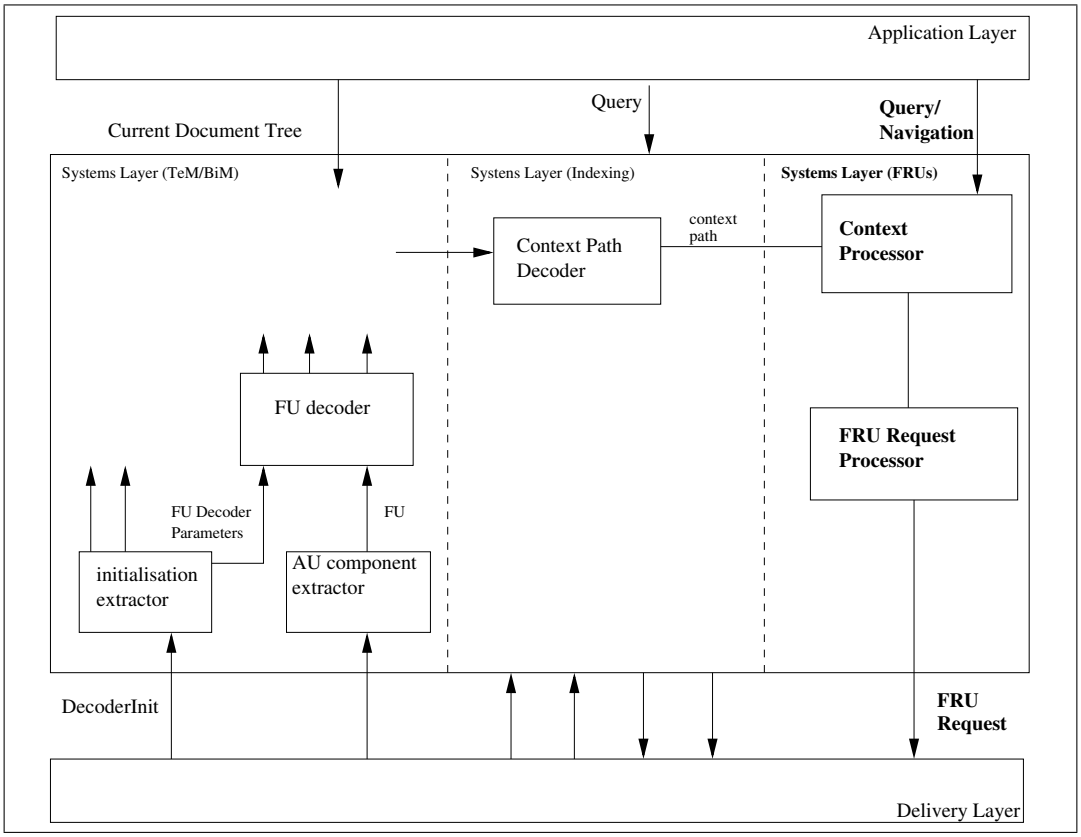

Figure 4.54 Modified MPEG-7 Systems [7] diagram to accommodate the addition of FRUs

```
<xs:element name="FRU">
  <xs:complexType>
    <xs:sequence>
       <xs:element name="Src" type="srcType" minOccurs="0"/>
       <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Query" type="queryType"/>
          <xs:element name="XMLPull" type="xmlpullType"
                      maxOccurs="unbounded"/>
          <xs:element name="Stream" type="streamType"/>
       </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 4.55 MPEG-B Fragment Request Unit Syntax described using XML Schema

```
<FRU>
   <Src>Test_DI.xml</Src>
   <Query>/DIDL/Item[2]</Query>
</FRU>
```

Figure 4.56 Example MPEG-B Fragment Request Unit

```
<FragmentUpdateUnit>
  <FUCommand>addNode</FUCommand>
  <FUContext>/DIDL/Item[2]</FUContext>
  <FUPayload>
    <Descriptor>
      <Statement mimeType="text/plain">
          Two Ton Shoe Music Album
      </Statement>
     </Descriptor>
  </FUPayload>
</FragmentUpdateUnitType>
```

Figure 4.57 Example Fragment Update Unit in response to the Fragment Request Unit in Figure 4.56

4.9 Conclusion

This Chapter presented a novel XML protocol (RXEP), that provides a full two-way exchange of XML documents. Within RXEP, users are able to retrieve fragments and also add fragments to a remote XML document, without requiring the entire XML document to be available locally. Data contained within remote XML documents can be retrieved on the basis of a small initial download. RXEP has advantages over existing XML delivery techniques as shown in Table 4.2.

This Chapter demonstrated that RXEP can provide 'random access' and navigation of remote XML documents through the use of already standardised XPath queries and expressions. Furthermore, RXEP becomes an ideal candidate for collaborative editing of XML documents, especially in the upcoming XML document standards used in OpenOffice and Microsoft Office.

This Chapter further demonstrated the consequence of storing data within flat-structured XML documents when compared to a highly structured design. When RXEP is used for navigation of flat XML document structures, the savings are not as significant. It was shown that highly structured XML documents, such as the structure used in wordML, produces significant savings when using RXEP, as more of the unwanted data and structure can be skipped.

Finally, this Chapter presented an implementation of RXEP (without the collaborative editing

Table 4.2 Comparison of RXEP with other technologies

	RXEP	Millau	XMill	XFI[1]	XOP	XStream	TeM	FCMD[2]
User Defined Queries	Y	N	N	N	N	N	N	Y
Retrieve up to n branch levels	Y	N	N	N	N	N	N	N
XML Fragments	Y	N	N	Y	Y	Y	Y	N
Random Access	Y	N	N	N	N	N	N	Y
Supports Modify	Y	N	N	N	N	N	Y	P
Supports Delete	Y	N	N	N	N	N	Y	P
Supports Insert	Y	N	N	N	N	N	N	P
Protocol Valid to XML Schema	Y	N	N	N	N	N	Y	N
Partial Downloads	Y	N	N	N	N	N	N	Y
Two-Way Editing	Y	N	N	N	N	N	N	N
Protocol can be Extended by user	Y	N	N	N	N	N	N	N
User defined Streaming	Y	N	N	N	N	N	N	N

1. XML Fragment Interchange
2. Fragment Caching for Mobile Devices
P - Partial Support

features) which has been adopted by MPEG-B.

Chapter 5

XML Compression and the Binary RXEP Protocol

5.1 Introduction

The process of augmenting raw data with XML tags inevitably increases the file length, due to the addition of new text and structure information. Depending on the content creator's chosen method for describing the raw data and the application space, the increase in file size may be considerable. To minimise the impact of the size increase, XML compression techniques are required; these provide a reduction in the transmission cost of the structural information, and may be combined with compression of the raw data. Several lossless XML compression techniques were reviewed in Chapter 2.

While XML compression is typically applied to complete, valid XML documents, it can also be applied to fragments of XML. Hence, RXEP (see Chapter 4), which relies on the on-demand transmission of XML fragments, also take advantage of XML compression techniques. The result is an efficient protocol with possible reductions in data size and structural information. This Chapter evaluates current lossless XML compression techniques to determine which of these is best suited for the compression of RXEP and its constituent XML fragments. This Chapter also discusses the creation of two new modifications which can be incorporated into MPEG-B BiM [77] to efficiently compress MPEG-21 DIDs which contain embedded resources and embedded XML.

A novel technique is presented for the combination of extra XML Schema structural information into the XML DOM (called SDOM) to assist in the compression of randomly accessed XML fragments, as well as providing additional controls to provide compressed RXEP data for remote navigation of XML documents.

A new technique is introduced of combining the SDOM and XML schema-based compression techniques to compress RXEP packets and the data contained within the RXEP packet. These compressed RXEP packets further reduce the amount of data downloaded and uploaded from the device. This Chapter also presents a novel technique for the compression of RXEP XPath locators to further increase the compression of RXEP messages.

This Chapter concludes with a measurement of the savings the BinRXEP can achieve when applied to collaborative editing of office documents such as those used in Openoffice and MS Office.

5.2 XML Compression

XML compression techniques aim to reduce the overall size of an XML document and can be either lossless or lossy. To date, many lossless methods of compressing XML files have been proposed and demonstrated [10,46,48,56,57,64,79,80] (further information see Section 2.4).

Lossy algorithms such as that introduced by Cokus et al [48] offer reductions in XML filesize by dropping elements and/or changing the data (i.e., rounding numerical precision etc). Lossy algorithms are not considered in this Section as they destroy information, and require user input or prior knowledge of the format to obtain good compression ratios. This Chapter focuses on a generic compression technique which can be applied to a wide range of XML documents and deliver fragments of these files efficiently from one device to another. In particular, this Chapter focuses on the MPEG-21 Digital Item Declaration (DID), which is the XML representation of a Digital Item. DIDs are a good test for XML compression techniques as they may contain a wide range of embedded content ranging from data and XML to binary (i.e., encoded as base64).

Generally, XML compression techniques can be classified into one of three main approaches: redundancy, schema-based and hybrid methods (see Section 2.4 for further information).

From the literature [52], researchers have determined that there is a point, in terms of file length, at which redundancy compression techniques overtake the efficiency of schema based encoding techniques. The reason behind this is primarily that as files become longer the probability of sequence repetition, and the length of such sequences increases. This is a well known property of redundancy compression techniques (such as Lempel-Ziv [61]).

For example, consider the XML fragment in Figure 5.1. Figure 5.1 shows a simple XML document with three `value` nodes, of which two are identical. In schema based compression, each value node is compressed as it is encountered, i.e., if each value node and data requires 8 bits, then to encode all three would require 24 bits. In contrast, a redundancy compressor would recognise that the `<value>5</value>` string is repeated, and the second occurrence could then be encoded as a pointer to the first string occurrence.

```
<list>
  <value>5</value>
  <value>8</value>
  <value>5</value>
</list>
```

Figure 5.1 Example repetition within XML

5.2.1 Comparison of Lossless XML Compression Techniques

The extensive use of XML for structuring and describing relationships between multimedia content in MPEG-21 DIDs may generate large, verbose XML documents, as seen earlier in Chapter 2. XML compression techniques can be used to reduce transmission times and bandwidth requirements. In order to determine which compression techniques are ideal for MPEG-21 DIDs (and thus XML documents), the following freely available lossless compression approaches are used to compress some sample MPEG-21 DIDs (XML documents): XBIS, Winzip9, BiM, bzip2, XMill, XML PPM and XML Xpress.

A series of test documents were assembled or generated to compare the various XML compression techniques. The contents of each of the testfiles are summarised as follows:

1. An MP3 playlist containing links to 214 songs. Each track is described using MPEG-7 descriptors, such as title, date, genre etc;
2. A CD Audio DID describing all the songs on the CD, including MPEG-21 terminal capabilities and Choices for different bandwidths. The DID additionally contains extras such as interviews and embedded photos;
3. A list of news text with choices between different languages and news sources;
4. An MP3 playlist containing links to 74 songs, using only the MPEG-21 descriptors;
5. A University subject information dataset containing subject information etc, with choices between different file formats;
6. A CD audio DID describing all the songs on the CD, as in Test2, but without the embedded resources;
7. A tourist photo album containing a list of a users photos after a trip;
8. A tourist photo album containing a subset of the photos from Test7, but including MPEG-7 descriptors such as title, creation date, description etc;
9. A simple picture embedded in DID using base64 encoding;
10. A list of movie trailers containing choices between different resolutions for each trailer;
11. A list of classical music with track information and references;
12. A list of movie trailers (similar to Test10) but with MPEG-7 metadata; and
13. A list of a users favourite music tracks;

These XML documents were selected as they represent typical XML descriptors which could be used in future MPEG-21 systems. Results obtained via different compression algorithms can be found in Table 5.1. XML documents which produced errors during the compression process (i.e., unsupported features in the XML) are indicated as 'Did Not Compress' (DNC).

The BiM results were obtained by combining the output binary file with the decoder configuration file, since both are required for successful decompression. BiM was configured to use

Table 5.1 Comparison of compression results on test files 1 - 13 (all units in Bytes)

Test	Original	XBIS	Winzip 9	BiM	bzip2	XMill	XML PPM	XML Xpress
1	662,255	470,971	13,919	DNC	11,083	14,461	19,328	23,433
2	330,705	322,452	176,598	175,833	177,217	176,767	191,229	178,716
3	55,145	36,792	2,471	DNC	2,441	2,200	2,708	3,497
4	45,270	28,627	5,040	DNC	4,854	5,406	4,764	5,213
5	29,478	21,215	2,370	2,045	2,305	2,142	2,260	2,602
6	27,363	19,053	2,491	1,597	2,324	2,333	2,241	2,623
7	21,040	14,084	1,552	1,143	1,376	1,110	1,338	1,605
8	16,967	14,084	1,871	DNC	1,767	1,442	1,481	1,998
9	15,300	15,189	11,135	10,926	11,452	11,097	12,364	11,512
10	10,755	7,894	1,635	1,314	1,702	1,526	1,462	1,752
11	9,060	6,111	1,148	750	1,211	1,026	1,054	1,292
12	4,656	2,553	1,118	DNC	1,144	1,063	946	1,123
13	2,029	1,512	668	336	601	525	464	618

ZLib [67] for string compression and the deferred nodes option (which allows fragmented transmission of XML files) was turned off. WinZip version 9 was used and configured to use maximum compression.

Bzip2 [81] was added to the compression algorithm set since it uses Huffman coding, however, bzip2 loses its advantages when dealing with small serial blocks from XML documents [82]. Bzip2 builds a Huffman tree by assigning the smallest bit combination to the most frequently occurring characters. This mapping is required to be present at the decoder in the same way that BiM requires the decoder config file.

The results in Table 5.1 indicate that BiM performed better than the other compression programs producing the smallest files in all cases except one (for the files it was capable of compressing). This is expected since the majority of these files are mostly structure with a small proportion of data. The only exception to this was for test7 (photo album) in which XMill generated the smallest file. Test7 contains mostly structure but has large repetitive sequences.

XMill performs better then gzip due to it being XML aware, whereas gzip treats the file purely as a stream of bytes. From the results in Table 5.1, XMLPPM performed better than XMill; both performed well for the files BiM was unable to compress.

Test2 and test9 are quite different to the other test files as they contain much more data

content; examples of this are embedded content such JPEG images. The embedded content, represented in base64, will reduce compression ratios as there is less repetition within the base64 content. XML-Xpress did not utilise schema information (due to restrictions on the demonstration version), however it still performed at a level comparable to the other XML aware algorithms.

Overall, schema and hybrid techniques performed better then traditional redundancy techniques. This is not surprising since these DIDs contain a significant amount of structure information, which is well suited to these methods. The main disadvantage of redundancy techniques for XML is that they perform compression at the character level and do not consider the overall structure of the document [6]. From the results shown in Figure 5.1, BiM is best suited for compression of DIDs. BiM's performance, especially on small XML documents, is therefore well suited for XML protocols (such as RXEP).

5.2.2 MPEG-B BiM in-Depth

Since BiM obtained the best compression on the test files in Section 5.2.1, determining why it failed to compress test files 1,3,4,8 and 12 requires an in-depth review of BiM.

The first phase of the BiM encoding process creates a Syntax Tree [7] of the corresponding XML Schema(s). During this phase, other transforms (for more information see [7]) are applied to compact the schema tree and order its nodes while still ensuring the semantics are maintained (i.e., if there is one XML choice with a child node of one XML choice, then they are collapsed into just a single choice).

Since this Section is considering the compression of MPEG-21 DIDs, a sample DIDL Syntax Tree is shown in Figure 5.2. This may be be compared to the fragment of the DIDL XML Schema shown in Figure 5.3 (generated by XMLSpy).

The syntax tree is composed of the following [7]:

- Element declaration nodes - which associate an element name to its type, represented in Figure 5.2 by bold font;

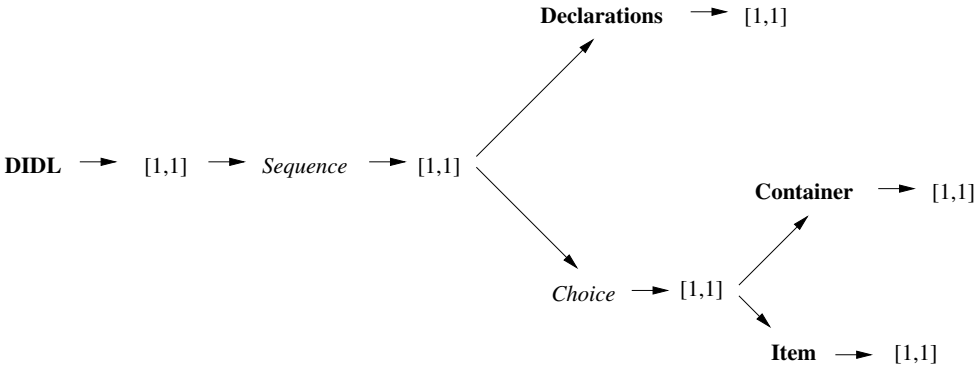

Figure 5.2 Example BiM Syntax Tree generated from the MPEG-21 DIDL XML Schema

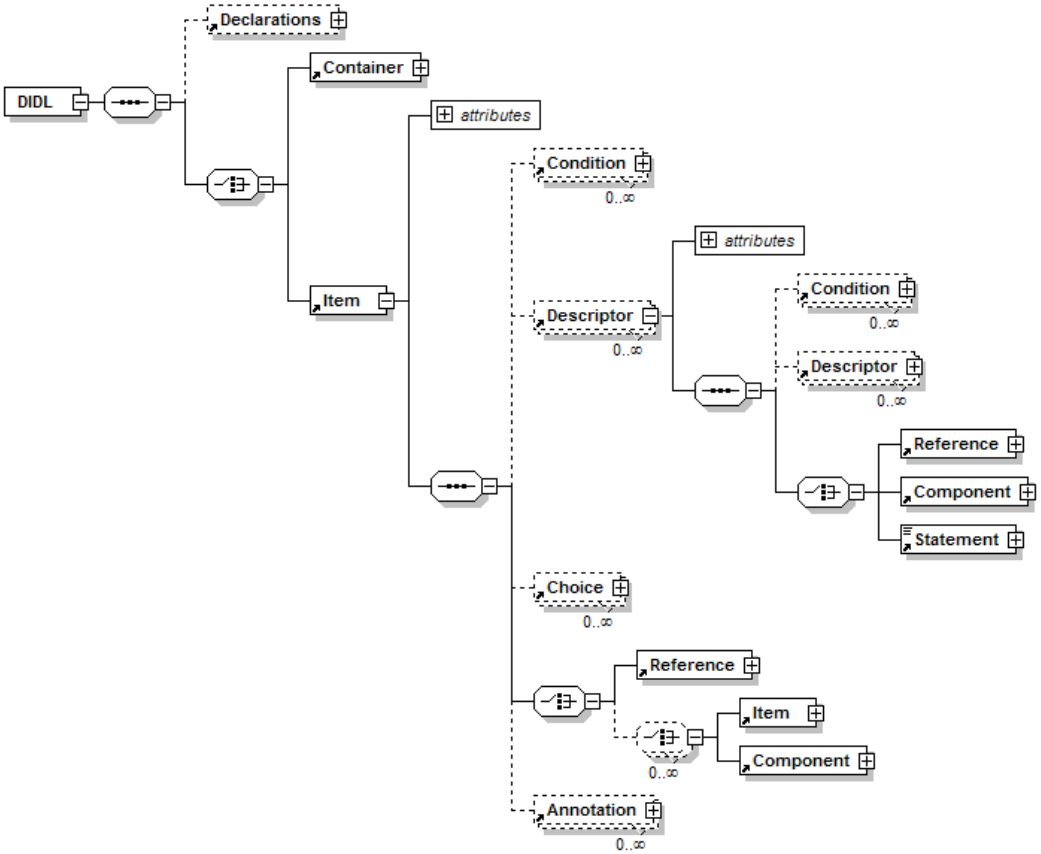

Figure 5.3 Example fragment of the DIDL XML Schema (shown graphically)

- Group nodes - which define the composition group (i.e., choice, sequence or all) represented in Figure 5.2 by italics font; and
- Occurrence nodes - which are derived from the “min” and “max” number of occurrences of the particle [3], represented in Figure 5.2 by the format: [minOccurs, MaxOccurs].

Once the Syntax Tree has been created, generation of the BiM bitstream is simply a matter of simultaneously traversing the syntax tree and the XML, outputting binary codes where appropriate. An example of the binarisation process for the example DID in Figure 5.4 is shown in Figure 5.5. Line numbers indicate the order in which the bits are written. Bold text indicates the bits produced by the compression process.

```
<DIDL xmlns="urn:mpeg:mpeg21:2002:01-DIDL-NS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:mpeg:mpeg21:2002:01-DIDL-NS
    didl.xsd">
   <Item>
     <Descriptor>
       <Reference target="file://c/file.txt"/>
     </Descriptor>
   </Item>
</DIDL>
```

Figure 5.4 Example of a simple DID (valid to DIDL XML Schema) which declares a reference to a file on the local disk

From the XML shown in Figure 5.4, the only namespace defined is the DIDL namespace; in the DIDL XML Schema (at the time of writing), there are fifteen global elements. The XML must start with one of these global elements in order to be valid. The `<DIDL>` tag is the seventh global element in the DIDL XML Schema, and is therefore given the code `0111` (represented by four bits as there are fifteen possibilities, i.e., the first element would have the code `0000`).

XML global elements can only occur once within an XML instance, and as such, no occurrences need to be encoded. The BiM header is written so as to indicate the options used during the encoding process; this is always 8 bits [7]. The next node in the structure is a se-

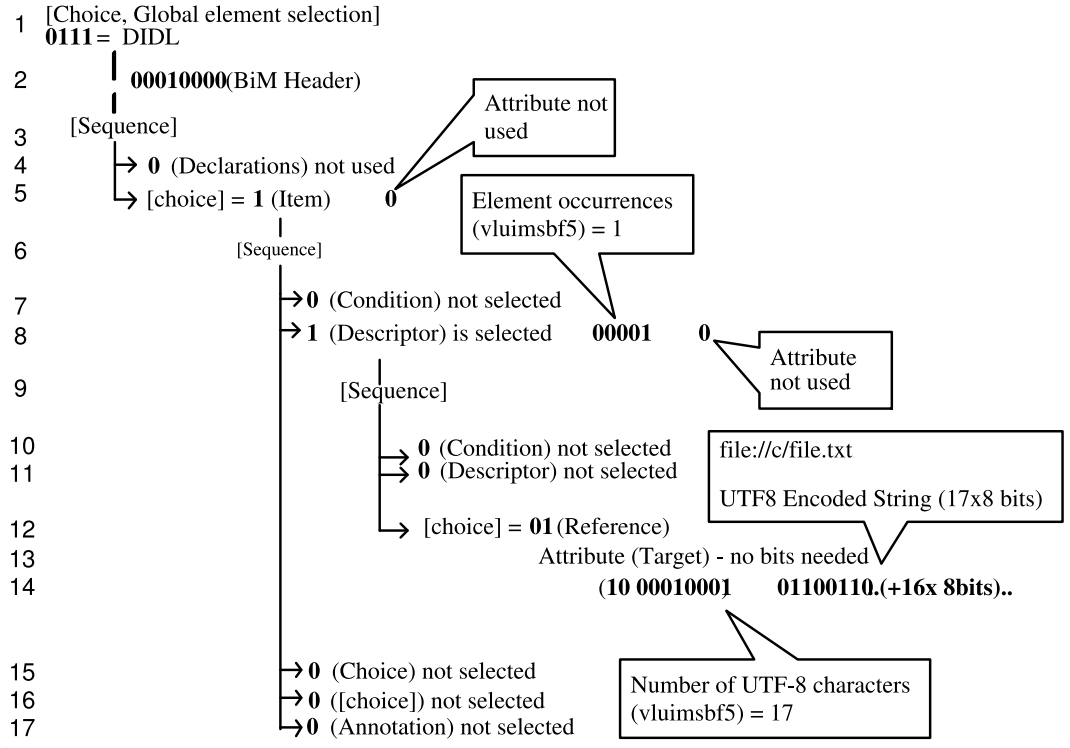

Figure 5.5 Breakdown of the BiM Encoding technique

quence which can only occur once, and thus no bits need be encoded, as this must be present in a valid XML document according to the XML Schema.

Since the `<Declarations>` tag is not present in the XML instance, a zero is written to the stream to indicate this (see line 4 of Figure 5.5). Next, the choice is then used, hence a one is written; the choice can only occur once and thus no occurrence codes are written. There are no attributes defined in the XML and as such, a zero is written.

The `<Item>` tag is present in the XML instance, and in the syntax tree there are two choices (represented by one bit). The `Item` node is the second node in the choice declaration and is assigned the binary code of `1`. The `Item` node can only occur once and hence again no codes are written to represent the occurrences.

The process of following the XML and syntax tree continues until the end of the XML is reached. The original XML file in Figure 5.4 is 309 bytes and the output is 21 bytes, as can be seen by concatenating all the 0s and 1s in bold text from Figure 5.5. Evidently, the

number of bits required is significantly less than the number of bits needed to represent the XML as text. The compression ratio achieved here are typical, however the advantages of the approach even for simple schema are clear.

BiM has the added advantage that the decompression program knows that the XML file is valid to a schema (the XML is required to be valid in order to perform compression) and thus, no additional XML Schema validation needs to be performed after decompression. An additional advantage of this approach is that the structure of the original XML file is preserved - choices, sequences and positions of elements remain. This makes the addition and removal of sections of the XML much easier as the entire file need not be fully decompressed before operating on the XML.

BiM also has the capability to become a hybrid technique through the use of extra codecs to compress various datatypes. One of these codecs defined is the Zlib compression; this is used by BiM to compress strings in the XML file, improving compression ratios by eliminating redundancies in the XML string datatypes. It is important to note that this technique is only applied to individual string values, hence, repetition of strings throughout the document is not considered.

After the BiM encoding is complete, there are two output files; the first is the binary output as generated from the codes, and the second is a decoder config file. The decoder configuration file is used to define all the namespaces and XML Schema file locations as defined in the original XML file. The decoder configuration file tells the decompressor which XML Schemas it needs to locate and load for decompression. From the required XML Schemas, all global elements are read yielding a global elements list. This list defines how many bits should be read from the beginning of the file in order to select the correct global element as used in the original XML file.

5.3 Binarisation of MPEG-21 DIDs

As seen in Section 5.2.1, BiM failed on certain types of MPEG-21 Digital Items. Furthermore, some MPEG-21 DIs did not compress as well as expected.

The BiM compression algorithm relies heavily on the semantic information provided by the XML Schema [7]. This works well with many standard XML documents, obtaining very good compression ratios (as seen in Table 5.1). As a result of this thesis, a number of input documents (refer to Chapter 1.3.5) were created and presented during MPEG meetings to offer solutions to these problems of the BiM technique.

MPEG-21 DIDL differs to many XML Schemas as the DIDL specifies a 'container'. This container-like XML allows users to insert any type of data whilst maintaining a standard way of structuring or holding the data together (using the DIDL). This versatility is achieved through extensive use of XML Schema's *any* element.

An *any* element does not strong-type the element to any of the standard XML data-types; rather it allows arbitrary types of data to be inserted, provided that they adhere to the XML rules. Data contained within an *any* node may be embedded files (such as pictures, audio etc), or even additional well-formed XML. This is not efficiently compressed by current schema based compression techniques such as BiM, since it cannot choose an appropriate data-type compressor (as the datatype cannot be determined from the XML Schema). Thus, the BiM compressor resorts to the use of string compression (redundancy) as a fall-back if data is contained in a text node. If the data is embedded XML, the BiM encoder attempts to encode the XML as normal, but returns an error as the next node after the `any` node cannot be determined from the XML Schema. Furthermore, the contained XML data may not even be valid to an XML Schema, and cannot be compressed using the BiM technique.

The creators of MPEG-21 DIs proposed a solution to a similar problem related to the *any* type. Where an element in the DIDL contains an *any* node, an XML attribute is appended to the node to specify the type of proceeding embedded data. This attribute uses standard mimetypes to describe the data. For example, if there is another XML document contained within the *any* element, the mimetype of text/xml is used. Hence to increase the compression efficiency of MPEG-21 DIDs, these attributes can be exploited to assist the BiM compression algorithm. A new extension has been developed for this thesis, which can be used by BiM to efficiently compress MPEG-21 DIDs.

5.3.1 Embedded Binary Data in MPEG-21 DIDs

Within MPEG-21, there are many possible reasons to embed binary data into DIs (i.e., embedding a JPEG photo along with a description of where/when it was taken). To embed binary data into an XML document the binary content must be first encoded into its textual representation (i.e., using base64 encoding). As discussed earlier (see Section 2.3.2), the consequence of the base64 codec is that encoded files increases by 33 percent when compared to the original file. However, since the XML is being transformed into the binary domain, the data would be more efficiently represented as the original binary.

Unfortunately, the current BiM technique is unable to determine whether the embedded data contained within the *any* element is base64 encoded. Thus, BiM defaults to the default string encoding, rather than to store the base64 as binary which would instantly reduce the size of the embedded data by 33 percent.

In order for the BiM encoder/decoder to utilise this feature, it needs to be extended to become MPEG-21 aware. This requires changes to the encoding of the *any* element, to utilise the encoding information on the `encoding` attribute.

```
<DIDL>
 <Item>
   <Item id="myPhoto">
     <Descriptor>
       <Statement mimeType="text/plain">
          Sample Photo
       </Statement>
     </Descriptor>
     <Component>
       <Resource mimeType="image/jpeg"
                 encoding="base64">
/9j/4AAQSkZJRgABAQEASABIAAD/4RFRRXhpZgAASUkqAAgAAAAKAA4
...
```

Figure 5.6 Example DID containing a description of a JPEG image, as well as the photo contained as an embedded resource, encoded as base64

For example, suppose that a user has an MPEG-21 DID containing an embedded JPEG image together with a brief description as shown in Figure 5.6. The original photo (in binary form) occupies 2,329,753 bytes, and when transformed to base64 and stored in the XML

document this increased to 3,149,483 bytes. When using the standard BiM technique, the BiM compressor does not know that this data is actually base64 encoded and applies the zlib compression technique to the file. The result of this compression is determined by the redundancy within the base64 text of the photo, and in this example, the photo data (base64 encoded and then compressed with zlib) is 2,385,280 bytes. If the BiM MPEG-21 extension is enabled, the encoder now knows that the data is base64 (by examining the `encoding` attribute and recognising that the photo is encoded using base64) and can thus save 55,527 bytes for the photo content.

5.3.2 Embedded XML in MPEG-21 DIDs

Within MPEG-21, it is useful to store additional XML within the DID (i.e., using MPEG-7 descriptors to describe a photograph). This embedded XML may be a user generated XML document without a schema, an XML document valid to other XML Schemas, or even another DID. For embedded XML which is valid to an XML Schema(s), it would be desirable to binarise the embedded XML as well to further improve compression efficiency.

The DID already contains enough information to determine that the embedded data is XML via utilising the `mimeType` attribute. The BiM MPEG-21 extension thus needs to recognise the text/xml mimetype.

When the BiM MPEG-21 extension encounters the `text/xml` mimetype, it needs to examine the XML contained within. There are two possible situations which need to be handled:

1. Embedded XML is valid to an XML Schema - in this case, the encoder can use the BiM technique to compress the embedded XML; and
2. Embedded XML is not valid to any XML Schema - in this case, the encoder can be configured to either compress the embedded XML with an XML aware compressor (such as those described in Section 2.4.2), or by applying the standard redundancy compression techniques (e.g. zlib) to the embedded XML. The type of compressor for non schema-valid XML is defined in the configuration.

Consider an example where a user embeds some MPEG-7 descriptors into an MPEG-21

DID, as illustrated in Figure 5.7. The MPEG-7 description contains some descriptors for a song that the user has selected. The `mimeType` attribute of the `Statement` tag is set to `text/xml`, to indicate that there is XML embedded within the document. The compressor determines that there is also an MPEG-7 XML Schema specified for the embedded content, and hence decides to use the BiM technique for compression of this embedded content. In this example, the embedded MPEG-7 content is originally 1,670 bytes in size. If a standard Zlib compression was applied to this element (i.e., without the MPEG-21 extension), the MPEG-7 content would be compressed to a size of 560 bytes. Using the MPEG-21 BiM extension developed in this Chapter, the MPEG-7 content compresses down to 136 bytes, this represents a saving of 424 bytes for this simple example.

If the MPEG-7 XML Schema had not been available, and the MPEG-21 BiM extension was configured to use an XML-aware compression technique, such as XMLPPM, the MPEG-7 embedded XML would compress to 506 bytes. This still represents a saving of 54 bytes when compared to the zlib compression technique.

The savings calculated in the above example demonstrate the advantage of using the MPEG-21 BiM extension to compress embedded XML within the MPEG-21 DID.

5.3.3 BiM DID Extension

Sections 5.3.1 and 5.3.2 demonstrated that through two simple extensions of the BiM technique as developed in this thesis, MPEG-21 DIDs can benefit from improved compression ratios.

The new MPEG-21 BiM extension is used when the encoder and decoder encounters the `Resource` and `Statement` elements within the MPEG-21 namespace. The following new algorithm is used:

if `mimeType` attribute is `text/text` **then**
  normal string (redundancy) compression is applied
else if `mimeType` attribute is `text/xml` **then**
  if the embedded XML is valid to an XML Schema(s) **then**
    write the binary code for the global element of the embedded XML document and

```
<DIDL xmlns="urn:mpeg:mpeg21:2002:01-DIDL-NS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001">
 <Item id="TRACK_214">
  <Descriptor>
    <Statement mimeType="text/plain">
           A Song (Track 214)</Statement>
  </Descriptor>
  <Descriptor>
   <Statement mimeType="text/xml">
    <mpeg7:Mpeg7>
     <mpeg7:Description xsi:type="CreationDescriptionType">
      <mpeg7:CreationInformation id="track-214">
        <mpeg7:Creation>
         <mpeg7:Title type="songTitle">The Song</mpeg7:Title>
         <mpeg7:Title type="albumTitle">Best Songs Ever
         </mpeg7:Title>
         <mpeg7:Creator>
          <mpeg7:Role href="urn:mpeg:RoleCS:2001:PERFORMER"/>
          <mpeg7:Agent xsi:type="PersonType">
            <mpeg7:Name>
              <mpeg7:FamilyName>Singer</mpeg7:FamilyName>
              <mpeg7:GivenName>Allen</mpeg7:GivenName>
            </mpeg7:Name>
          </mpeg7:Agent>
          </mpeg7:Creator>
            <mpeg7:CreationCoordinates>
                <mpeg7:Date>
                  <mpeg7:TimePoint>1990</mpeg7:TimePoint>
                </mpeg7:Date>
              </mpeg7:CreationCoordinates>
            </mpeg7:Creation>
            <mpeg7:Classification>
              <mpeg7:Genre href="urn:id3:cs:ID3genreCS:v1:X">
               <mpeg7:Name>Rock</mpeg7:Name>
              </mpeg7:Genre>
            </mpeg7:Classification>
        </mpeg7:CreationInformation>
       </mpeg7:Description>
     </mpeg7:Mpeg7>
    </Statement>
  </Descriptor>
 </Item>
</DIDL>
```

Figure 5.7 An example of an MPEG-7 descriptor embedded in a MPEG-21 DID

use BiM to compress the embedded XML. This technique is illustrated in Figure 5.8

else

write the binary code `0` to indicate BiM is not used

if use an XML-aware compression technique **then**

write a 1 followed by the binary code for the selection of the XML compressor (if more than one is defined in the decoderInit) and write the binary output using defined technique to compress the embedded XML. This technique is illustrated in Figure 5.9, where XYZ is the binary code to select the requested XML-aware compressor. The number bits for the XML-aware compressor selection field will depend on the number of encoders which are defined in the decoderInit

else

write a `0` followed by the binary output from the zlib compressor. This technique is illustrated in Figure 5.10

end if

end if

else if the `encoding` attribute is base64 **then**

convert the base64 embedded data back to its original binary form, and write the resulting binary data

else

normal string (redundancy) compression is applied

end if

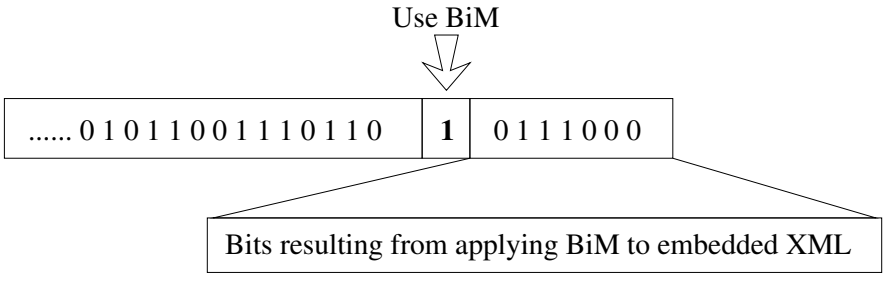

Figure 5.8 Example of a bitstream generated using the MPEG-21 BiM extension when applied to an embedded XML which is valid to an XML Schema

Since the `mimeType` and `encoding` attributes are encoded during the normal BiM process, the BiM MPEG-21 extension re-uses these attributes for the selection of data-type encoding

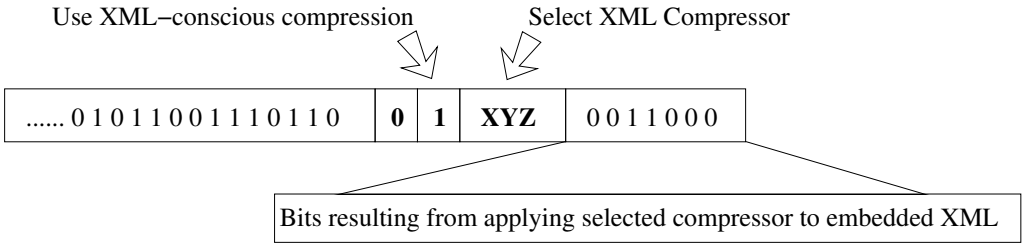

Figure 5.9 Example of a resulting bitstream using the MPEG-21 BiM extension applied to an embedded XML which is not valid to an XML Schema (where the XML-aware compressor is defined in the decoderInit)

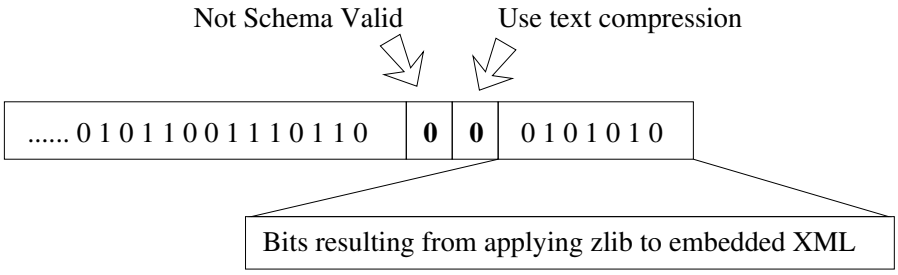

Figure 5.10 Example of a resulting bitstream using the MPEG-21 BiM extension applied to embedded XML which is not valid to an XML Schema (i.e., using a Text Compressor)

of the embedded data.

5.4 Binary RXEP (BinRXEP)

Section 5.2 demonstrates that XML Schema-based compression techniques can achieve good compression ratios, especially on small XML documents. In contrast, traditional redundancy compression algorithms applied to small messages often achieve very little reduction, and in some cases, the file size may actually increase [83].

The native RXEP commands (see Chapter 4) are expressed in XML, validated to an associated XML Schema, and thus, the protocol itself is inherently able to be binarised. The results shown in Section 5.2 show that XML Schema based compression algorithms are the best candidates for compression of RXEP (see Chapter 4), to further reduce the size of the protocol (as it is defined in XML) as well as the XML data being fragmented and transmitted.

For example, a portion of the RXEP XML Schema is illustrated as a tree-like view in Figure

```
RXEP
 [CHOICE] {1,1}
   Request (0) {1,1}
    [SEQUENCE] {1,1}
       Src (0) {0,1}
         'openMode'
       [CHOICE] (1) {0,Unbounded}
          Query   (00)     {1,1}
          XMLPull (01)     {1,1}
            ...
          Stream  (10)     {1,1}
   Response (1)
      ...
```

Figure 5.11 Example of assigning binary codes to the RXEP XML Schema

```
<RXEP>
 <Request>
   <Src openMode="true">
     /examples/xml/book1.xml
   </Src>
 </Request>
</RXEP>
```

Figure 5.12 An example RXEP Request

5.11, and an example RXEP request which is valid to the XML Schema in Figure 5.11 is shown in Figure 5.12. The equivalent binary codes are surrounded by round brackets, i.e., (10) defines the binary code for the second child. The minimum and maximum occurrence of for each node is written using brace notation, e.g., {0, Unbounded} indicates that the node does not need to occur, and there is no upper bound on the number of times it may appear. Attributes of an element are represented by single quotes (i.e., 'openMode').

To illustrate the bit savings, the RXEP XML from Figure 5.12 is composed using the codes from Figure 5.11. Here, the `RXEP` and its choice requires no bits (they must be present). The `Request` element requires one bit (code `0`) and the `Src` element is used, also requiring one bit (code `0`). The optional attribute `openMode` requires one bit (code `1` to indicate its presence) and one bit to represent the 'true' value (code `1`), with the string requiring 184 bits. Lastly, one bit is written to indicate that the choice in the sequence is not used (code of `0`). The original RXEP request is 103 bytes (824 bits) and the compressed version is 189 bits, yielding a compression ratio of 77%.

Hence, in this example, the RXEP structure (without the string to locate the file) requires only 5 bits, demonstrating the effectiveness of XML Schema-based compression algorithms when applied to RXEP.

The problem remaining is the binarisation of randomly requested fragments of the XML file which is to be transmitted. While any compression technique can be applied to the fragments, compression efficiency is maximised when the XML is schema valid and a schema based compression (e.g. BiM) can be employed. BiM is not able to perform this task directly since it does not possess mechanisms to control the binarisation process. Instead, the assumption is made that the whole XML document will eventually be transmitted.

Generally, BiM and schema-based compression schemes will generally encode the entire document subtree at once, recursing though each child of every branch. BiM does allow compressed fragments to be sent, however, the fragments are predetermined by the author and cannot be dynamically created and compressed via user requests. Furthermore, BiM uses a mechanism known as 'deferred nodes', which are used to indicate to the receiver that there is data in a subtree and that the data is currently unavailable to the application. This

mechanism however, is only useful in a 'push' streaming approach and is not helpful when the requirement is to 'navigate' remotely through the XML document or if only parts of the document are to be retrieved.

Thus, the binarised RXEP (BinRXEP) will extend upon XML tree/schema compression techniques (as used in BiM) to incorporate controls on the binarisation derived from the RXEP commands (i.e., node location and level depth). In particular, this requires that the binarisation is applied only to direct children of nodes with all their information (such as attributes and values, when RXEP navigation is performed). It is also possible to specify level depths to request all descendants of a node to a certain level, and to utilise full XPath expressions to access multiple elements and descendants in a single request.

5.4.1 Schema DOM Tree

To allow compression of randomly chosen XML fragments, the compression of the XML fragment cannot be performed ahead of time with BiM, since different clients will require different fragments. Furthermore, navigation and queries require different return formats; navigation requires only the direct children of the selected node to be sent, whereas queries may require many levels of child nodes. In the previous Section, Figures 5.11 and 5.12 illustrate that the structural information (provided by *choice*, *sequence* and *all* types) are not present in the XML instance, however, this information is added into the binarised file during the compression process. This structural information is needed to guide the decompressor through the correct path within the associated XML Schema.

For example, consider the tree-view schema in Figure 5.13 and the XML instance in Figure 5.14. In Figure 5.14, the node `A1` occurs twice, through the use of the parent choice (see Figure 5.13). Thus, the binarisation process encodes that this choice is selected twice followed by two `A1`s, since the decoder is using the schema to determine which node to decode next.

A second issue which arises when binarising RXEP requested fragments is that when a fragment is selected for binarisation, the encoder needs to validate the fragment prior to the encoding process to ensure that the correct codes are generated. For example, Figure 5.14 shows that there are four sequential `Element1` nodes. Upon inspection of the schema from

```
Top
 [Choice] {1, 2}
   Element1 {0, 2}
   Element2 {0, unbounded}
   [Choice] {0, unbounded}
      A1 {0, 1}
      ...
   [Choice] (0, unbounded}
      ....
```

Figure 5.13 An example Schema, presented as a tree view

```
<Top>
  <Element1/>
  <Element1/>
  <Element1/>
  <Element1/>
  <A1/>
  <A1/>
...
```

Figure 5.14 Valid XML accordind to the Schema in Figure 5.13

Figure 5.13, it is clear that the only way that four sequential nodes can occur is by two instances of `Element1` nodes, where each `Element1` node has an element occurrence of two.

Since the server will be combining XML fragments from different places in the tree, it makes sense that all the validation information is present within the one tree - suggesting the creation of a document which closely resembles the XML Schema document; this document is known the as Schema DOM tree (SDOM).

SDOM is a DOM tree which merges the relevant XML Schema structure (i.e., sequence, choice and all) into the XML DOM. This phase is much like the validation phase of an XML parser, to ensure correctness, it must be able to check all the combinations.

For example, the XML instance in Figure 5.14 would be represented as the corresponding SDOM in Figure 5.15 by integrating the structure information from Figure 5.13.

Within the SDOM tree, the structural information (i.e., choices, sequence and all) contains

```
<Top>
 [Choice]
  (Element1, Occurrence = 2)
    <Element1/>
    <Element1/>
  (Element1, Occurrence = 2)
    <Element1/>
    <Element1/>
  [Choice]
    <A1/>
    <A1/>
...
```

Figure 5.15 SDOM representation of the XML in Figure 5.14 combined with the XML Schema information from Figure 5.13

pointers selecting the appropriate XML Schema node. This approach allows the binarisation process to avoid the need to validate the XML fragment each time the fragment is selected and compressed.

In summary, the SDOM technique provides the following advantages:

- Reduce the need to validate against the XML Schema each time a fragment is to be binarised - since the SDOM combines the XML Schema information into the XML which requires a valid XML document, the encoder knows the fragment is valid and already contains all the necessary schema information;
- Allow all BiM schema transformation rules to be applied before merging into SDOM - the BiM Schema transformation simplifications (see [7] for more information) to transform the XML prior to conversion to SDOM;
- SDOM documents can be saved as a file to reduce the need to convert into SDOM later;
- Allows easy stopping and starting at random locations in the xml of the encoding process - where dynamic XML fragments can be selected, the SDOM already has all the schema information for each node, and can begin compression at any point in the SDOM;

- Supports RXEP navigation functionality - when the SDOM is set to operate in navigation mode, only the direct children of a selected SDOM node are encoded; and
- Increased speed when binarising fragments - since all the schema information is already contained within the SDOM, there is no need to check against the schema each time a fragment is selected for binarisation.

5.4.2 BinRXEP Algorithm

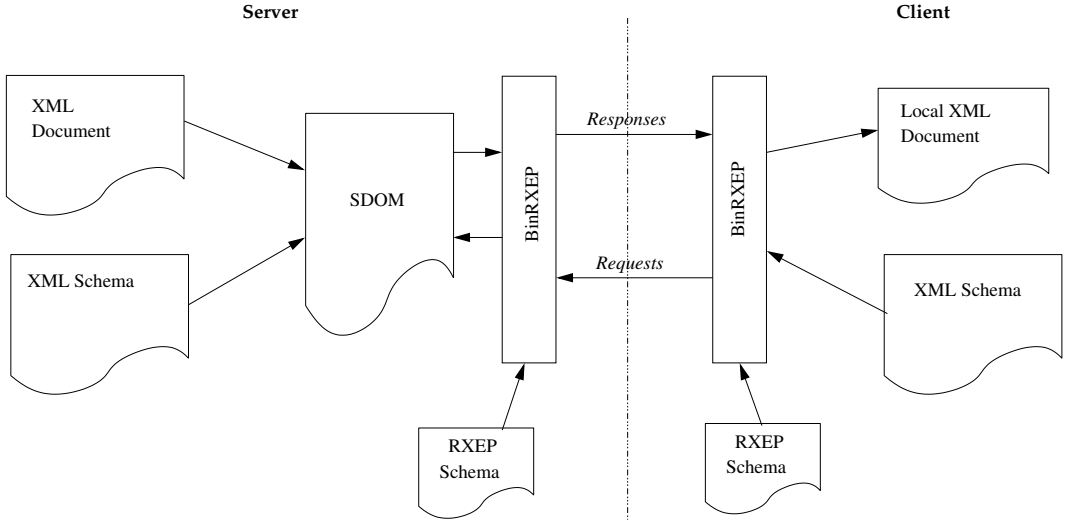

Figure 5.16 Block diagram of the Binary RXEP system

The block diagram of the BinRXEP system is illustrated in Figure 5.16. In BinRXEP, a server extracts structural information from the XML Schema and merges that information into the XML document, creating an SDOM document (see Section 5.4.1). As the client requests fragments from the XML document, the server maps the RXEP XPath locator onto the SDOM document, selecting the fragment for binarisation. The fragment is inserted into the RXEP response, and both the RXEP XML and the requested XML fragment are binarised and delivered to the client. The client decompresses the binary RXEP response and appends the (uncompressed) XML fragment to its local version of the document.

The encoding process steps though the SDOM and writing the appropriate bits (similar to the BiM procress in [7]). The flowchart in Figure 5.17 illustrates the process used for encoding each node in the SDOM. As can be seen from the flowchart, the first step is to determine

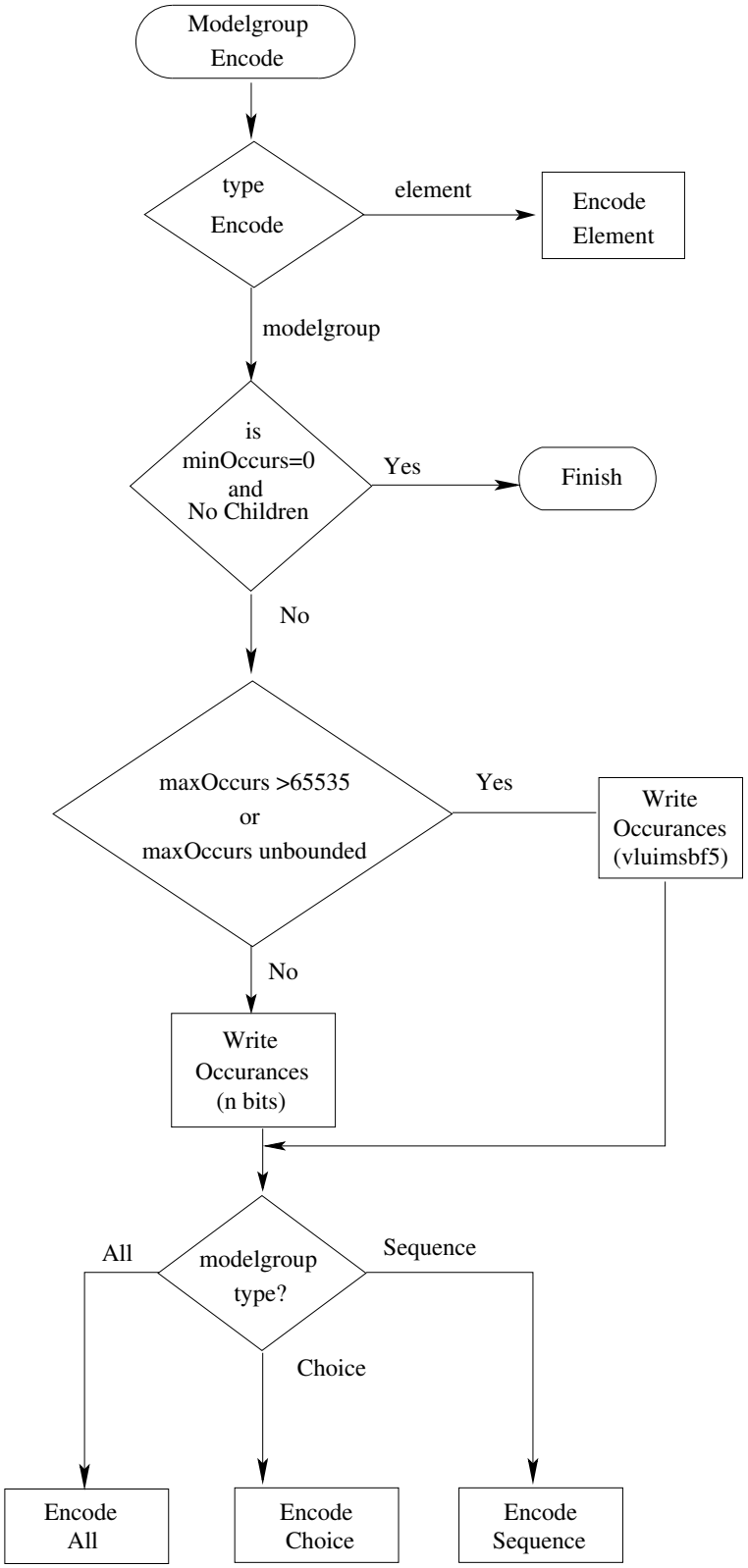

Figure 5.17 Flowchart for encoding an XML Node

if the SDOM node is an XML element or an XML modelgroup node. If the SDOM node is an element type, then the encoding process follows the flowchart shown in Figure 5.18 - otherwise the encoding continues as shown in Figure 5.17. If the SDOM indicates that the minOccurs of the modelgroup element is zero and there are no occurrences in the SDOM, then a zero is written to indicate no further bits are encoded. If there is at least one or more occurrences of the SDOM modelgroup node then these are encoded using either the encoder for Choice, Sequence or All.

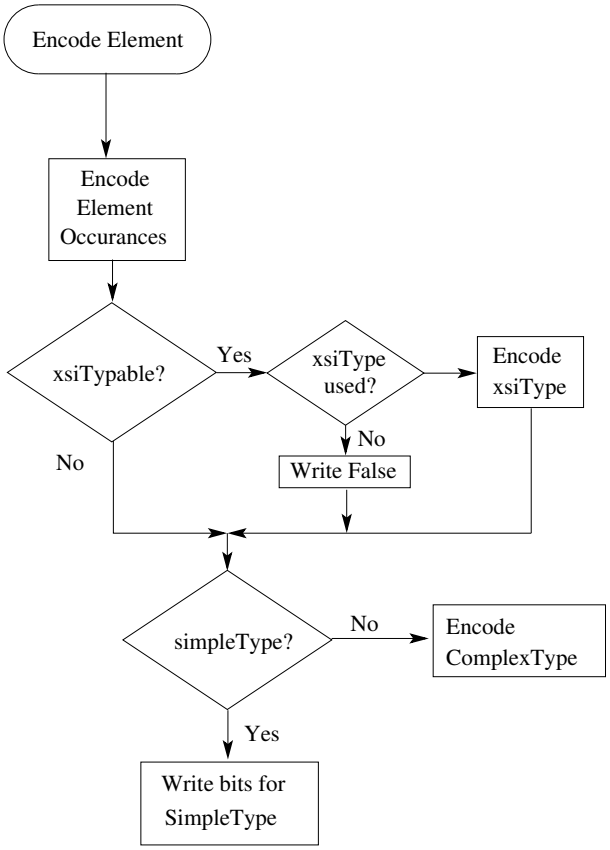

Figure 5.18 Flowchart for encoding an Element XML Node

The flowchart as illustrated in Figure 5.18 applies to all SDOM XML elements encountered during the encoding process. For each element occurrence, the xsi-type attribute is checked, and is encoded if applicable. If the element being encoded is a simpleType (e.g., strings and integers) then the appropriate datatype compressor is used to encode the simpleType data. If the element is a complexType then complexType encoding is performed as illustrated in Figure 5.19.

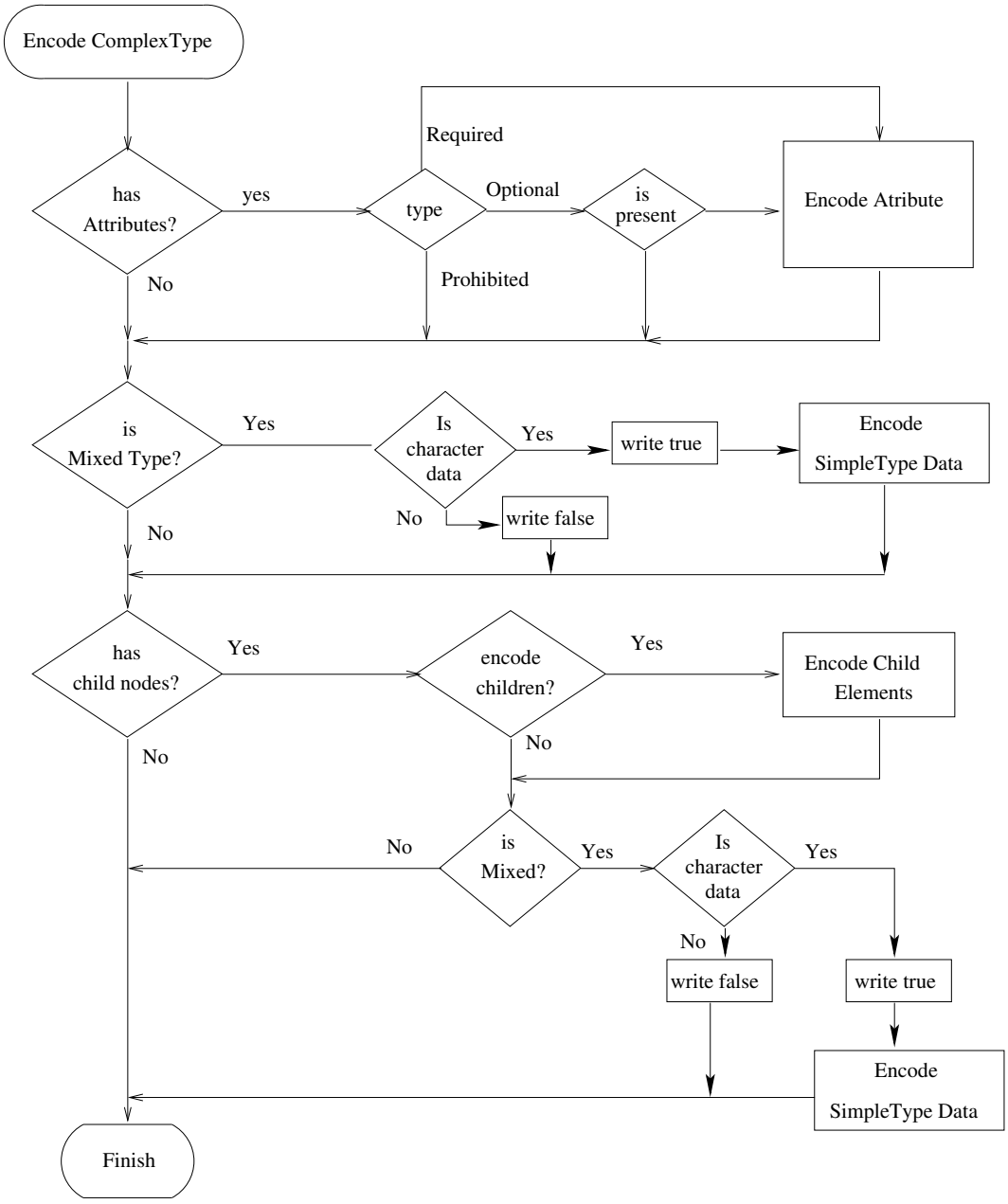

Figure 5.19 Flowchart for encoding a complexType XML Node

The first step in the complexType encoding process is to check the associated XML Schema for that node and determine if there are possible attributes, and if so, whether they are optional or mandatory. Any attributes present in the SDOM are then encoded. If an SDOM element can be a 'mixed' type (i.e., there may be character data before and after, an XML element), and if character data is present then this is encoded. If there are child nodes of the current SDOM node, then the character before the XML child is encoded, then the child node is encoded, followed by encoded character data after the XML child.

5.4.3 Binary RXEP XPath Locators

RXEP XPath locators are used to specify the element for retrieval (client requested) and also used to specify the location on the client side to apply the RXEP action (for information on RXEP locators see Section 4.3.3.2). RXEP locators are used with RXEP response commands such as Add, Delete, Update and Insert. Since the XML is valid to an XML Schema, then the RXEP XPath Locators must follow the schema rules to be valid. By exploiting this information, this Section presents a novel technique which utilises the same binarisation techniques as used to compress the XML document for the binarisation of the RXEP XPath locator. This process is slightly different to the binRXEP process since the binarised RXEP Locators do not contain the XML model group information and the action is always a choice (identifying exactly one node). Figure 5.20 illustrates the new technique used to binarise an RXEP locator.

Consider the example schema (illustrated with binary codes) as shown in Figure 5.21. Figure 5.22 shows an example XML instance valid to this schema, and Figure 5.23 shows an example RXEP packet, where the XPath locator is "/Media/Music/Song[2]". Since `Media` is the only root node, it is mandatory and thus no bits are required. A `1` is written to indicate that the element is used, the binary code for `Music` element is `0` and since there can be an unbounded number of Music nodes (via the choice), a position code of `00001` is encoded using VLC5 [7]. Following the same process for Song[2] a `1` is written to indicate element name and a `0` is used to select the `Song` element and `00010` for the second position code. Since there may be a number of Songs the 'index' is needed to represent the [2], which would be a 0 (to indicate the standard XPath index is used) and `00010` as VLC5 to represent the

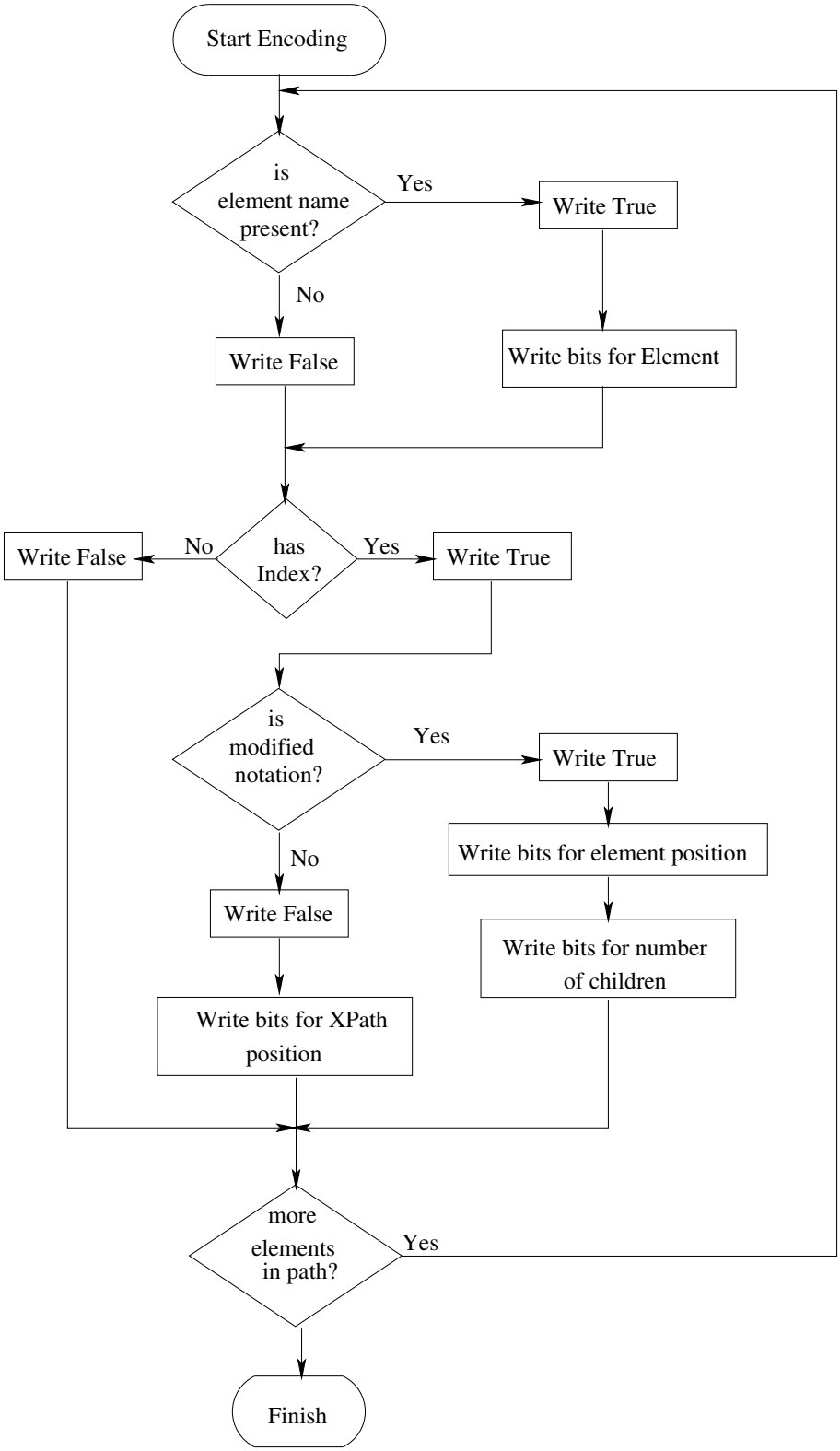

Figure 5.20 Flowchart illustrating the encoding process of an RXEP XPath Locator

```
Media
  [CHOICE] {1,Unbounded}
   Music (0) {0,1}
    [CHOICE] {0,Unbounded}
       Song (0) {0,1}
        [SEQUENCE] {1,1}
          Title        {1,1}
          Description  {0,1}
          Artist       {1,1}
          Format       {1,0}
          Rating       {0,1}
          Length       {0,1}
     ...
   Videos (1)
      ...
```

Figure 5.21 Tree view of an example XML Schema describing the format for a collection of media

```
<Media xmlns="mediaNS:2004">
   <Music>
     <Song id="Hit1">
       <Title>Hit.1</Title>
       <Description>Song 1</Description>
       <Artist>A. Artist</Artist>
       <Format>MP3</Format>
       <Length>02:23</Length>
     </Song>
     <Song id="Hit2">
       <Title>Hit.2</Title>
       <Description>Song 2</Description>
       <Artist>B. Artist</Artist>
       <Format>OGG</Format>
       <Length>03:46</Length>
     </Song>
   </Music>
</Media>
```

Figure 5.22 Example XML document valid to the Schema shown in Figure 5.21

```
<RXEP xmlns="RXEP:2004">
  <Response>
   <Add location="/Media/Music/Song[2]">
      <Title>Hit.2</Title>
      <Description>Song 2</Description>
      <Artist>B. Artist</Artist>
      <Format>OGG</Format>
      <Length>03:46</Length>
   </Add>
  </Response>
</RXEP>
```

Figure 5.23 Example RXEP packet

index of '2'. The total output is `1 0 00001 1 0 00010 0 00010` (20 bits) which is significantly smaller then the textual representation which requires 160 bits.

As with the RXEP XPath locator, the Binary XPath locator also supports the modified XPath expression, which contains an additional number indicating where each element is positioned. This provides additional information so that the the element can be placed in its 'exact' position, preserving element order (if necessary). For example, if the Music node in Figure 5.22 was altered to have three children (i.e., Song, Other and Song), /Media/Music/Song[3,3] from the XPath selects the second song element (which is actually the third child). The binarisation process follows the flowchart as illustrated in Figure 5.20, and the output is quite similar to the previous example. In this example however, the [3,3] is encoded by VLC5 which is `00011 00011`. The output of the binRXEP XPath Locator would now be `1 0 00001 1 0 00010 0 00011 00011` (25 bits).

5.4.4 Navigation with BinRXEP and encoded RXEP XPath Locators

As with RXEP, binRXEP can be used to to navigate through remote XML documents (see Section 4.3.1.3), where the data and the protocol are transmitted in a binary (compressed) form. Section 5.4.3 demonstrates an efficient method of representing the RXEP XPath locators as binary which will be used for the encoding of the RXEP locators in the RXEP Requests.

For example, Figure 5.21 and Figure 5.22 illustrate an example schema with the corre-

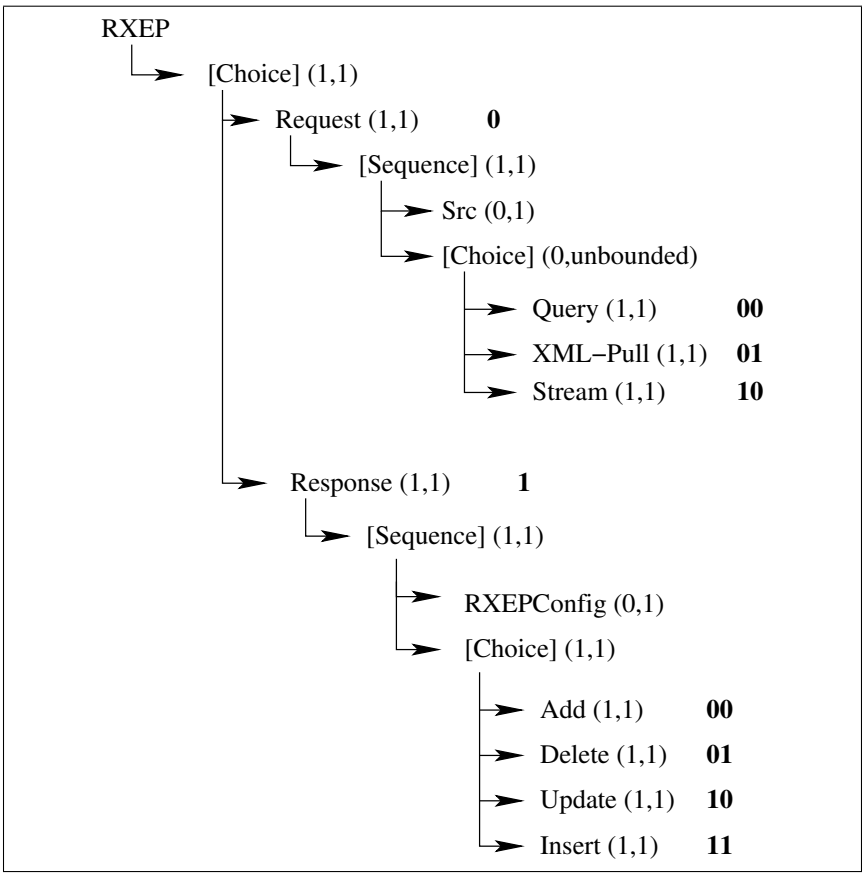

Figure 5.24 The RXEP XML Schema illustrated as a tree-like structure, where the minOccurs and MaxOccurs are denoted using parentheses (), and the nodes corresponding binary code for each node shown in bold font

sponding binary code of each element and an example XML instance, respectively. In this example a users wishes to navigate the XML node located by the XPath expression `/Media/Music/Song[2]`; an RXEP request containing a Src command (in open mode) and a Query command (in navigation mode) is first constructed as illustrated in Figure 5.25.

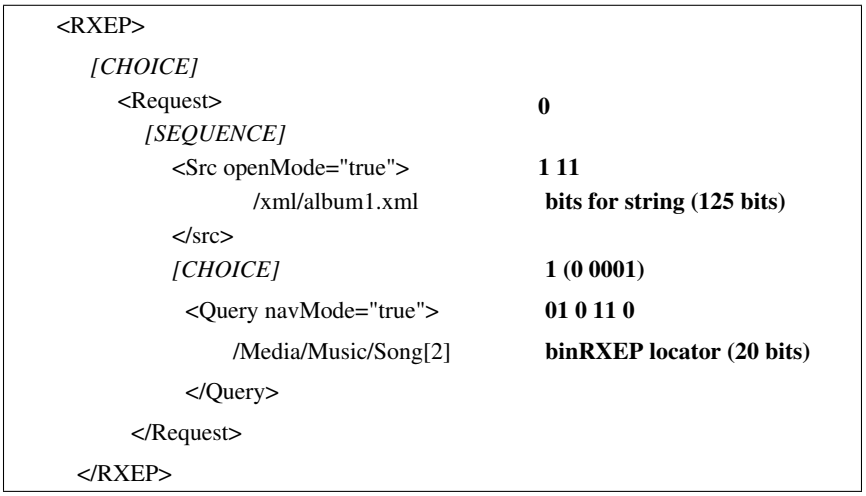

Figure 5.25 SDOM representation of an RXEP request illustrating the binary output for each node in bold font. The total size of the binary output is 161 bits

Figure 5.25 shows the binary codes generated for the RXEP request by using the RXEP Schema as shown in Figure 5.24 and the schema for the album as shown in Figure 5.21. In this example, the RXEP root node is required and thus no bits are needed. The `[Choice]` is also mandatory and can only occur once and so again, no bits are needed. The `Request` element is selected and one bit is written. This process continues and the output is shown in Figure 5.25. In this example the RXEP locator (`/Media/Music/Song[2]`) is compressed using the technique in Section 5.4.3.

After the server has received the binRXEP packet, it is then decompressed and processed. Since the `Query` element has the `navMode` attribute set to true, the server knows that RXEP navigation is desired (for more information see Section 4.4.1). The RXEP Response constructed by the server is shown in Figure 5.26 along with the corresponding binary codes. The total output of the RXEP response is 39 bits.

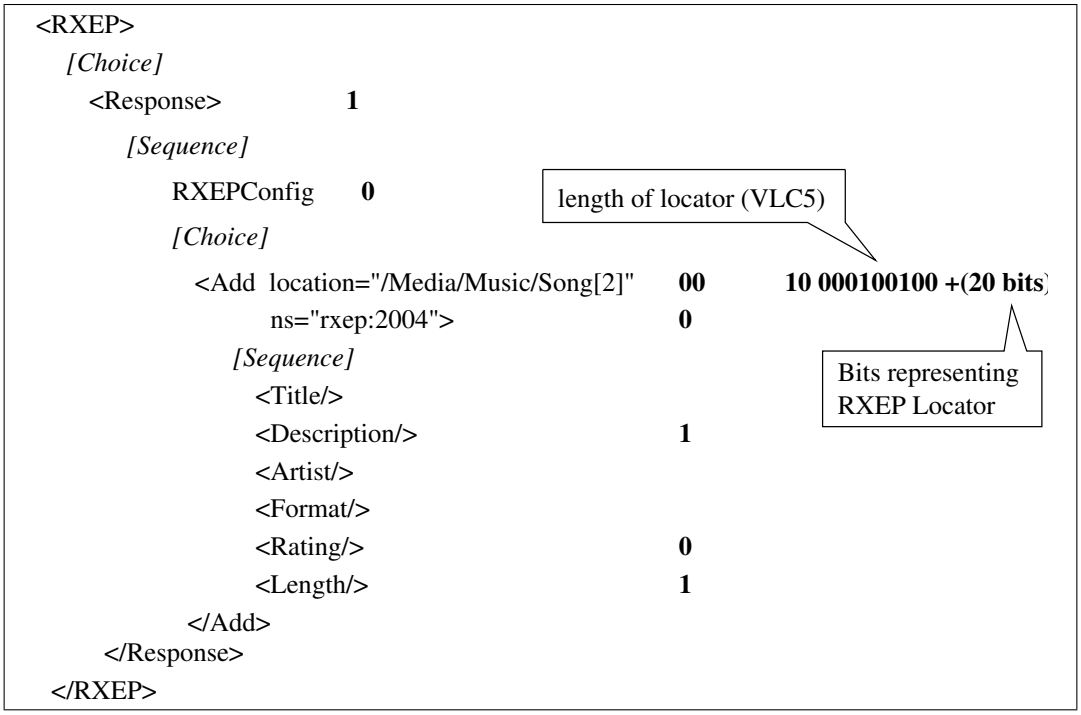

Figure 5.26 Example RXEP response illustrating the binary codes (as bold font). The total binary output is 39 bits

5.4.5 Navigation with BinRXEP XML-Pull

Section 5.4.4 demonstrated how the binRXEP and binRXEP XPath locators can be combined to efficiently perform remote navigation. This Section will now consider binRXEP navigation using XML-Pull commands (refer to Chapter 4) rather than XPath locators (as described in Section 5.4.4). Instead of receiving all the direct child nodes from a selected node, the encoding is preformed on a per-node basis. This is analogous to XML Remote Pull-parsing, however, by using RXEP the 'parsing' is truly remote (i.e., across the network).

Figure 5.27 illustrates the RXEP request as SDOM, showing corresponding binary codes using the code from the schema in Figure 5.24 and the XML-Pull definition in Section 4.3.1.3. Reading the binary output (bold text) from Figure 5.27 from top to bottom, the output binary for the XML-Pull Next request is `0000001011000`. The binary output for the Up, Expand and Back commands are obtained by changing only the code which represents the Next command from Figure 5.27. In this case the total output for an XML-Pull Up request would be `000001011`**`01`**`0`. Similarly, the Expand request is `000001011`**`10`**`0` and the Back request

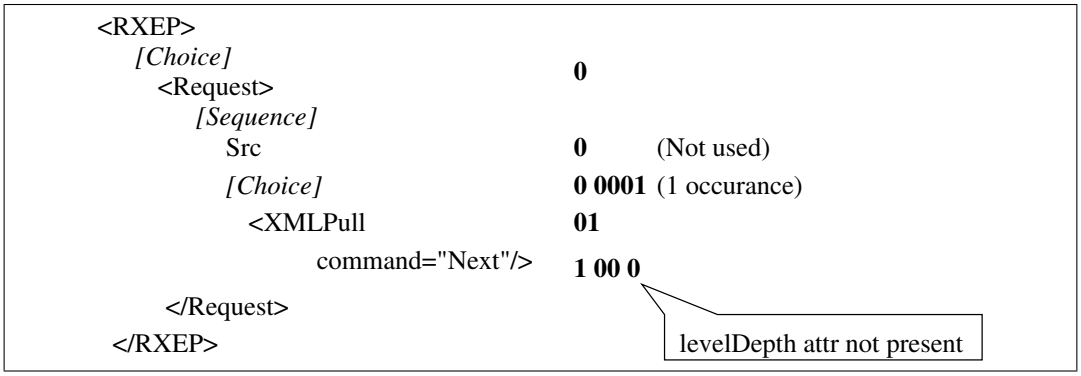

Figure 5.27 Example RXEP XML-Pull Request illustrating the binary codes (as bold font). The total binary output is 13 bits

is `000001011`**`11`**`0`. The binary codes for these XML-Pull RXEP requests will not change as there are no dynamic strings (i.e., no Src command is specified) and the RXEP Schema does not change in these RXEP requests.

Figures 5.21 and 5.22 show an example schema view with corresponding binary codes and an example XML instance (valid to the schema in Figure 5.21), respectively. In this example, the client has already sent an initial Src command and received the document root node (`Media`) as well as the `Music` node (which is now the 'current' node). The client sends the XML-Pull Expand command to navigate to the `Music` node as illustrated in Figure 5.28. Since all the RXEP responses in this example will be an RXEP Add, then all responses will contain the same beginning binary codes (these code can be seen in Figure 5.21); these bits are `10000`. Since the client and server are both in open mode and using XML-Pull commands, the client knows the selected node, and thus does not require the RXEP response to contain the locator attribute.

If the client decides to 'expand' the current node, it sends the appropriate binary codes as illustrated in Figure 5.28. The server receives this request and selects the first child of the `Music` node, which is `Song`. The `Song` node is binarised and sent back to the client. In this case, the client decides that it does not wish to listen to hit1 and transmits the 'Next' command, in binary, to the server. The server selects the second `Song` node, binarises the node, and sends the result to the client. This process, as illustrated in Figure 5.28, continues until the user has the description of the second song.

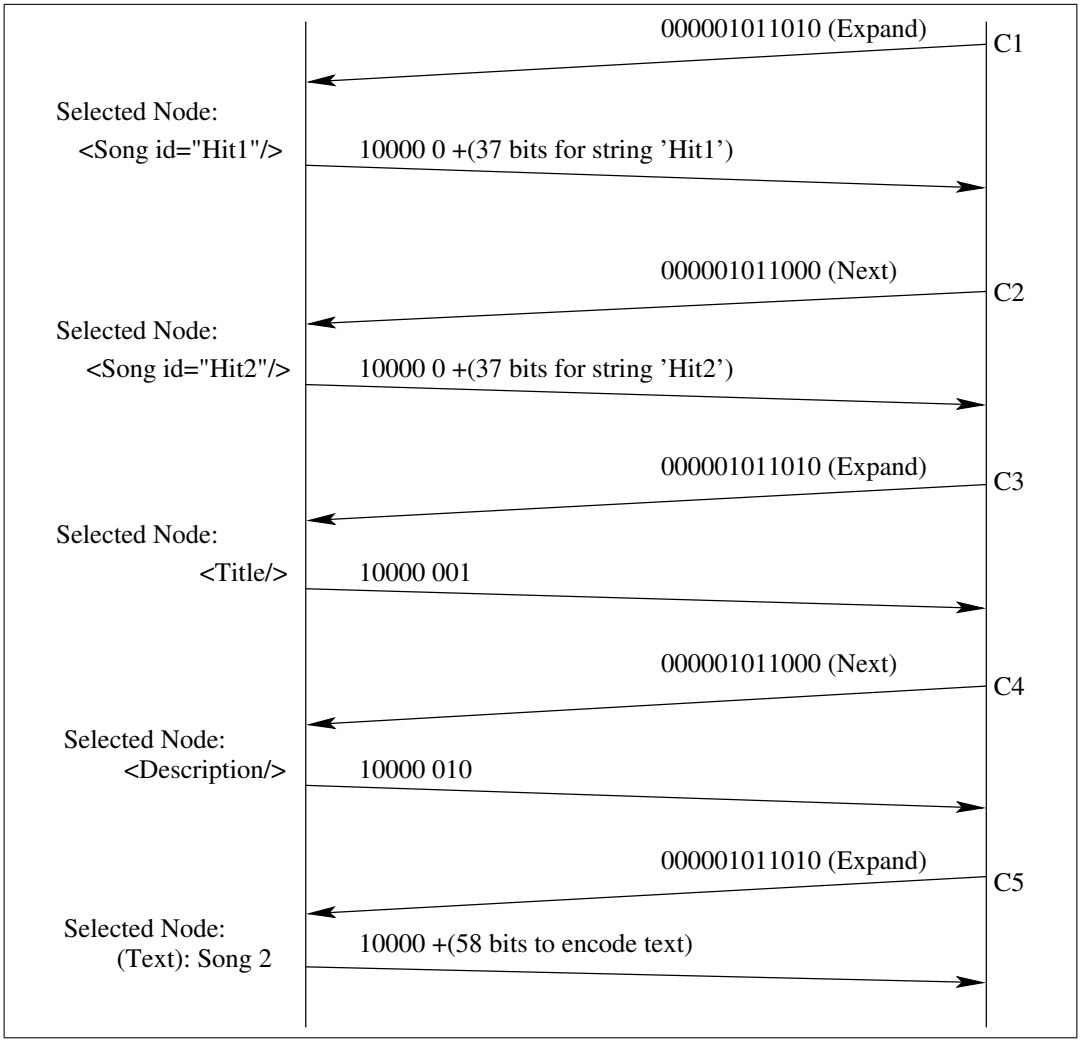

Figure 5.28 Example timing diagram illustrating the exchange of the binary codes using binRXEP to navigate through an XML document

When in navigation mode using RXEP XML-Pull, the sequences are not used in the encoding process, instead, a choice action is used to encode the currently selected node. In Figure 5.28 the children of the `Song` node are encoded using a choice action rather than the sequence as specified in the schema. This allows the client to step through the nodes on a node-by-node basis.

5.4.6 BinRXEP Fragments from Queries

There are many cases when it is desirable to request multiple elements or XML fragments rather than using navigation as seen in Sections 5.4.5 and 5.4.4. Multiple elements or XML fragments can be requested within a single query using XPath queries within RXEP. The process will be illustrated using the simple 'Example XML' in Figure 5.22 and the RXEP packet shown in Figure 5.23.

For example, a client constructs an XPath expression (e.g. `//Song[@id="Hit2"]`) to query the remote XML document, which requests all songs beginning from the root node, with an attribute ID "Hit2". In the XML code of Figure 5.22, there is only one matching element and the fragment of XML returned by RXEP will be that element and its immediate children; this fragment is placed within an RXEP response as shown in Figure 5.23.

Although the `ns` attribute (which specifies the correct namespace for the XPath locator) is present on the `Add` element, in this example it is not needed as there is only one namespace associated for the remote XML document.

During binarisation, the binary codes will follow the same procedure as described in the example in Section 5.4.4. However, when nodes are selected on the basis of a Query (from the client), it cannot be assumed that the client already has sufficient information, therefore it must use the modified RXEP XPath locator (see Section 5.4.3) for exact placement. In this example, the RXEP XPath locator would be `/Media/Music/Song[2,2]`.

In more complex cases, where a level depth parameter is included with the query, the encoder continues down all subtrees until reaching either the end of the subtree or the desired level, and all requested fragments will be transmitted within a single RXEP response.

5.4.7 BinRXEP Experimental Results

Due to the reliance on user based or application specific selections of XML elements, simulated results will be presented by way of a number of simple scenarios to demonstrate the effectiveness and advantages of binRXEP. Results have been generated using a binRXEP implementation using JAVA (for more information see Appendix A). The binRXEP software consists of two separate applications, a server and a client. The client provides the user with a simple interface to select remote XML documents (via a URL) and the XML is presented to the user as a JTree. As a user selects a node, the client constructs the appropriate RXEP request, binarises it and transmists the compressed RXEP request to the server. When the client receives the binRXEP response from the server, it is then decompressed and used to update the JTree to reflect the newly updated local XML instance.

5.4.7.1 Scenario One

Using the same scenario as in Section 4.6.1, a baseball fan is at a live baseball match and wishes to settle a discussion with his friend about some players and teams from 1998. The baseball fan uses his mobile device to access the baseball statistics which are stored in an XML document. Having previously searched the baseball statistics, the baseball fan is has knowledge of the structure and layout of the statistics.

The baseball fan and his friend wanted to know which players in the national league have scored more than 120 runs and played more than 150 games. The baseball fan uses binRXEP to connect to the 1998 baseball statistics XML document and constructs an RXEP Query which is binarised as illustrated in Figure 5.29. To ensure that all the details of each returned player is retrieved, the `leveldepth` attribute is set to `-1`.

The server decompressed the binRXEP packet and executes the query. The query matched four players from the baseball XML document, so the server constructs an RXEP response which is binarised and delivered back to the baseball fan's mobile device.

The size of the 1998 baseball statistics document is 763,831 bytes. The binRXEP Query in Figure 4.49 is only 119 bytes which represents uploaded data by the baseball fan's mobile

<RXEP>	
[CHOICE]	
<Request>	**0**
[SEQUENCE]	
<Src openMode="true">	**1 11**
/1998bbstats.xml	**bits for string (138 bits)**
</src>	
[CHOICE]	**1 (0 0001)**
<Query navMode="true">	**01 1 (bits for lvldepth) 0 0**
//bb:PLAYER[compare(ancestor::LEAGUE/LEAGUE_NAME, "National League")=0 and GAMES > 150 and RUNS > 120]	**binRXEP locator (110 bits)**
</Query>	
</Request>	
</RXEP>	

Figure 5.29 Example binRXEP request where binary codes are illustrated in bold font

Table 5.2 Comparison of RXEP, binRXEP and Zip for scenario one (All units in bytes)

	Upload	**Download**
RXEP	222	3,135
Zip	269	767
binRXEP	119	232

device. The total data download (i.e., from the binRXEP response) to retrieve the desired information is 232 bytes.

The overheads added by the binRXEP protocol for the upload and download in this example are 138 bits (17 bytes) and 102 bits (12 bytes) respectively. The total overheads of the protocol is therefore 29 bytes. Thus, the overheads added by binRXEP are not significant, and the user has only needed to transfer a total of 351 bytes of data.

Table 5.2 compares the sizes of the upload and download messages when using RXEP, binRXEP and RXEP packets zipped. As may be seen in the table, binRXEP obtains the smallest messages for upload and download.

5.4.7.2 Scenario Two

Using the same Scenario as in Section 4.6.2, a user wishes to search the world facts (stored as an XML document 'Mondial-3.0.xml'). The user constructs an RXEP query with an XPath

Table 5.3 Comparison of RXEP, binRXEP and Zip for scenario two (All units in bytes)

	Upload	**Download**
RXEP	207	10,360
Zip	145	2,544
binRXEP	96	8,913
binRXEP (zipped)	84	2,180

expression of `//country[@gdp_total > 900000]`. This RXEP query is binarised with a output size of 24 bytes. The server matched 9 countries from this binRXEP query, and sends back the results in a binRXEP response, with a size of 2,823 bytes. In this example, only the first level (i.e., navigation) of each country is retrieved, where further elements will be selected for download (on-demand).

After browsing through the result set, the user decides to expand the USA country statistics and constructs the relevant binRXEP query. The binRXEP query is 33 bytes and the user receives the binRXEP response, 6,045 bytes. The user decides to retrieve the number of religions in that country resulting in a binRXEP query of 39 bytes and a binRXEP response of 45 bytes.

Table 5.3 compares the sizes of the upload and download messages when using RXEP, binRXEP and RXEP packets zipped.

In this scenario, Table 5.3 shows that the zipped RXEP packets downloaded are significantly less then the binRXEP. This is due to large number of strings used, stored in a very flat structure. Furthermore, zip finds redundancy within the entire XML packets, compressing on a global level, rather than the compression retaining structural information (i.e., redundancy in strings are constrained within each element) used by binRXEP. However, when binRXEP responses are zipped the bytes uploaded and downloaded become 84 and 2,180 respectively.

Thus, using zipped binRXEP only 2,264 bytes are transferred when compared to the original filesize of 1,784,877 bytes.

5.5 Collaborative Editing with BinRXEP

Section 4.7 demonstrated how RXEP could be used in a collaborative editing environment. Whilst RXEP performed well, further savings can be achieved by using binRXEP.

5.5.1 Receive XML Documents with RXEP

The first set of results examines RXEP's efficiency when downloading only sections of data. Here textual and binary representations are compared to the original document size and results are taken from the upload and download traffic totals from the client's device.

The first text file for comparison is an XML version of Shakespeare's Othello. This file contains little structure and mostly string text, with an original file size of 248,777 bytes. The test was performed by navigating to Act 8->Scene1->1st speaker, changing the speaker name, and uploading the alteration. The text mode resulted in an upload of 1,176 bytes and a download of 1,598 bytes, while the binary mode resulted in a 50 byte upload and 1,349 byte download. To stream the text to that position (i.e. downloading the whole file to that point) would have required 207,030 bytes; this demonstrates the savings achieved by skipping the unwanted parts of a file.

The second example consists of a photo album represented by a MPEG-21 Digital Item. Initially, this XML file is of length 21,040 bytes. The test navigated to the level of photo descriptors and altered the description of a photo. The text mode resulted in a 1,409 bytes upload and 1,914 bytes download while the binary method results in a 54 byte upload and a 1,307 byte download. To stream to the chosen location in text would require 3,997 bytes. Figure 5.30 graphically illustrates the comparison of RXEP and binRXEP with the simple examples while Figure 5.31 illustrates the comparison of text and binary representation of the complete test files. Since these are very simple examples, and navigation skips most of the unnecessary structure information, the text and binary download results are not significant. However, more complex examples, involving more navigation through the structure would result in further improvements as illustrated in Figure 5.31. The results show that using binRXEP the uploaded data is very small and is a minimal cost for the flexibility afforded by RXEP.

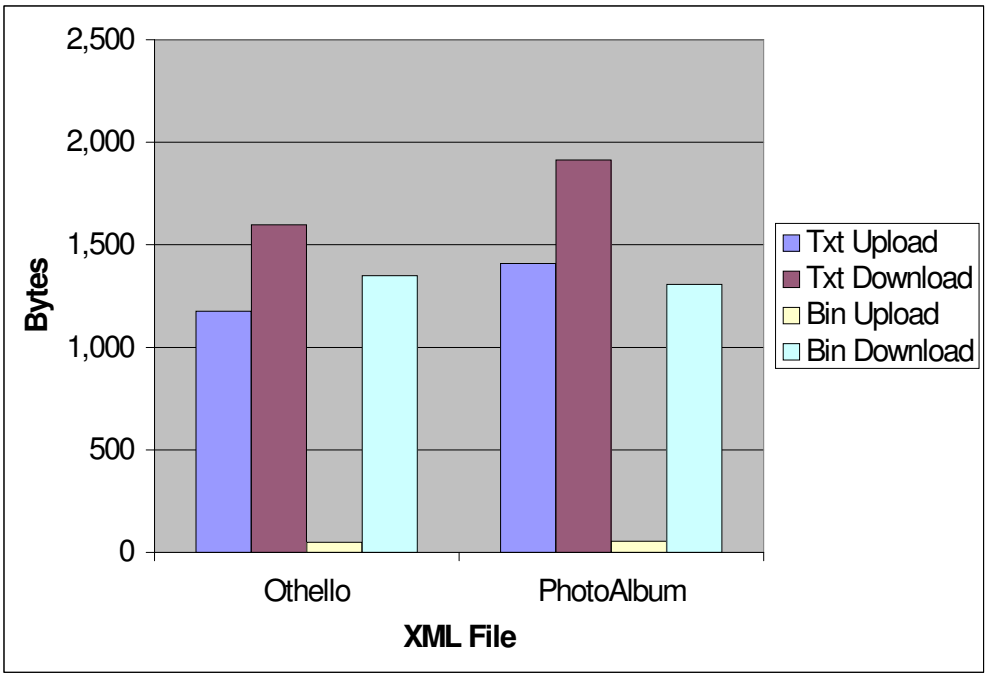

Figure 5.30 Comparison of upload and download of the test files using both binRXEP and RXEP

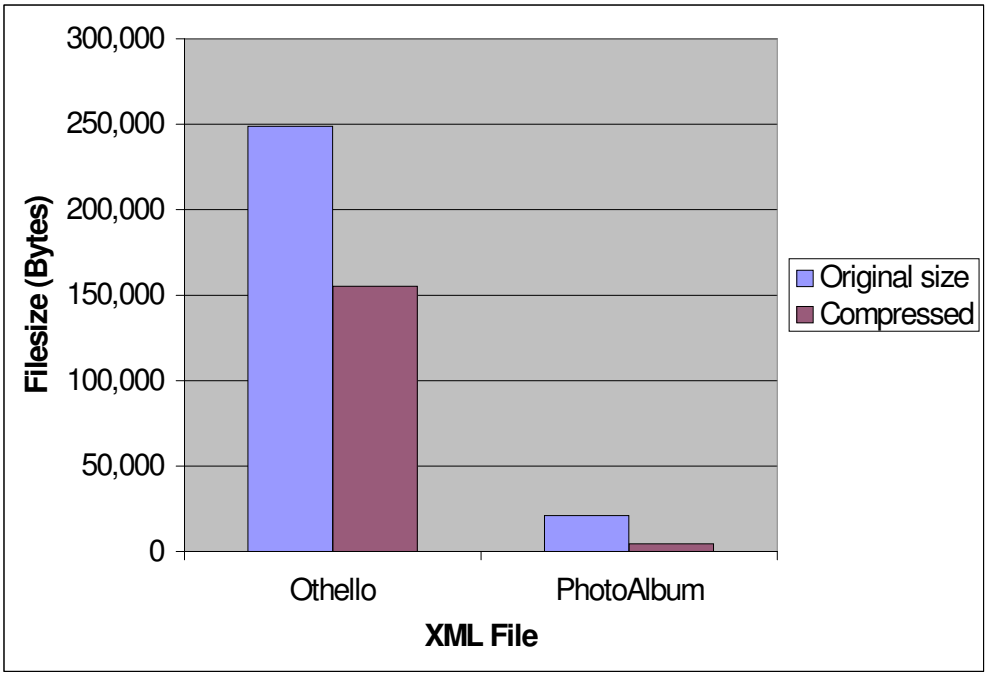

Figure 5.31 Comparison of text and binary compression

5.5.2 Binarisation of Office XML Documents

Section 4.7.1.2 compared the two most popular office suites offering XML format for office document when using RXEP. It was found that the two office suites use a different structure to represent the same data (i.e. WordprocessingML uses a highly structured representation, while openDocument uses a flatter structure).

Unfortunately, the openDocument XML could not be binarised since the schemas are written in Relax-NG [15], which is not supported by the current version of the compression code. Furthermore, the MS WordprocessingML documents could not take full advantage of the tree-based compression due to the way the XML Schema has been written and structured. The problem arises in this example after the `wx:sect node`. Figure 5.32 illustrates a portion of the WordprocessingML test document concentrating on the body of the document. The corresponding XML Schema for this node is shown in Figure 5.33. As can be seen in Figure 5.33, the `sectElt` type (which is the type of the sect node) has a sequence of a sub-section or an any node. Additionally, the sub-section type also contains an any node in its sequence.

The *any* node in this case uses the "lax" processing directive which allows any well-formed XML to be present as a subchild of the node, and validation of this XML is done on a "can do" basis [3]. Whilst the any node allows for flexibility, this has the unfortunate consequence of limiting its compressibility. The problem is that tree-based encoding relies on knowledge of the number of possible child nodes that can be present after the current node. Unfortunately, in this case the compressor cannot easily determine what node must come next after an *any* node with the lax processing directive.

5.5.3 Improving the Compression of WordML

One solution to improve the compression is to restrict nodes following an *any* node to be global elements from a known namespace. One bit would be used to indicate if the node is valid to an namespace, and the following bits used to select the namespace and a particular node within that namespace. Without this, it is not possible to determine the next node in relation to the schema, and hence to continue compression using the schema tree-based

```
<w:wordDocument ....   >
    ...
  <w:body>
     <wx:sect>
       <w:p>
         <w:r>
           <w:t>This is a Test sentence.</w:t>
         </w:r>
       </w:p>
       <w:p/>
       <w:p/>
       <w:sectPr>
         <w:pgSz w:w="12240" w:h="15840"/>
         <w:pgMar w:top="1440" w:right="1800"
                  w:bottom="1440" w:left="1800"
                  w:header="708" w:footer="708"
                  w:gutter="0"/>
         <w:cols w:space="708"/>
         <w:docGrid w:line-pitch="360"/>
       </w:sectPr>
     </wx:sect>
  </w:body>
</w:wordDocument>
```

Figure 5.32 Sample of a Wordprocessing ML document

```
<xsd:element name="sect" type="sectElt">
</xsd:element>

<xsd:complexType name="sectElt">
  <xsd:sequence>
    <xsd:element name="sub-section"
                 type="subsectionElt"
                 minOccurs="0"
                 maxOccurs="unbounded">
    </xsd:element>
    <xsd:any namespace="##other"
        processContents="lax" minOccurs="0"
        maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="subsectionElt">
  <xsd:sequence>
   <xsd:any namespace="##other"
       processContents="lax" minOccurs="0"
       maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Figure 5.33 Relevant portion of the WordprocessingML Schemas showing the schema for the section elements

technique. If the schema node cannot be determined, then the contents of all XML under that node will be compressed with a standard character redundancy compression technique, such as Zlib [67]. This approach is less flexible and may result in reduced compression efficiency when compared to tree-based compression (see results in Chapter 5). One further disadvantage is the inability to directly 'edit' small portions the binary data, when using Zlib, while the schema binarisation approach allows that possibility. For example, if the document is stored in binary, then in order to simply insert additional data into the document (under the *any* node branch), the entire branch needs to be decompressed before the addition can be made, then recompressed afterwards. Where tree-based compression is used, the new data can easily be inserted without the need to decompress the entire branch.

In this example (see Figure 5.33), since the `Section` element is an *any* node and the following paragraph (`p`) element is not a global element of the corresponding namespace, hence tree-based schema based compression is not feasible, limiting the achievable efficiency.

5.6 Conclusion

This Chapter presented two new extensions to MPEG-B BiM to enhance compression on MPEG-21 DIDs that contain embedded resources or embedded XML.

This Chapter also demonstrated the effectiveness of combining RXEP with a binarisation technique (tree/schema compression) to efficiently transmit required parts of an XML document. Access to the required parts of the XML can be achieved by remotely navigating or querying the XML file. By eliminating unnecessary data transmission, binRXEP significantly reduces overall bandwidth and storage requirements. A comparison of binRXEP to other technologies is shown in Table 5.4.

This Chapter presented a novel technique whereby the structure from a corresponding XML Schema is embedded into the XML DOM instance to assist on-the-fly compression of user-tailored fragments as well as dynamic binarisation.

It was shown that through the use of binRXEP, significant savings can be obtained though the combination of retrieving only requested fragments together with tree-based XML compres-

Table 5.4 Comparison of BinRXEP with other technologies

	BinRXEP	Millau	XMill	XOP	BiM
User Defined Queries	Y	N	N	N	N
Retrieve up to n branch levels	Y	N	N	N	N
XML Fragments	Y	N	N	Y	Y
Random Access	Y	N	N	N	N
Supports Modify	Y	N	N	N	Y
Supports Delete	Y	N	N	N	Y
Supports Insert	Y	N	N	N	N
Protocol Valid to XML Schema	Y	N	N	N	Y
Partial Downloads	Y	N	N	N	N
Two-Way Editing	Y	N	N	N	N
Protocol can be Extended by user	Y	N	N	N	N
User defined Streaming	Y	N	N	N	N

sion. With the RXEP protocol defined wholly in XML, the protocol can thus be binarised, reducing the overheads of the binRXEP protocol (in some cases to less than one byte). This Chapter also demonstrated that a highly structured XML document achieve better compression ratios, whereas flat structured XML documents do not compress as well.

Finally, this Chapter also presented a novel technique to compress RXEP XPath locators using XML Schema information to reduce the number of bits required for their representation.

Chapter 6

XML Schema Exchange

6.1 Introduction

The review in Chapter 2 showed that XML Schemas can be used to impose restrictions on datatypes and structure of XML documents. Chapter 2 also illustrated the problem of the verbosity of XML documents. This problem of verbosity is also a problem for XML Schemas; where schemas provide more choices to cater for possible XML document structure and data (i.e., a choice between using Celsius or Fahrenheit for a temperature value). Furthermore, a single XML document can often link to several, potentially large XML Schemas.

This Chapter thus proposes a novel technique to allow users to request and maintain only partial views of an XML Schema. This new technique is advantageous for users who wish to modify sections of an XML document, only retrieving the XML Schema fragments needed to validate that section of XML, without the need to retrieve the entire XML Schema. The lack of research into the area of XML Schema negotiation techniques is possibly due to the assumption that the entire XML Schema is always required.

This Chapter proposes a new technique for representing RXEP XPath locators for XML Schema in a shorter form, through the application of simplification rules utilising the XML Schema rules.

Finally, this Chapter demonstrates that RXEP Schema techniques for community creation of XML Schemas is possible.

6.2 XML Schema Fragmentation

XML Schemas provide flexibility through the use of choices and sequences. Unfortunately, not all of these choices and sequences within the XML Schema are required by the receiving user as the XML document being retrieved may not utilise them. Furthermore, many XML Schemas contain elements that will not be required (i.e., choice between imperial and metric units).

For example, an XML Schema may contain video, audio and photograph elements (with choices between many encoding formats and bitrates) to describe an album within a community. A user joins the community who are sharing albums and is only interested in retrieving photographs and relevant metadata. The user wishes to also retrieve the XML Schema to ensure that all retrieved data is indeed valid, and typically, the user would need to retrieve the entire XML Schema even though only a very small portion is required for the photographs. Ideally, the user should be able to only retrieve the parts of the XML Schema which are relevant.

MPEG-7 [7] defined a technique to fragment and deliver XML Schema fragments; Schema Update Units (SUU) which are used to gradually build up the required XML Schemas in a local version on the user's terminal. However, MPEG-7 does not provide a mechanism to allow users to request fragments of XML Schema, and does not prodive a method to support the random access of XML Schema fragments - the user needs to wait for the necessary XML Schema fragments to be sent before the user can use them.

Since an XML Schema document has the restriction that it must be a well formed XML document, the same fragmentation and delivery techniques as previously presented in Chapter 4 can be used. This Section thus details a novel technique which allows user requested fragmentation of an XML Schema(s), whereby a user can receive only the portions of the XML Schema(s) relevant to the XML document being retrieved. Users can then maintain partial views of the XML Schema(s) locally, thus avoiding retrieval of entire XML Schemas.

Throughout this Section, an example XML Schema and XML instance will be used and these are shown in Figures 6.1 and 6.2 respectively. The XML Schema in Figure 6.1, illustrates a

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="mediaNS:2004"
           xmlns:ns1="mediaNS:2004">
 <xs:element name="Media">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="ns1:Music"/>
    <xs:element ref="ns1:Videos"/>
       ...
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="Music">
  <xs:complexType>
   <xs:sequence>
    <xs:element maxOccurs="unbounded" ref="ns1:Song"/>
     ...
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="Song">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Title" type="xs:string"/>
    <xs:element name="Description" type="xs:string"
                minOccurs="0"/>
    <xs:element name="Artist" type="xs:string"
                minOccurs="0"/>
    <xs:element name="Format" type="xs:string"/>
    <xs:element name="Length" type="xs:string"/>
    <xs:element name="MediaLocation" type="xs:string"/>
   </xs:sequence>
   <xs:attribute name="id" use="required"
                 type="xs:NCName"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 6.1 Example XML Schema for a Simple Media Album XML

simple schema describing a collection of media, however for brevity, only the <Song> node is detailed (which is a child of <Music>). Missing XML Schema elements from Figure 6.1 are illustrated by ellipsis. The XML in Figure 6.2 illustrates a simple XML document, valid to the XML Schema in Figure 6.1. This XML instance only utilises the Music part of the Media element.

```
<Media xmlns="mediaNS:2004">
 <Music>
  <Song id="Hit1">
    <Title>Hit.1</Title>
    <Description>Song 1</Description>
    <Artist>A. Artist</Artist>
    <Format>MP3</Format>
    <Length>02:23</Length>
  </Song>
  <Song id="Hit2">
    <Title>Hit.2</Title>
    <Description>Song 2</Description>
    <Artist>B. Artist</Artist>
    <Format>OGG</Format>
    <Length>03:46</Length>
  </Song>
 </Music>
</Media>
```

Figure 6.2 Example XML document, valid to the XML Schema as shown in Figure 6.1

6.2.1 Extending RXEP for XML Schema Fragmentation

Dynamic creation of user-side schemas requires fragmentation commands such as add, insert, update and delete; these are already provided by RXEP. However, to utilise the schema fragmentation strategies with RXEP, the user would need to open an RXEP connection and request the XML Schema fragments as XML fragments are delivered.

For example, a user selects an XML document from which to receive fragments, and the XML document links to several XML Schemas. If the user receives a fragment of XML, the user then needs to use RXEP (separate connection) to request the schema fragment from the appropriate XML Schema. In this scenario, each schema (and thus namespace) is maintained separately.

Instead of maintaining and requesting fragments using multiple RXEP connections and requests, it would be ideal to use a single RXEP response containing both the XML fragment and corresponding XML Schema fragments. Thus, in order to identify the correct XML Schema to perform these operations, the RXEP XML Schema requires some extensions to provide the additional information.

Client-side Placement of XML Schema Fragments

In order for the client to be able to place a schema fragment within the correct namespace, the RXEP responses need to be able to specify the schema namespace. This is achieved by appending a new attribute `schemans` to the RXEP XML Schema - which defines the target XML Schema namespace to apply the RXEP operations. The `schemans` is added to the RXEP response commands (i.e., add, delete, update and insert).

Client XML Schema Fragment Requests

To enable users to request an XML Schema fragment for a given node, it is necessary to identify that the XML Schema fragment for that request (and not the XML fragment) is to be delivered; the RXEP request was thus modified to specify an additional, optional, attribute `getSchema`. The new boolean attribute `getSchema` is added to the `queryType` to provide a client with a method to request an XML fragment as well as its corresponding XML Schema fragment using only one request.

In addition to the attribute `getSchema`, an additional attribute `keepValid`, is also made available to the client. The `keepValid` attribute is only available if the `getSchema` attribute is used, and specifies that all nodes of the XML Schema (back to the root) required to keep the XML fragment valid are requested. For example, ancestor nodes with minOccurs greater than one would be retrieved. With this technique, the client has enough information to request further XML fragments in order to keep them valid with respect to the schema.

Querying an XML Schema

To provide the client with the ability to query an XML Schema (as linked by the open XML document), an additional RXEP command, `SchemaQuery`, is added as shown in Figure 6.3.

```
<xs:complexType name="requestType">
  <xs:sequence>
    <xs:element name="Src" type="srcType"
                minOccurs="0"/>
      <xs:choice minOccurs="0">
        <xs:element name="Query" type="queryType"/>
        <xs:element name="SchemaQuery"
                    type="schemaQueryType"/>
        <xs:element name="XMLPull" type="xmlpullType"
                    maxOccurs="unbounded"/>
        <xs:element name="Stream" type="streamType"/>
      </xs:choice>
  </xs:sequence>
</xs:complexType>
```

Figure 6.3 Modified RXEP requestType to provide the new SchemaQuery command to the RXEP XML Schema

```
<xs:complexType name="schemaQueryType" >
 <xs:complexContent>
   <xs:extension base="queryType">
    <xs:attribute name="forXML" type="xs:boolean"/>
    <xs:attribute name="keepValid" type="xs:boolean"/>
    <xs:attribute name="nameSpace" type="xs:string"/>
   </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Figure 6.4 The new schemaQueryType command to the RXEP XML Schema

Figure 6.4 shows the `schemaQueryType` which is the definition for the `SchemaQuery` command. The `schemaQueryType` extends the RXEP `queryType` and adds two new attributes; `forXML` and `nameSpace`. If the `forXML` attribute is set to 'true', then the XPath expression is used to match the node in the XML document (instead of the XML Schema document), and the associated XML Schema node for the matched XML node is delivered back to the client. As with the RXEP query command, the `SchemaQuery` definition also contains the `keepValid` attribute. The `nameSpace` attribute for the `schemaQueryType` allows the target namespace to select which XML Schema document to apply the the XPath expression; this allows a client to query a namespace directly, without requiring a new RXEP connection to be established for the remote XML Schema document.

With these extensions applied to RXEP, it provides the user with three options for the reception of both XML and XML Schema fragments: client requested, progressively sent (server determined) or a combination or server determined and user requested. These approaches will now be investigated in more detail.

6.2.2 Client Requested XML Schema Fragments

In the first technique, a client can retrieve XML Schema fragments simply by requesting fragments from the server. However, instead of initiating separate RXEP connections for all linked schemas of a particular XML document, the schema extended RXEP version can be used.

The user can now initiate an RXEP connection to the server, and reuse that connection to also request XML Schema fragments from the schemas referenced by the XML document. Figure 6.5 illustrates a block diagram depicting how a client can request XML and XML Schema fragments.

It is now possible that when a user randomly accesses an XML fragment (and thus its corresponding XML Schema fragment), the ancestor elements of the selected XML Schema fragment may not have been received. In this case, the required ancestors can be traced back to the XML Schema root element, and delivered to the user.

In this case, the user 'asks' the server for the fragments it does not have. A client may wish

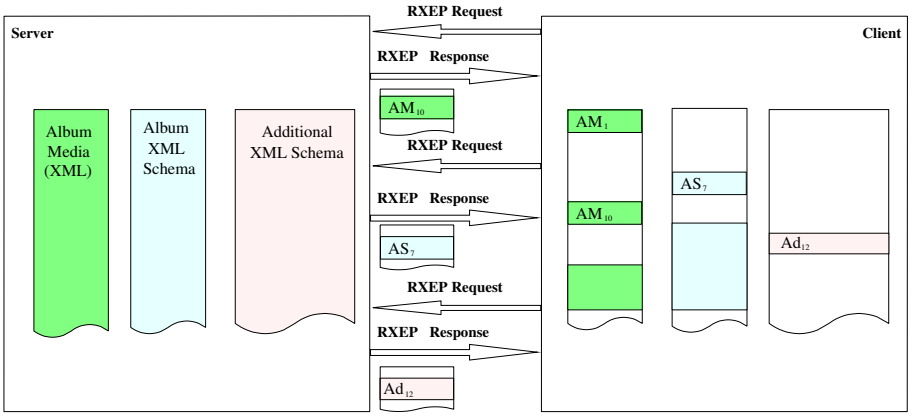

Figure 6.5 Block diagram illustrating how clients can request XML fragments as well as XML Schema fragments

to use this method in the following situations:

- Request schema fragment for the last XML fragment - in this case the user has been receiving fragments of XML, but then receives a fragment which the client does not have a local copy of the corresponding schema fragment.
- Request schema fragment based on XML query - the user may have queried a remote XML document and received one or multiple fragments in one request. The user can then ask the server for any schema fragments corresponding to a given Query (i.e., RXEP query using XPath). This may well be the original query or, a different query if the user already has schema fragments for some of the multiple fragments received; and
- Request a schema fragment directly from schema - the user can query the original schema directly to receive the desired fragment(s).

For example, a user is employing RXEP to a receive XML fragments from a remote XML document containing the users favourite music albums. At this point, the have only been receiving XML fragments, and does not have the corresponding XML Schema fragments. The user discovers that there is a mistake in the title of one of the songs, and wishes to fix this error. Using the schema extended RXEP, the user can now request the XML Schema fragment corresponding to the title element, to ensure that any corrections will still conform

```
<RXEP>
 <Request>
  <SchemaQuery forXML='true'>
    /Media/Music/Song[2]/Title
  </SchemaQuery>
 <Request>
</RXEP>
```

Figure 6.6 Example of an RXEP request to ask the server for the the XML Schema required for a specified XML Node

to the schema. Now, the user would construct an RXEP request as shown in Figure 6.6. Unlike the standard RXEP approach used in Chapter 4, the client sends the new `forXML` attribute set to 'true'. This informs the server that the XML Schema fragment, which is needed to validate the selected XML fragment as specified by the XPath locator, is to be sent to the client. The server receives the RXEP request as shown in Figure 6.6 and generates the RXEP response as shown in Figure 6.7. This fragment contains all necessary ancestor XML Schema information back to the XML Schema's root node.

```
<RXEP>
 <Response>
   <Add location="/" schemans="mediaNS:2004"
     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="mediaNS:2004"
           xmlns:ns1="mediaNS:2004">
       <xs:element name="Song">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="Title" type="xs:string"/>
          </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:schema>
   </Add>
 </Response>
</RXEP>
```

Figure 6.7 Example RXEP response containing the XML Schema fragment in response to the RXEP request in Figure 6.6. All ancestor schema information back to the root node is also sent to the client

6.2.3 Server Determined XML Schema Fragments

In addition to a client requesting XML Schema fragments (see Section 6.2.2), the client can request that the server deliver the requested XML fragment and the corresponding XML

Schema fragment within a single RXEP response. For example, when a fragment of XML is requested, the relevant fragment of XML Schema is also packaged inside the RXEP response. This concept is further illustrated in Figure 6.8, where the server is holding the XML document and the corresponding XML Schemas of which the client is requesting. As illustrated in Figure 6.8, RXEP delivers small XML fragments and corresponding XML Schema fragments, hence building a partial document and partial XML Schema(s) locally at the client. Additionally, intelligent servers may preempt usage (either based on current usage, or statistics of previous users), and deliver additional fragments of XML and/or schema upfront.

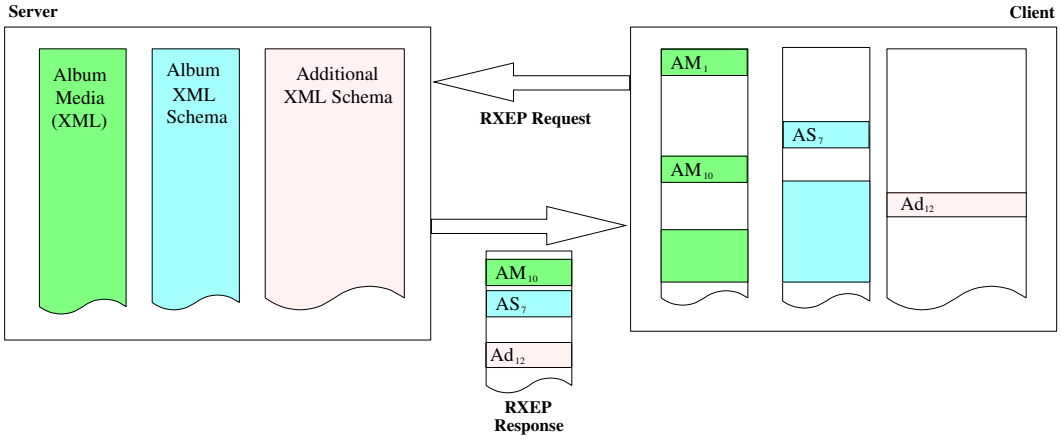

Figure 6.8 Block diagram illustrating how many XML Schema fragments (selected by the server) and the requested XML fragment can all be contained within a single RXEP response

Consider for example, a user who has previously navigated through a remote XML document as shown in Figure 6.2 using RXEP, also requesting corresponding the schema fragments. The user now wishes to request the title of the second song node, and constructs an RXEP request as shown in Figure 6.9. The RXEP response (containing the XML fragment and the corresponding XML Schema fragment), is shown in Figure 6.10. The user has now received the requested fragment as well as the fragment of XML Schema (required for validation of that fragment), within a single RXEP request.

6.2.4 Combination of Server Determined and Client Requested

The final technique by which a user can receive fragments of XML Schema is a combination of both client determined (see Section 6.2.2) and server determined (see Section 6.2.3). This technique is advantageous for users who wish to store or cache XML Schema fragments from

```
<RXEP>
 <Request>
  <Query getSchema='true'>
   /Media/Music/Song[2]/Title
  </Query>
 </Request>
</RXEP>
```

Figure 6.9 Example of an RXEP request specifying that the user wishes to receive both the XML Fragment and the corresponding XML Schema

```
<RXEP>
 <Add schemans="mediaNS:2004"
      location="xs:Schema/xs:Element[@name=\"Song\"]/
xs:complexType/xs:Sequence/xs:Element[@name=\"title\"] ">
    <xs:element name="Title" type="xs:string"/>
 </Add>
 <Add location="/Media/Music/Song[2]/>
   <Title>Hit.2</Title>
 </Add>
</RXEP>
```

Figure 6.10 Example of a fragment of XML and corresponding XML Schema contained within a single RXEP packet, as a result from the RXEP request from Figure 6.9

previous RXEP navigation or queries. In this case, users do not need to download already retrieved XML Schema fragments by only requesting the fragments they do not have. If a user then encounters a subbranch of XML Schema they do not currently possess, then the user can switch over to server determined fragment retrieval to automatically retrieve fragments for that subbranch.

For example, consider a community of users who share a common XML Schema to provide descriptors and structure for storing holiday information and metadata. A user from outside the community searches XML documents from within this community and finds a few XML documents which may be of interest. After the user has navigated through the first XML document from the search using server determined XML Schema fragment retrieval, the user has already received and stored a number of XML Schema fragments locally. Now, if the user navigates through another document (located on another peer), the user can decide to switch to client request, and only request XML Schema fragments for elements that have not previously been retrieved. If a user then encounters a subbranch of interest and has not yet received any fragments for that subbranch, the user can switch back to server determined, and begin navigation through the new subbranch, retrieving the relevant XML and XML Schema fragments together.

6.3 RXEP Schema XPath Locators

Section 4.3.3 described RXEP XPath locators, which are used for exact paths within an XML Document. The `location` attribute of the RXEP response commands contains the RXEP XPath locator, which plays a vital role in RXEP response commands (such as add a new fragment of XML).

Whilst the RXEP XPath locators can be used for the placement of XML Schema fragments, they can prove to be lengthy expressions. For example, and referring to Figure 6.1, if the `Title` element is to be referenced, the standard RXEP XPath locator would be:

```
/xs:Schema/xs:Element[@name="Song"]/xs:complexType/xs:Sequenc
e/xs:Element[@name="title"]
```

Fortunately, some assumptions can be made about the XML Schema to reduce the complexity of the RXEP XPath locators, which will be referred to as RXEP Schema XPath locators. Utilising some of the rules defined in XML Schema, and considering RXEP query and navigation commands in particular, the following assumptions and simplifications can be applied:

- When requesting XML fragments, only elements are selected, and thus, the locators can be restricted to specify only elements (i.e., XML schema attributes cannot be selected);
- Following from the above point, there is no need to specify the XML line `<element name="name">`, but rather specify just the defined name. For example in this case it would be `/name/`;
- A modelgroup (i.e., choice/sequence/all) node cannot contain multiple child elements with the same element name [3] (they must be unique);
- To avoid confusion with elements which have the same name as a modelgroup, modelgroup name will be enclosed within square brackets []. For example [Sequence];
- Modelgroup [3] nodes (i.e., choice, sequence and all) must be declared inside a complexType tag and thus the complexType portion can also be omitted. For Example the following `/Song/xs:complexType/[Sequence]/` can be reduced to `/Media/ [Sequence]`; and
- The first schema element (i.e., `<xs:Schema>`) can be omitted as this must be present to represent a valid schema.

Whilst observing these rules when using RXEP to partially receive XML Schemas, the example RXEP Schema XPath locator from our previous example can be reduced as illustrated in Figure 6.11.

6.3.1 Experimental Results

To fully explore the possibility of XML and XML Schema fragmentation, a client and server application was written in JAVA (see Appendix A). Results in this Section are given by way

```
/xs:Schema/xs:Element[@name="Song"]/xs:complexType/xs:Seque
nce/xs:Element[@name="title"]
```

```
/xs:Song/[xs:Sequence]/xs:Title
```

Figure 6.11 Example reduction of an RXEP XPath locator when applying the RXEP Schema XPath simplifications

of some example scenarios.

6.3.1.1 Scenario One

As a simple example, a user begins a new RXEP session to a remote device, and requests the file `embPic.xml`. This file is a small and very simple Digital Item, containing only one embedded picture and title descriptions using MPEG-21 definitions only and is originally 15,557 bytes in length. The DIDL XML Schema, referenced by the XML document is 14,014 bytes.

The user navigates to the description of the remote XML document, also requesting the XML Schemas for each selected node to be transmitted. Both the XML fragments and corresponding XML Schema fragment are thus packages within one RXEP response.

At this point, the user has received 3,357 bytes and transmitted 848 bytes. The received total includes the XML Schema fragments, and a screenshot of the XML Schema representation of fragments are shown in Figure 6.12 (Note: this screenshot represents the XML Schema as stored in memory on the clients application where XML Schema reference elements are not shown. This representation is saved to an XML Schema document when the client has finished).

Now, the client has downloaded and uploaded a total of 4,205 bytes. The total filesize of both the XML document and DIDL XML Schema is 29,579 bytes, thus, the client has saved receiving 25,375 bytes of data when compared to downloaded both files. This illustrates the savings for just one referenced XML Schema, and demonstrates the potential savings that

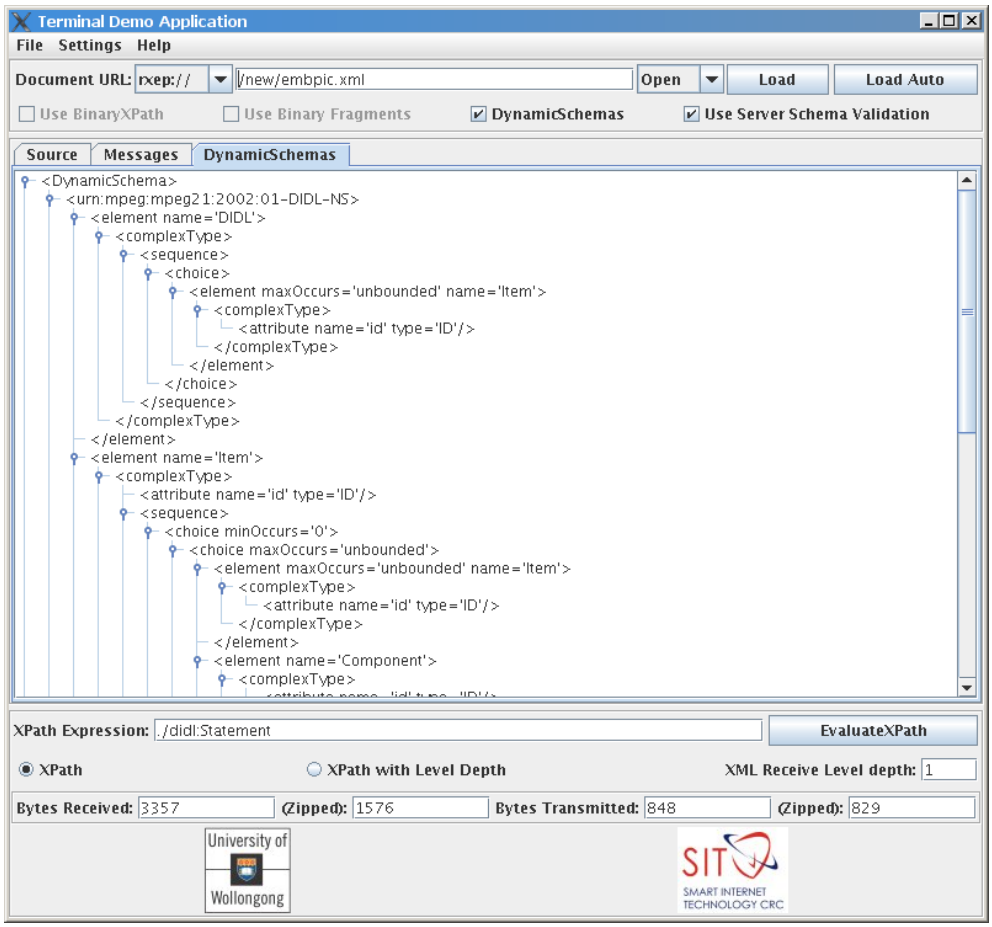

Figure 6.12 Screenshot of the client application's representation of the fragments of XML Schema received

can be achieved, especially when more then one XML Schema is referenced by an XML document.

6.3.1.2 Scenario Two

In a second scenario, consider a user browsing though an MPEG-21 DID (XML document) on their mobile device in a low bandwidth environment. Upon browsing using RXEP, the user notices that an audio track descriptor is missing the song title and wishes to make the alteration. Figure 6.13 shows a partial view of the XML which is missing the data. To ensure that the newly added metadata is conformant to the XML Schema, the user then issues an RXEP request to retrieve the `mpeg7:Title` fragment from the MPEG-7 XML Schema to ensure that the newly entered data in the XML remains valid and conformant to the XML Schema. The user receives the relevant MPEG-7 XML Schema fragment which is 1,410 bytes in size, the user has thus avoided downloading the entire MPEG-7 schema (which is 528,841 bytes) when only a very small part of of the schema was required.

```
...
<Statement mimeType="text/xml">
  <mpeg7:Mpeg7>
   <mpeg7:Description xsi:type="CreationDescriptionType">
    <mpeg7:CreationInformation id="beethoven5th">
      <mpeg7:Creation>
       <mpeg7:Title type="songTitle">
          BLANK
       </mpeg7:Title>
       <mpeg7:Title type="albumTitle">Classical Favs
       </mpeg7:Title>
       <mpeg7:Creator>
...
```

Figure 6.13 Portion of the DID XML document received by the user, where the song title of the track is missing

6.4 XML Schema Generation

Many descriptors are possible for the description of multimedia content, and consequently, many XML Schemas have been generated. Some descriptors have been standardised such as those descriptors found in standards such as MPEG-7 [7] and MPEG-21 [28]. Furthermore,

some users find these standardised descriptors inadequate and prefer to create their own descriptors or extend current sets, and have thus generated user defined schemas. Increasingly, communities of users are sharing and accessing content on the Internet. In such user groups, there is increasing usage of accompanying metadata which allows the community to label the content according to shared opinions of metadata requirements and hence schemas.

Often, many XML Schema documents are created by once source and standardisation efforts of XML Schemas are long, arduous tasks which can take several years. For example, MPEG-7 first started in 1998 and is currently still undergoing changes to the XML Schema. Furthermore, the process to get descriptor updates into the MPEG XML Schema requires the following steps:

1. A set of proposed descriptors are submitted;
2. Newly proposed descriptors are reviewed and discussed;
3. Voting on the set (or subset) of descriptors; and
4. Adding or modifying the descriptors in the 'standard' XML Schema.

This process alone takes many months, especially since MPEG only meets approximately every three months. Furthermore, step 2 usually takes several iterations (i.e., separate MPEG meetings) to refine the descriptors and accompanying documentation. Step 3 requires two meetings to have passed before votes are retrieved and counted. Finally, if the vote was in favour of the descriptors (or a modified set), then these descriptors are appended to the standard.

Often when there are multiple people providing input into e.g., a metadata set, ideas are sparked from other ideas (the basis of brainstorming); in XML Schema creation, this includes both the descriptors and the structure of the descriptors. Unfortunately, with a process like MPEG, people need to be present at the meetings to be fully involved in the discussions; many people can not make it to every meeting or are involved within other MPEG activities. Some of these absent people may have good ideas which inspire ideas from other users or

find mistakes in descriptors, and these ideas or corrections may not be received until the next meeting. Consequently, this results in a standard taking much longer to become stable.

A possible solution to reduce times to append or amend descriptors within a standard is to allow multiple users, connected to the internet, to all add to a schema where other users can see the updates in real time (or close to real time). After users have finished adding descriptors (e.g., this could be a week or so), voting on the set to be standardised can begin and redundant descriptors can be removed.

6.4.1 Collaborative XML Schema Generation

Section 6.2.1 presented a novel technique which utilises RXEP to retrieve only the required XML Schema fragments. Section 4.7 illustrated the advantages of RXEP in a collaborative environment when editing XML documents. Since an XML Schema is also a valid, well-formed XML document, the collaborative editing techniques can be combined with the schema fragmentation ideas in Section 6.2.1; this now allows users to execute remote operations such as add, insert, update and delete nodes on a remote XML Schema. One advantage of collaboratively creating an XML Schema, is that as the XML Schema grows other users may instantly notice missing descriptors, or trigger ideas for other descriptors which may have otherwise been missed.

A collaborative XML Schema generation technique would be beneficial in situations where an XML Schema is generated requiring contributions from many people. Also, with the increase in online communities where people are from all over the world, the need for an online collaborative technique for XML Schema generation becomes necessary.

For example, Section 6.4 illustrated the delays required to get descriptors into an MPEG standard. Through the use of online XML Schema collaboration software, users who are unable to physically attend meetings can still contribute and monitor the creation of XML Schemas, almost as if they were at the meeting.

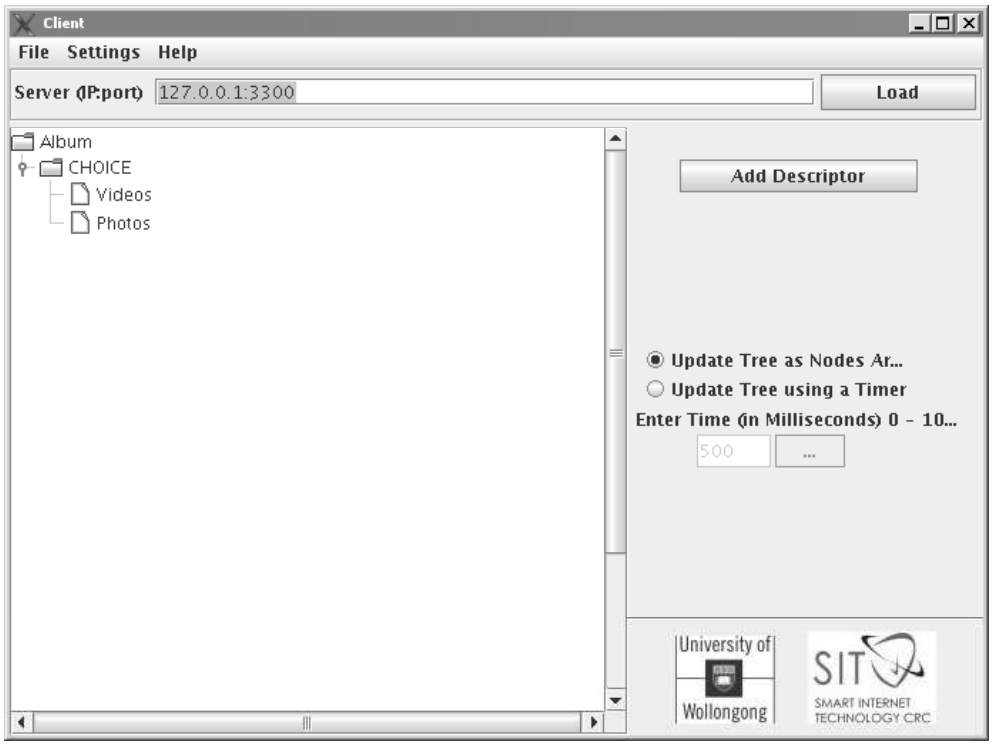

Figure 6.14 Screenshot of the client JAVA application used for the schema experiment

6.4.2 Experimental Results

The aim of this experiment was to determine if the RXEP Schema techniques (see Section 6.2.1) can be used for the creation of an XML Schema generated in a community scenario (consisting of a small group). For this experiment, users were only permitted to add descriptors to the XML Schema, and the interface (see screenshot in Figure 6.14) was such that the user did not need to be familiar with the finer details of XML Schema.

The following setup was used for the experiment:

- One server - An application written in JAVA (for more information see Appendix A), which was used to track all connected clients. When a client adds to the XML Schema, the change was also pushed out to all connected clients (using RXEP) to reflect the addition.
- One or more clients - Each client runs a small JAVA application which provides the user with a basic interface, which allows a users to select a node displayed in the tree

structure. Selecting the 'Add Descriptors' button, presents the user with a dialog box to create a list of nodes. After the user has accepted the new descriptors in the dialog box, this list of nodes is transmitted to the Server (using RXEP), and changed are updates in the client's application.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Album">
        <xs:complexType>
            <xs:choice>
                <xs:element name="Videos"/>
                <xs:element name="Photos"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 6.15 Initial XML Schema used as the starting point for users to add their descriptors to. This XML Schema specified a root element of Album which a choice of two child nodes, Videos and Photos

Using this setup, five people were given a set of photos which all had a common theme of landscapes (the set of photos can be found in Appendix C). An initial XML Schema, as illustrated in Figure 6.15, was used as the starting point for all the users to begin creation of XML descriptors to describe the landscapes in the sample set of photos. The initial XML Schema was used to give users a sample of how descriptors can be grouped and structured, and to prevent users adding descriptors to the root element (i.e., flat structure being generated). It is expected that restricting the photo set to just landscapes limits the set of descriptors being generated. If the photoset was not limited, then users could potentially each create separate branches of descriptors with no collaboration between each other.

To shield users from the specifics of XML Schema, there are no options to add Choices, Sequences and All nodes. This is a deliberate attempt to determine if the protocol itself is a viable technique for the collaborative creation of a schema, otherwise, users would need to learn and understand XML Schema before participating in the experiment. All nodes in these case, with be assumed to have be appended using the Choice modal group element.

The XML Schema was presented to users within the JAVA client as a 'tree' and was updated

as other users made changes. A screenshot of the client is shown in Figure 6.14.

The experiment ran for fifteen minutes and the resulting schema is illustrated in Figure 6.16. Note that this is the structure as represented by the JTree for ease of viewing (i.e., all complexType etc nodes are removed).

Figure 6.17 illustrates the total number of descriptors added to the schema per user during the experiment. As Figure 6.17 shows, three of the five users contributed fourteen or more descriptors each whereas users A and E only contributed ten and eight descriptors, respectively. Figure 6.18 illustrates the number of updates per client over time.

Figure 6.18 shows that user E added descriptors only within the first minute, however, the initial contribution to the schema was six descriptors followed by two descriptors. User B had been adding only one descriptor at a time for the first minute, but, after user's E and user C's (first addition) did B add four descriptors within one RXEP packet. Interestingly, user B seemed to add multiple descriptors after other users had also added two or more descriptors in one request. User D generally preferred to add one descriptor at a time but only during the middle of the experiment. User C was consistent in adding descriptors during the experiment, with a number of two and four descriptors updates in one RXEP packet.

Figure 6.19 illustrates the number of updates versus the leveldepth at which the descriptors were added. Since the users given the starting point for the schema (as shown in Figure 6.15), the users did not add descriptors to leveldepths one (the root node, which can only have one element) or two (i.e., child nodes of the root). As shown in Figure 6.19 the majority of descriptors were added to leveldepth four, which were mainly used for storage of data (e.g., coordinates for resolution). Descriptors added to leveldepth three were generally used for structural information (e.g., season or location). Interestingly, the number of users did not go beyond leveldepth five, which only consisted of three updates (near the end of the experiment).

Overall, the descriptors created made sense and demonstrate that a schema can be created in a collaborative situation. From the plot in Figure 6.18 after a number of descriptors were added (within one RXEP packet), there usually followed a cluster of updates; this shows the

```
<xs:element name="Photos">
   <xs:element name="Location">
     <xs:element name="GPS"/>
     <xs:element name="Text"/>
     <xs:element name="Longitude"/>
     <xs:element name="Latitude"/>
     <xs:element name="CoordinateSystem"/>
     <xs:element name="Place"/>
   </xs:element>
   <xs:element name="TimeofDay">
     <xs:element name="Morning"/>
     <xs:element name="Afternoon"/>
     <xs:element name="Evening"/>
   </xs:element>
   <xs:element name="Time" />
   <xs:element name="location"/>
   <xs:element name="Season">
     <xs:element name="Autumn"/>
     <xs:element name="Spring"/>
     <xs:element name="Summer"/>
     <xs:element name="Winter"/>
     <xs:element name="Wet"/>
     <xs:element name="Dry"/>
   </xs:element>
   <xs:element name="Country">
     <xs:element name="City"/>
     <xs:element name="Province"/>
     <xs:element name="Village"/>
   </xs:element>
   <xs:element name="Culture" />
   <xs:element name="Religion"/>
   <xs:element name="Elevation"/>
   <xs:element name="Year" />
   <xs:element name="TouristLocation" />
   <xs:element name="Hotels"/>
   <xs:element name="Author">
     <xs:element name="FirstName"/>
     <xs:element name="LastName"/>
     <xs:element name="Address"/>
   </xs:element>
   <xs:element name="ShutterSpeed"/>
   <xs:element name="LandscapeType">
     <xs:element name="MountainView"/>
     <xs:element name="Ocean"/>
     <xs:element name="Metropolitan"/>
     <xs:element name="Rural"/>
     <xs:element name="SeaSide"/>
     <xs:element name="Desert"/>
     <xs:element name="Architectural"/>
   </xs:element>
   <xs:element name="Holiday"/>
   <xs:element name="Business"/>
   <xs:element name="FileSize"/>
   <xs:element name="PredominantColor"/>
   <xs:element name="Family"/>
   <xs:element name="Persons">
     <xs:element name="Person">
       <xs:element name="LastName"/>
       <xs:element name="FirstName"/>
       <xs:element name="NickName"/>
     </xs:element>
   </xs:element>
   <xs:element name="SurroundingEnvironment">
     <xs:element name="Hot"/>
     <xs:element name="ReallyHot"/>
     <xs:element name="Cold"/>
```

```
    <xs:element name="Freezing"/>
     <xs:element name="UnbearablyHot"/>
     <xs:element name="Humid"/>
   </xs:element>
   <xs:element name="Camera">
     <xs:element name="Model"/>
     <xs:element name="Make"/>
   </xs:element>
   <xs:element name="Resolution">
     <xs:element name="X"/>
     <xs:element name="Y"/>
   </xs:element>
   <xs:element name="Date"/>
   <xs:element name="Pricing"/>
   <xs:element name="Filetype">
     <xs:element name="compressionType"/>
   </xs:element>
</xs:element>
```

Figure 6.16 Output from the collaboratively edited XML Schema experiment created by five people using the RXEP techniques. Note that this is not a valid XML Schema as the complexType and choice elements have been removed for ease of viewing for the people in the experiment

collaborative nature, where ideas lead to other ideas. The resulting schema was generally flat in structure, however, this may be due to no other users adding highly structured descriptors earlier in the experiment, to give ideas to other users.

6.5 Conclusion

This Chapter proposed a novel technique to fragment and deliver user requested XML Schema fragments. This technique allows users to either request parts of an XML Schema directly or to request XML fragments and corresponding XML Schema fragments. This technique has advantages especially for limited mobile devices, as it allows users to retrieve fragments of XML without the penalty of downloading and/or storing large XML Schema documents.

This Chapter also demonstrated that utilising RXEP schema techniques to generate an XML Schema in a collaborative environment is possible. From the experiment a usable schema was generated which could be used to describe the set of photos given to the users. The experiment also showed groups of additions after a number of two or more descriptors were added, demonstrating the collaborative nature of the experiment.

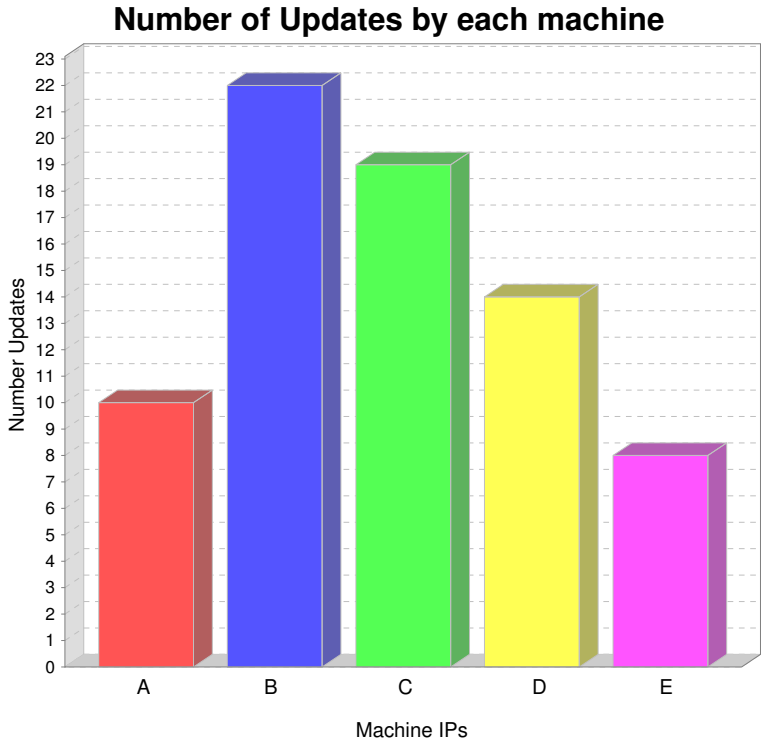

Figure 6.17 Illustrates the number of updates per user during the XML Schema experiment

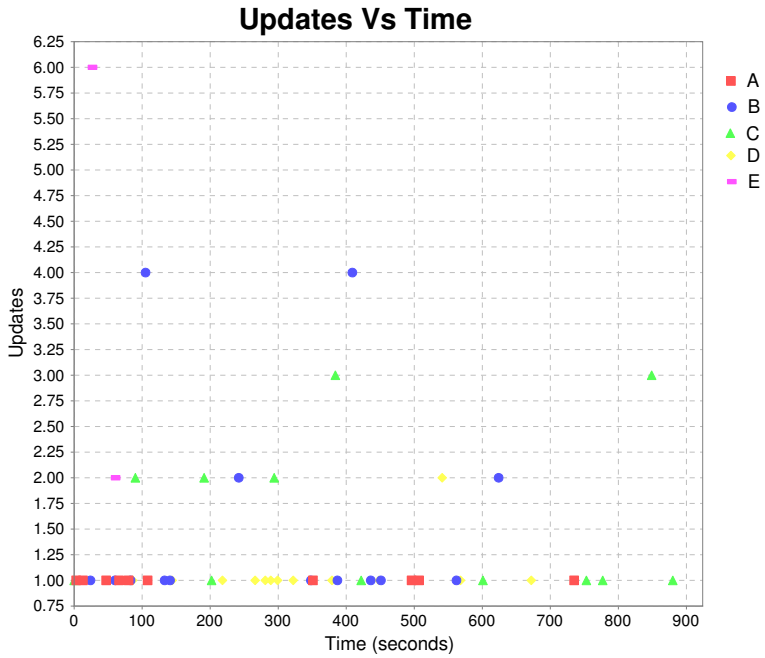

Figure 6.18 Illustrates the number of updates, over time, by each user during the XML Schema experiment

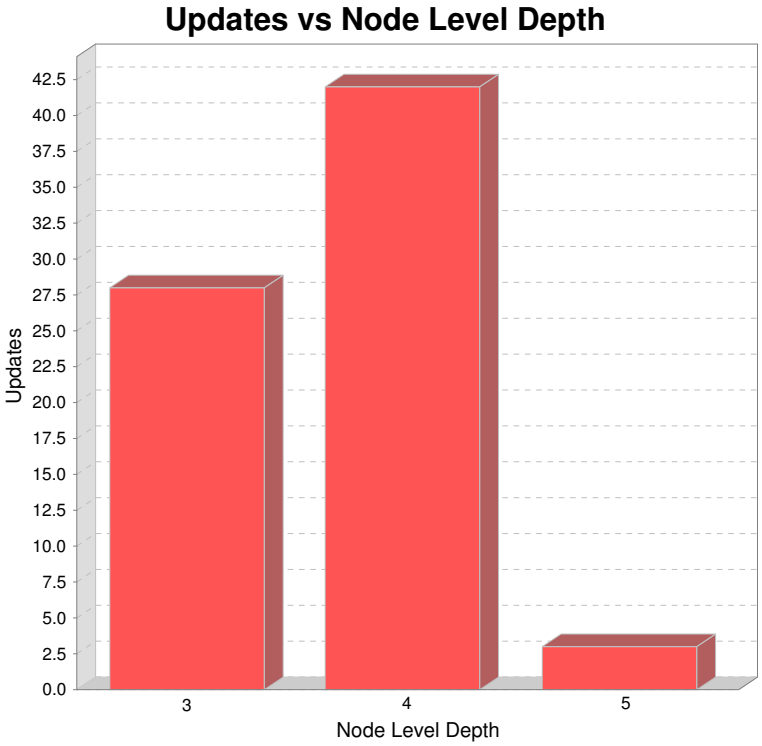

Figure 6.19 Plot of the number of updates vs the level depth of the added nodes

Chapter 7

Conclusions and Future Work

This Chapter presents the main conclusions as reported from previous Chapters. Finally, this Chapter proposes possible future work.

7.1 Conclusions

XML is increasingly being used for the storage and representation of data. When raw data is transformed into an XML document, there is often a significant in terms of filesize due to the added structural information. This verbosity consequently increases file sizes resulting in larger downloads for users, and hinders users in mobile environment where bandwidth is much slower and usually charged per kilobyte downloaded. Furthermore, XML Schemas which are needed for successful validation of XML documents can often be large, and XML documents may link to many XML documents; for successful validation users require the XML Schemas to be present.

The following main conclusions from this thesis are as follows:

XML Protocol for Mobile Devices

This thesis detailed a new protocol called RXPP, which is designed to be small and lightweight for use on mobile devices. RXPP provides users with the ability to navigate through remote XML documents by delivering the structure for selected nodes. Through navigation, users are able to traverse the XML document selecting relevant nodes and data, whilst not retriev-

ing data that is not desired. Mobile devices typically operate within low bandwidth environments where data is often charged per kilobyte download. By utilising RXPP in mobile environments, users can save both time and money by only retrieving the data that is actually required.

RXPP also provides random access into XML documents by utilising XPath locators, such that a user can begin navigation from any point within the XML document, without the need to download the XML up to that point in the document.

RXPP is also beneficial on mobile devices that have limited processing and memory. RXPP effectively pushes the processor and memory intensive tasks such as traversing DOM trees onto the server.

A Two-way XML Exchange Protocol

Through extension of RXPP, this thesis produced another protocol called RXEP designed to provide users with a generic and flexible approach for the complete two-way exchange of XML documents. RXEP allows users to download fragments of XML through both navigation and querying techniques. In many cases, a user can generate a well crafted query and request all the relevant XML data within one RXEP request. RXEP also allows users to modify remote XML documents where the user does not need to possess the entire XML document in order to make modifications. Online collaboration of XML documents (such as office documents like MSOffice and Openoffice) becomes possible when using RXEP where many users can modify a remote XML document.

RXEP locators were developed to allow clients to precisely position fragments of XML in their own local version, where previous XML nodes had not been retrieved. RXEP locators extend XPath methods to provide extra information such as the nodes absolute location and total number of sibling nodes. RXEP locators allows clients to retrieve fragments of XML whilst replicating the exact structure of the original XML document.

This thesis demonstrated that RXEP can provide significant savings in terms of data transferred (upload and download) by retrieving user requested data. This thesis also demonstrated that flat structured design of XML Schemas can have adverse effects on the savings obtained

with using remote navigation techniques.

Binarised RXEP

This thesis performed some tests on several MPEG-21 Digital Item Declarations (DIDs) (XML documents) and found that tree-based XML compression algorithms (such as MPEG-B BiM) performed the best when compared to traditional redundancy compression algorithms. These tests also highlighted some deficiencies in BiM, and thus, some new BiM extensions were created to provide successful compression and improve compression for certain DIDs.

Binarised RXEP (binRXEP) was then investigated to reduce both the size of the protocol and the data within the XML fragments. Unlike BiM, binRXEP cannot binarise fragments in advance as each client can define different fragments of XML; thus the creation of Schema Document Object Model (SDOM). SDOM is a novel technique created in this thesis which transforms the XML document into its DOM representation and merges in appropriate XML Schema structural information. By using SDOM, XML Schema nodes and XML nodes are also linked to speed up compression of XML fragments and validation of newly inserted XML fragments.

Binarised XPath locators were then proposed to offer the advantages of binarisation to RXEP XPath locators. Binary XPath locators can significantly reduce the length of the RXEP locator as most of the locator provides structural information, which are best compressed using tree-based algorithms.

It was shown that binRXEP significantly reduces the overheads added by RXEP, and in some cases down to less than a byte. BinRXEP compression ratios depend heavily on the structure of XML Schema, but in many cases, offer a significant reduction in total data transferred by the user.

XML Schema Fragmentation and Delivery

XML Schemas are used to validate XML instances as the schemas provide information about how the XML document should be structured and what datatypes can be used to represent

data. Like XML documents, XML Schemas are text documents and also valid XML documents. Utilising RXEP techniques, this thesis presented a new approach which provides users with the ability to request XML Schema fragments. RXEP schema techniques allows users to either query the XML Schema directly utilising the already established RXEP connections or to deliver XML Schema fragments corresponding to the XML fragments requested, packaged together within a single RXEP response. RXEP schema thus allows users to connect to a remote device, and receive only relevant XML and XML Schema fragments whilst not retrieving the unwanted data.

This Chapter also investigated RXEP Schema Locators which utilise XML Schema rules and information to significantly reduce the size of these locators.

Finally, this thesis demonstrated that the RXEP schema techniques can be used for successful collaboration for the creation of an XML Schema. With this technique, users can all contribute to the creation of the schema in realtime, while seeing the progress of other users. This collaborative creation of schemas can lead to quicker creation of XML Schemas. As descriptors are added to the XML Schema, users might extend or generate new ideas, thus resulting in a richer set of descriptors. This thesis demonstrated through a simple experiment, that people can collaboratively generate a reasonable rich and detailed XML Schema utilising RXEP techniques within a short time of fifteen minutes.

7.2 Future Work

This thesis presented techniques for the efficient delivery of XML and XML Schema. Possible future work based on this thesis is:

- Techniques to fragment XML documents that include fragmented media. Users could search the video and retrieve linked metadata, or a user could search for metadata and receive the relevant media chunk; this concept is illustrated in Figure 7.1. For example, an annotated (e.g., using MPEG-7 descriptors) video clip could deliver the video in chunks along with the XML descriptors (e.g., information about actors and objects). Furthermore, users could search through XML descriptors for a particular actor, and

receive the XML fragments as well as the relevant media chucks;

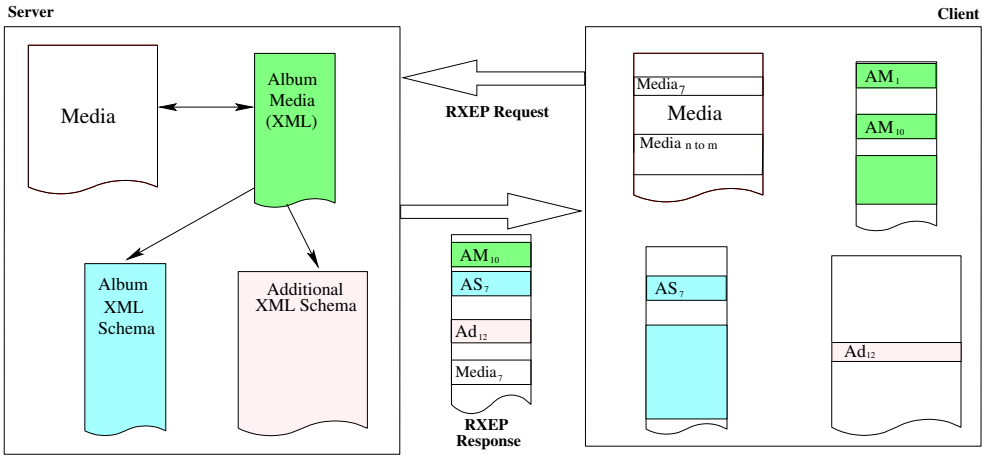

Figure 7.1 Block diagram illustrating

- Ontologies can be used to aid in the evolution of schemas. With many users contributing schema fragments to a schema, an intelligent system which can detect fragment duplication, differences between similar fragments so that they can be merged and duplicate XML fragments can be removed;

- Extending RXEP techniques to negotiate languages between peers from scratch utilising XML Schemas and ontologies;

- Integration of RXEP techniques with search engines. A user may use a search engine which results in a number of XML documents matching a given query, and return RXEP locators so that users only retrieve the relevant XML fragments from the XML documents in the result set;

- Combining RXEP techniques with knowledge bases. Users could extend RXEP to also query XML knowledge bases (e.g., OWL and RDF) when parts of an XML is not understood. Users would then receive XML fragments as well as fragments of relevant knowledge to give a better understanding of the received XML fragment; and

- Application of RXEP schema techniques in peer-to-peer networks where a tracker could be used to track fragments of XML Schema throughout a community (using RXEP XPath locators). Within each community, a ‘virtual schema’ exists that consists

of XML Schema fragments located on many peers. Fragments of XML in the virtual schema may exist as parts of another schema on each client. As fragments within a community become obsolete, they can be removed from the tracker.

Bibliography

[1] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yereagu, "Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation," Online: `http://www.w3.org/TR/REC-xml/` [Last Accessed: 21/07/2006], 04 February 2004.

[2] R. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki, "An evaluation of binary xml encoding optimizations for fast stream based xml processing," *ACM Proceedings of the 13th Conference on World Wide Web, WWW2004*, 2004.

[3] D. Fallside and P. Walmsley, "XML Schema Part 0: Primer Second Edition," Online: `http://www.w3.org/TR/xmlschema-0/` [Last Accessed: 21/07/2006], 28 October 2004.

[4] SAX, "Simple API for XML (SAX)," Online: `http://www.saxproject.org/` [Last Accessed: 21/07/2006].

[5] W3C, "Document Object Model (DOM) Technical Reports," Online: `http://www.w3.org/DOM/DOMTR` [Last Accessed: 21/07/2006], 2005.

[6] M. Girardot and N. Sundaresan, "Millau: and encoding format for efficient representation and exchange of XML over the web," *9th conference of the W3C*, 1999.

[7] Information Technology, "Multimedia content description interface - part 1: Systems," *ISO/IEC 15938-1:2001*, 2001.

[8] Information technology, "MPEG B – Part 1: Binary MPEG format for XML," *ISO/IEC 23001-1:2005*, 2005.

[9] "XML.com," Online: `http://www.xml.com/` [Last Accessed: 21/07/2006].

[10] J. Cheney, "Compressing XML with Multiplexed Hierarchical PPM Models," *Proceedings of the 2001 IEEE Data Compression Conference*, pp. 163 – 172, 2001.

[11] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (mime) part one: Format of internet message bodies," *Network Working Group, Request For Comments (RFC) - 2045, Online: `http://www.ietf.org/rfc/rfc2045.txt` [Last Accessed: 21/07/2006]*, November 1996.

[12] A. Møller and M. Schwartzbach, *An Introduction to XML and Web Technologies*, Addison-Wesley, January 2006, ISBN: 0321269667.

[13] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML Schema Part 1: Structures Second Edition," Online: `http://www.w3.org/TR/xmlschema-1/` [Last Accessed: 21/07/2006], 28 October 2004.

[14] P. Biron and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition," OnlineL `http://www.w3.org/TR/xmlschema-2/` [Last Accessed: 21/07/2006], 28 October 2004.

[15] J. Clark and M. Makoto, "RELAX NG Specification," Online: `http://relaxng.org/spec-20011203.html` [Last Accessed: 01/09/2006], 3 December 2001.

[16] Anders Møller, "Document Structure Description 2.0," Online: `http://www.brics.dk/DSD/dsd2.html` [Last Accessed: 09/08/2006], December 2002.

[17] D. Gulbransen, *Using XML*, Indianapolis, Ind. 2002.

[18] T. Bray, D. Hollander, and A. Layman, "Namespaces in XML 1.1, W3C Recommendation," Online: `http://www.w3.org/TR/xml-names11` [Last Accessed: 21/07/2006], 4 Feburary 2004.

[19] E. Harold, *Processing XML with Java : a guide to SAX, DOM, JDOM, JAXP, and TrAX*, Addison-Wesley, 2003.

[20] XMLPULL, "XML Pull Parsing Common API," Online: `http://www.xmlpull.org` [Last Accessed: 21/07/2006].

[21] A. Slominski, "MXP1: Xml Pull Parser 3rd Edition (XPP3)," Online: `http://www.extreme.indiana.edu/xgws/xsoap/xpp/mxp1/` [Last Accessed: 09/09/2006], May 2005.

[22] "kXML2," Online: `http://kobjects.org/` [Last Accessed: 21/07/2006], 2003.

[23] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, and et. al., "XML Path Language (XPath) Version 2.0, W3C Candidate Recommendation," Online: `http://www.w3.org/TR/xpath20` [Last Accessed: 21/07/2006], 3 November 2005.

[24] S. Boag, D. Chamberlin, M. Fernández, and et. al., "XQuery 1.0: An XML Query Language," Online: `http://www.w3.org/TR/xquery/` [Last Accessed: 21/07/2006], 8 June 2006.

[25] J. Simpon, *XPath and XPointer*, O'Reilly, ISBN: 0-596-00291-2 , 2002.

[26] w3schools, "Introduction To XPath," Online: `http://www.w3schools.com/xpath/xpath_intro.asp` [Last Accessed: 09/09/2006].

[27] N. Rump, "MPEG-21 MDS - Frequently Asked Questions (FAQ) - Version 5.0," N5187, ISO/IEC JTC 1/SC 29/WG 11, October 2002.

[28] Information Technology, "Multimedia framework - part 2: Digital item declaration," *ISO/IEC 21000-2*, 2002.

[29] I. Burnett, F. Pereira, R. Van de Walle, and R. Koenen, *The MPEG-21 Book*, Wiley 2006, ISBN: 0-470-01011-8.

[30] J. Bormans and K. Hill, "MPEG-21 overview v.5," *ISO/IEC JTC1/SC29/WG11/N5231*, October 2002.

[31] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Ngoc, "Exchanging Intensional XML Data," *SIGMOD 2003, San Diago, CA.*, pp. 289–300, June 2003.

[32] P. Grosso and D. Veillard, "XML Fragment Interchange, W3C Candidate Recommendation," Online: `http://www.w3.org/TR/xml-fragment/` [Last Accessed: 21/07/2006], 12 Feburary 2001.

[33] E. Wong, A. Chan, and H. Leong, "Semantic-based approach to streaming XML contents using Xstream," *Computer Software and Applications Conference, 2003. COMPSAC. Proceedings. 27th Annual International*, pp. 91 – 96, 2003.

[34] E. Wong, A. Chan, and H. Leong, "Efficient Management of XML Contents over Wireless Environment by Xstream," *Internet Data Management (IDM) ACM, SAC'04*, pp. 1122–1127, 2004.

[35] M. Gudgin, N. Mendelsohn, M. Nottingham, and Herve Ruellan, "Xml-binary optimized packaging, w3c recommendation," Online: `http://www.w3.org/TR/xop10/` [Last Accessed: 21/07/2006], 25 January 2005.

[36] E. Levinson, "The mime multipart/related content-type," *Network Working Group, Request For Comments (RFC) - 2387, Online: `http://www.ietf.org/rfc/rfc2387.txt` [Last Accessed: 27/07/2006]*, August 1998.

[37] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol - http/1.1," *Network Working Group, Request For Comments (RFC) - 2616, Online: `http://www.ietf.org/rfc/rfc2616.txt` [Last Accessed: 21/07/2006]*, June 1993.

[38] J. Postel and J. Reynolds, "File transfer protocol," *Request For Comments (RFC) - 959, [online] `http://www.ietf.org/rfc/rfc959.txt` [Last Accessed: 21/07/2006]*, October 1985.

[39] J. Martinez, "MPEG-7 Overview," Online: `http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm` [Last Accessed: 13/08/2006], October 2004.

[40] S. Boettcher and A. Turling, "XML Fragment Caching for Small Mobile Internet Devices," *Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Erfurt, Germany*, pp. 268–279, October 7-10 2002.

[41] "Glossary of Mobile Terms," Online: `http://mobile.yahoo.com/resource_center/glossary` [Last Accessed: 17/09/2006], 2006.

[42] "Blackberry," Online: `http://www.blackberry.com/` [Last Accessed: 16/09/2006], 2006.

[43] "Nokia," Online: `http://www.nokia.com.au` [Last Accessed: 16/09/2006], 2006.

[44] K. Swanson and J. Judt, "CompreX: XML Compression applied to the Airborne Internet," *NASA ICNS Conference and Workshop*, May 2005.

[45] P. Sandoz, S. Pericas-Geertsen, K. Kawaguchi, and et.al., "Fast web services, sun microsystems technical artical," Online: `http://java.sun.com/developer/technicalArticles/WebServicees/fastWS/` [Last Accessed: 21/07/2006], May 26, 2004.

[46] H. Liefke and D. Suciu, "XMill: An Efficient Compressor for XML Data," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2000, pp. 153–164.

[47] D. Sosnoski, "Improve xml transport performance, part 1: Bandwidth versus processing trade-offs in transporting xml documents," `http://www-106.ibm.com/developerworks/library/x-trans1.html`, 2004.

[48] M. Cannataro, G. Carelli, A. Puglise, and D. Sacca, "Semantic Lossy Compression of XML Data," *International Workshop on Knowledge Representation Meets Databases, in conjunction with International Conference on Very Large Data Bases*, 2001.

[49] M. Cannataro and A. Puglise, "An Architecture for Compressing and Synthesizing XML Documents," *1st Int. Workshop on Document Compression and Synthesis in Adaptive Hypermedia Systems (DoCS 01), in conjunction with the 2nd International Conference on Adaptive Hypermedia and Adaptive Web Systems (AH 2002)*, May 28 2002.

[50] M. Cannataro and A. Pugliese, "An Architecture for Compressing and Synthesizing XML Documents," *1st Int. Workshop on Document Compression and Synthesis in Adaptive Hypermedia Systems (DoCS' 01), May 28, 2002 - Malaga (Spain), in conjunction with the 2nd International Conference on Adaptive Hypermedia and Adaptive Web Systems (AH 2002).*, May 2002.

[51] J. Boyer, "Canonical XML, Version 1.0, W3C Recommendation," Online: `http://www.w3.org/TR/xml-c14n` [Last Accessed: 21/07/2006], 15 March 2001.

[52] M. Cokus and D. Winkowski, "XML Sizing and Compression Study For Military Wireless Data," *XML Conference and Exposition 2002, Baltimore convention center, Blatimore, MD USA*, December 8-13 2002.

[53] H. Wang, J. Li, J. Luo, and Z. He, "XCpasq: Compression of XML Document with XPath Query Support," *Proceedings of the International Conference on Information Technology: Coding and Computing ITCC'04*, 2004.

[54] J. Min, M. Park, and C. Chung, "XPRESS: A Queriable Compression for XML Data," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 122 – 133, 2003.

[55] P. Buneman, M. Grohe, and C. Koch, "Path Queries on Compressed XML," *Proceedings of the 29th VLDB Conference, Berlin, Germany*, 2003.

[56] Winzip Computing Inc, "WinZip," Online: `http://www.winzip.com` [Last Accessed: 21/07/2006], 2004.

[57] P. Deutsch, "Gzip file format specification version 4.3," *RFC-1952, Online: `http://www.ietf.org/rfc/rfc1952.txt` [Last Accessed: 21/07/2006]*, May 1996.

[58] P. Tolani and J. Haritsa, "XGRIND: A Query-friendly XML Compressor," *In Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, 2002.

[59] D.A. Huffman, "A method for the construction of minimum redundancy codes," *in Proceedings of the Institute of Radio Engineers (IRE)*, vol. 40, pp. 1098 – 1101, September 1952.

[60] Introduction to Data Compression, *K. Sayood*, Boston : Elsevier 2006.

[61] A. Lempel and J. Ziv, "A universal algorithm for sequential data compression," *IEEE Transaction on Information Theory*, pp. 337–343, May 1997.

[62] B. Martin and B. Jano, "WAP Binary XML Content Format, W3C Recommendation," [Online:] `http://www.w3.org/TR/wbxml/` [Last Accessed: 21/07/2006], 1999.

[63] "Wireless Application Protocol (WAP) Downloads," [Online:] http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html [Last Accessed: 20/07/2006].

[64] M. Girardot and N. Sundaresan, "Efficient Representation and streaming of XML content over the Internet medium," *Multimedia and Expo, 2000 ICME 2000. 2000 IEEE International Conference on*, vol. 1, pp. 67 – 70, 30 July - 2 Aug 2000.

[65] N. Sundaresan and R. Moussa, "Algorithms and programming models for efficient representations of XML for internet applications," *In Proc. of the Tenth Internatinal world Wide Web Conference.*, 2001.

[66] J. Cleary and I. Witten, "Data Compression using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications*, pp. 396 – 402, 1984.

[67] P. Deutsch and J. Gailly, "Zlib compressed data format specification version 3.3, rfc 1950," Online: http://www.ietf.org/rfc/rfc1950.txt [Last Accessed: 21/07/2006], May 1996.

[68] U. Niedermeier, J. Heuer, A. Hutter, W Stechele, and A. Kaup, "An MPEG-7 Tool For Compression and Streaming of XML Data," *Multimedia and Expo, 2002 ICME Proceedings. 2002 IEEE International Conference on*, vol. 1, pp. 521–524, 2002.

[69] A. Arion, A. Bonifati, and G. Costa et. al., "XQueC: Pushing Queries to Compressed XML Data," *29th International Conference on Very Large Data Bases (VLDB)*, September 2003.

[70] "Wikipedia," Online: http://en.wikipedia.org/wiki/Main_Page [Last Accessed: 21/07/2006].

[71] "Concurrent versions system," Online: http://www.nongnu.org/cvs/ [Last Accessed: 21/07/2006].

[72] OASIS, "Open Document Format for Office Applications (OpenDocument)," Online: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office [Last Accessed: 21/07/2006], 2006.

[73] Microsoft, "Office 2003 XML Reference Schemas," `http://www.microsoft.com/office/xml/default.mspx` [Last Accessed: 21/07/2006], 2005.

[74] S. Helmer, C. Kanne, and G. Moerkotte, "Evaluating Lock-based Protocols for Cooperation on XML Documents," *SIGMOD Record*, vol. 33, no. 1, pp. 58–63, 2004.

[75] A. Gummadi, J. Yoon, B. Shah, and V. Raghavan, "A Bitmap-Based Access Control for Restricted Views of XML Documents," *ACM Workshop on XML Security*, pp. 60–68, October 2003.

[76] C. Lim, S. Park, and S. Son, "Access Control of XML Documents Considering Update Operations," *ACM Workshop on XML Security*, pp. 49–59, October 2003.

[77] Information Technology, "Mpeg system technologies - part 1: Binary mpeg format for xml," *ISO/IEC 23001-1*, 2006.

[78] Requirements, "Draft Requirements on XML Fragment Requests, ISO/IEC JTC1/SC29/WG11 N7067," April 2005.

[79] ICT, Intelligent Compression Technologies, "ICT's XML-Xpress (Whitepaper)," Online: `http://www.ictcompress.com/products_xmlxpress.html` [Last Accessed: 21/07/2006], 2000.

[80] D. Sosnoski, "Xbis xml information set encoding," [Online:] `http://xbis.sourceforge.net` [Last Accessed: 21/07/2006], 2004.

[81] "Bzip2," Online: `http://sources.redhat.com/bzip2` [Last Accessed: 21/07/2006].

[82] D. Mertz, "Compression and Streaming of XML Documents - White Paper on the Entropy of Documents," Online: `http://gnosis.cx/publish/programming/xml_compression.html` [Last Accessed: 20/07/06], 2004.

[83] S. Pericas-Geertsen, "Binary interchange of xml infosets," *The W3C Workshop on Binary Interchange of XML Information Item Sets*, September 2003.

Appendix A

Software Implementation

A.1 Introduction

This Appendix documents the software developed during the course of this thesis in order to generate and validate results.

The following freely available software was used in this thesis:

- Apache Xerces-J 2.6.2;
- JAVA JDK 1.5;
- Jaxen 1.0;
- Apache Xalan 2.6.0;
- JFreeChart 1.0.3; and
- JoSQL-0.9.jar.

The following libraries and applications were created:

- Binarisation library;
- SDOM library;
- RXEP server and client;

- RXPP Server and client; and
- Collaborative Schema server and client.

A.2 Implementation Details

A.2.1 SDOM Library

The SDOM library is written in JAVA responsible for converting the XML instance into SDOM and vice-versa. The SDOM library utilises Apache Xerces Post Schema Validation Infoset (PSVI). PSVI is used to access information about XML Schemas after all the schemas have been loaded and all references in the schema document have been resolved.

When the SDOM is parsed into memory, each node has is attached to the relevant schema information (such as minOccurs and maxOccurs) which will be used by the binarisation library. This schema information is also used for updates to the SDOM as there is enough information for validation, without the need to validate the entire XML document.

A.2.2 Binarisation Library

The binarisation library is also written in JAVA and is used to convert SDOM nodes into its equivalent binary form and vice-versa. Each SDOM node has two methods, encode and decode; both of which take a parameter of the Binary Configuration (BC). The BC contains all information of where to write binary, the options used in encoding and decoding, and the mode used for navigation and queries. The MPEG-21 extensions as described in Chapter 5 are also incorporated into the binarisation library. Figure A.1 shows a screenshot of the JAVA application which is used to test the binarisation library.

A.2.3 RXPP and RXEP Server and Client

The RXEP client software, written in JAVA, provides the user with a simple GUI to navigate or query through a remote XML document. All received XML fragments are converted into a JTree for visual representation (a screenshot of the application is shown in Figure A.2), and a listener is added when a node is clicked. As a node is clicked (and has not previously been clicked), the path is extracted from the JTree and an RXEP XPath Locator is created and

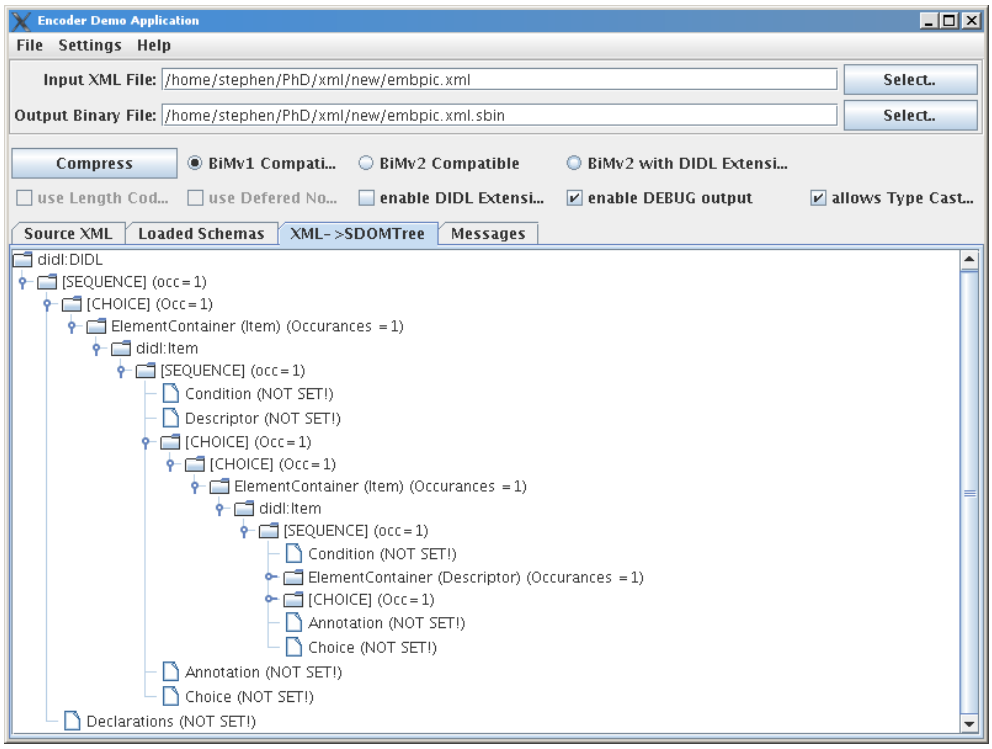

Figure A.1 Screenshot of the test binarisation encoder, showing the SDOM conversion of the input XML document

used within an RXEP Request. After the client receives RXEP Responses, the node location is determined from the RXEP XPath locator, and the JTree is updated to reflect the response action.

The RXEP server listens for incoming RXEP requests. The server decodes RXEP requests, evaluates the query or navigation commands, and generated an RXEP response.

A.2.4 Collaborative Schema Software

The collaborative schema client software, was written in JAVA, and a Figure A.3 shows a screenshot of the client application. The client was written to allow many people (each running a separate version of the client) to create a schema locally and updates sent to the server. Periodically, updates to the main schema are sent to all clients to keep them all synchronised with the main server.

RXEP is used to generate RXEP Add responses to the server when a client has submitted a new descriptor to append to the main schema. When a server receives an update, an RXEP

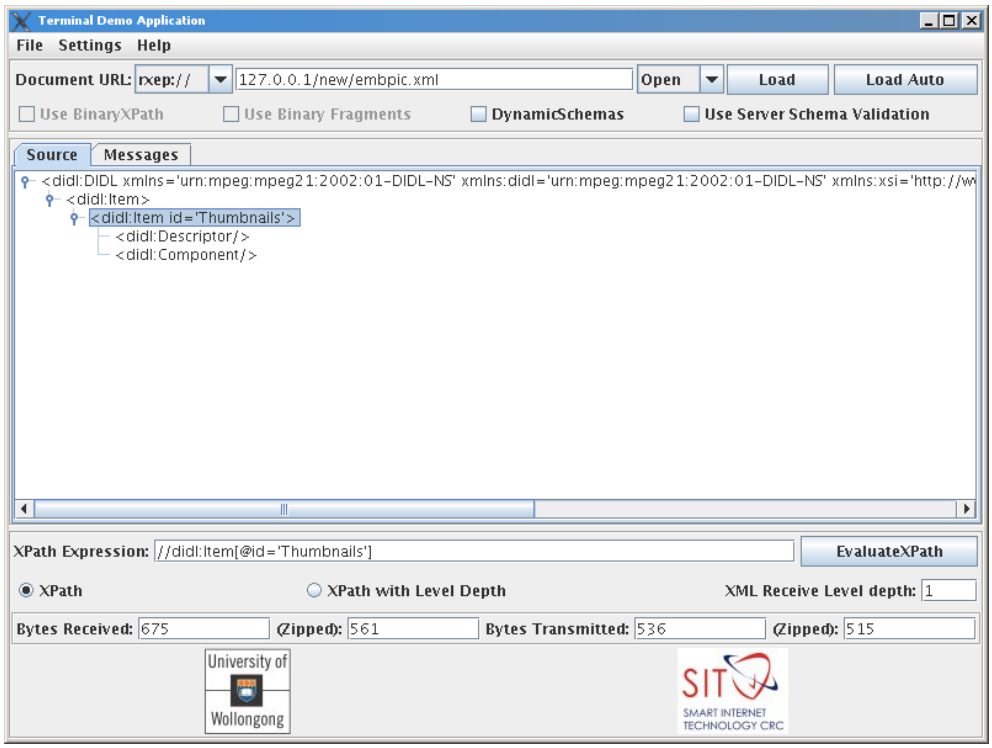

Figure A.2 Screenshot of the client RXEP JAVA application

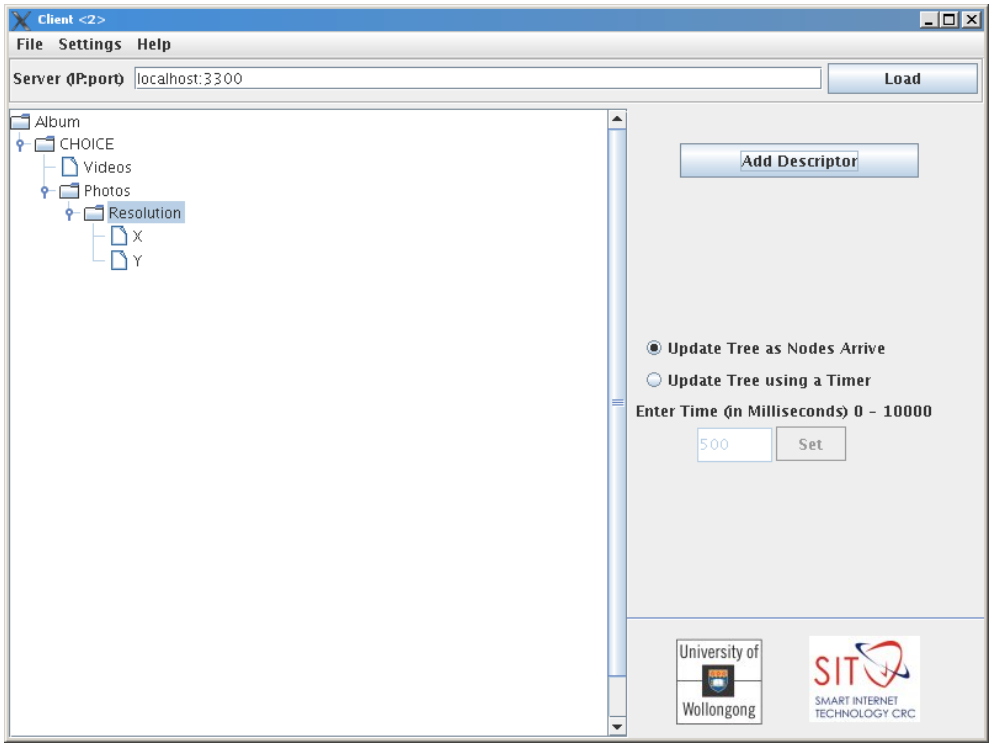

Figure A.3 Screenshot of the collaborative schema client JAVA application

Add response is sent to all connected clients to reflect this change on all client versions of the XML Schema.

The schema server application is also written in JAVA and Figure A.4 shows a screenshot of the application. The server keeps a track of all connected clients, and pushes fragments of XML to the client when the main schema receives changes (using RXEP). The server keeps a log of all transactions.

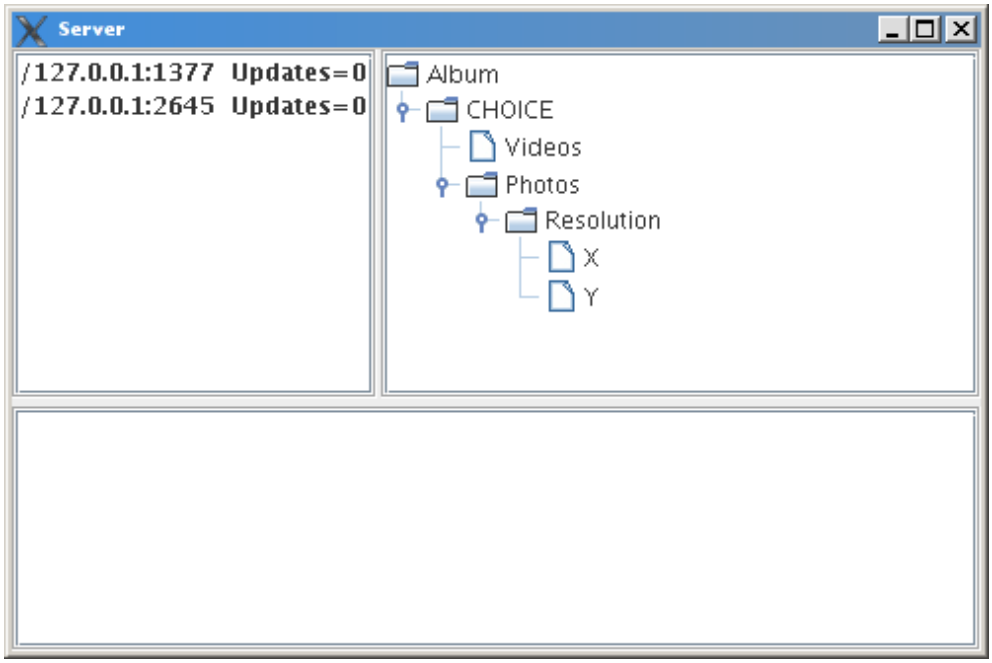

Figure A.4 Screenshot of the collaborative schema server JAVA application

The final application used in the collaborative schema experiment was the log analyser. Figure A.5 shows a screenshot of the log analyser application. The log analyser is run after the experiment, to analyse the log file created by the server software. The log analyser can show how the schema is modified over time, and also provides plots of relevant results.

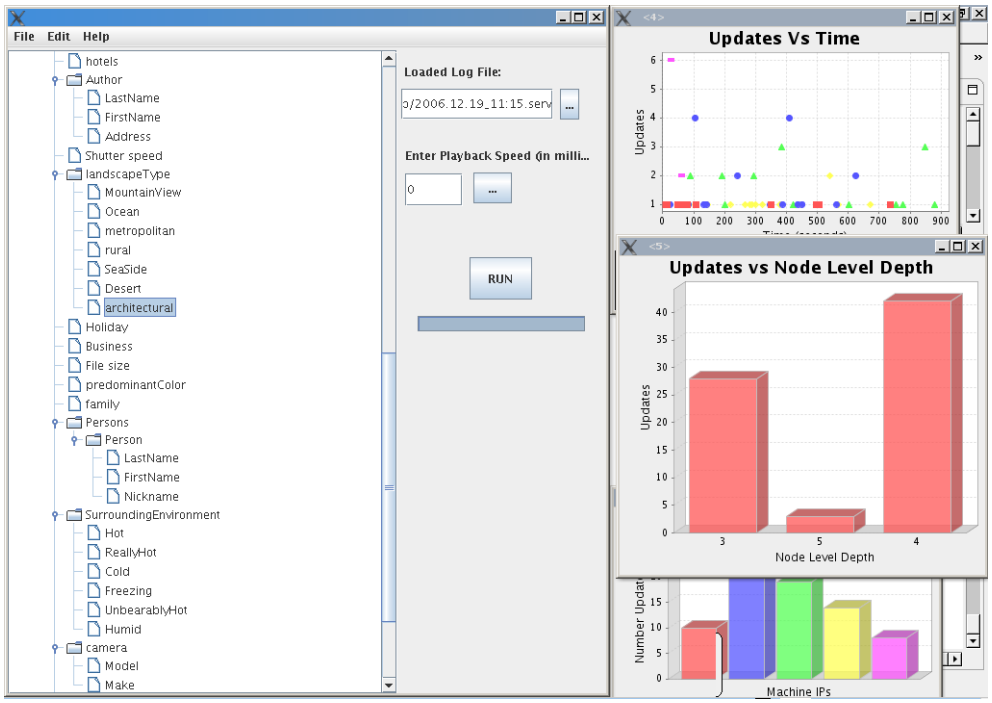

Figure A.5 Screenshot of the collaborative schema log analyser JAVA application

Appendix B

RXEP Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="RXEP">
   <xs:complexType>
     <xs:choice>
       <xs:element name="Request" type="requestType"/>
       <xs:element name="Response" type="responseType"/>
     </xs:choice>
   </xs:complexType>
 </xs:element>

 <xs:complexType name="requestType">
  <xs:sequence>
   <xs:element name="Src" type="srcType" minOccurs="0"/>
   <xs:choice minOccurs="0">
     <xs:element name="Query" type="queryType"/>
     <xs:element name="XMLPull" type="xmlpullType"
                 maxOccurs="unbounded"/>
     <xs:element name="Stream" type="streamType"/>
   </xs:choice>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="srcType">
  <xs:simpleContent>
   <xs:extension base="xs:string">
     <xs:attribute name="OpenMode" type="xs:boolean"
                   use="optional"/>
     </xs:extension>
  </xs:simpleContent>
 </xs:complexType>

 <xs:complexType name="queryType">
   <xs:simpleContent>
     <xs:extension base="xs:string">
       <xs:attribute ref="LevelDepth" use="optional"/>
       <xs:attribute name="navMode" type="xs:boolean"/>
       <xs:attribute name="qLang" type="xs:string"/>
     </xs:extension>
   </xs:simpleContent>
```

```
 </xs:complexType>

 <xs:complexType name="schemaQueryType" >
  <xs:complexContent>
    <xs:extension base="queryType">
      <xs:attribute name="forXML" type="xs:boolean"/>
    </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:attribute name="LevelDepth" type="xs:nonNegativeInteger"
       default="1"/>

 <xs:complexType name="xmlpullType">
   <xs:attribute name = "Command" type="xmlPullCommandType"
                 use="required"/>
   <xs:attribute ref="LevelDepth" use="optional"/>
 </xs:complexType>

 <xs:simpleType name="xmlPullCommandType">
   <xs:restriction base="xs:string">
     <xs:enumeration value="Next"></xs:enumeration>
     <xs:enumeration value="Up"></xs:enumeration>
     <xs:enumeration value="Expand"></xs:enumeration>
     <xs:enumeration value="Back"></xs:enumeration>
   </xs:restriction>
 </xs:simpleType>

 <xs:complexType name="streamType">
   <xs:sequence>
     <xs:any namespace="##any" processContents="lax"/>
   </xs:sequence>
   <xs:attribute name="location" type="xs:string"
                 use="optional"/>
   <xs:attribute name="method" type="xs:string"
                use="optional"/>
 </xs:complexType>

<xs:complexType name="responseType">
 <xs:sequence>
   <xs:element name="RXEPConfig" type="configType"
               minOccurs="0"/>
   <xs:choice minOccurs="1" maxOccurs="unbounded">
     <xs:element name="Add" type="addType"/>
     <xs:element name="Delete" type="deleteType"/>
     <xs:element name="Update" type="updateType"/>
     <xs:element name="Insert" type="insertType"/>
 </xs:choice>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="addType">
   <xs:sequence minOccurs="0">
      <xs:any namespace="##any" processContents="lax"
              minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="location" type="xs:string"/>
</xs:complexType>
```

```
<xs:complexType name="deleteType">
    <xs:attribute name="location" type="xs:string"/>
</xs:complexType>

<xs:complexType name="updateType">
  <xs:sequence minOccurs="0">
    <xs:any namespace="##any" processContents="lax"
            minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="location" type="xs:string"/>
</xs:complexType>

<xs:complexType name="insertType">
  <xs:sequence minOccurs="0">
    <xs:any namespace="##any" processContents="lax"
            minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="location" type="xs:string"/>
  <xs:attribute name="insertBefore"
                type="xs:boolean" use="optional"/>
</xs:complexType>

<xs:complexType name="configType">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="uri">
      <xs:complexType>
        <xs:attribute name="NameSpace" type="xs:string"/>
        <xs:attribute name="Prefix" type="xs:string"/>
        <xs:attribute name="xsdLocation" type="xs:string"/>
      </xs:complexType>
    </xs:element>
   </xs:choice>
 </xs:complexType>
</xs:schema>
```

Appendix C

Thesis Files

The files used throughout this thesis can be downloaded from `http://www.whisper.elec.uow.edu.au/people/sdavis/thesis.tar.gz` which contains the following:

1. XML Documents used in experiments;
2. Photos used for RXEP Schema experiment; and
3. MPEG-21, MPEG-7 and RXEP XML Schemas required for successful validation of XML documents used throughout this thesis.