libpackedobjects tutorial

libpackedobjects - an XML schema-aware compression library

Table of Contents

1 Introduction

1.1 What is libpackedobjects?

libpackedobjects is a C library which can be used to efficiently compress an XML DOM by using the information provided by a corresponding XML Schema. The level of compression achieved is very similar to EXI but unlike EXI, libpackedobjects is designed to be light-weight and simple to implement. Therefore libpackedobjects is suited to embedded systems and mobile devices. The tool is designed for writing network protocols which strive to minimise the amount of data communicated. In addition to compression all data is validated by the schema during the encode and decode process.

libpackedobjects is based on libxml2 and therefore should run on any system that libxml2 runs on.

1.2 Key features

- Very efficient encoding size
- Light-weight and fast
- Validates XML data on encode and decode
- Good choice of data types including the ability to apply range and size constraints
- Fully dynamic including the ability to change the protocol at runtime
- Simple API with two main function calls
- Highly portable - designed for embedded and mobile devices
- Simple subset of XML Schema required to create protocols

1.3 Limitations

libpackedobjects is not a general purpose document compression tool. It is intended to be used in an application that generate XML that you wish to communicate over a network. As such it provides a simple DOM-based API for encoding and decoding structured data. The compression technique used is based on applying knowledge of the data types specified in a schema to provide better performance over statistical compression techniques. Therefore, you must write a valid schema for your data. The style of schema required is based on a small subset of XML Schema. This schema serves the purpose of formalising the network protocol and provides validation. Thus we think it is a good thing!

2 Installation

2.1 Installing libpackedobjects

To install from the latest source:

```
git clone git://gitorious.org/libpackedobjects/libpackedobjects.git
cd libpackedobjects
autoreconf -i
./configure
make
make check
sudo make install
```

2.2 Further reading

3 Getting started

3.1 Quick start

After compiling and running 'make check' you should find a binary called 'packedobjects' in your src directory. This is command-line tool built with libpackedobjects which you can use to test out encoding and decoding:

```
$ ./packedobjects --help
usage: packedobjects --schema <file> --in <file> --out <file>
```

To encode run:

```
$ ./packedobjects --schema foo.xsd --in foo.xml --out foo.po
```

To decode run:

```
$ ./packedobjects --schema foo.xsd --in foo.po --out foo.new.xml
```

If you want to examine the performance of the tool you can use the `--loop` command-line flag. This will loop everything including opening and closing files but will only run the initialisation function one time to mirror intended use.

3.2 API basics

There are only 4 main function calls which are made available by adding `#include <packedobjects/packedobjects.h>` to your code.

```
packedobjectsContext *init_packedobjects(const char *schema_file);

char *packedobjects_encode(packedobjectsContext *pc, xmlDocPtr doc);

xmlDocPtr packedobjects_decode(packedobjectsContext *pc, char *pdu);

void free_packedobjects(packedobjectsContext *poCtxPtr);
```

You first must initialise the library using your XML Schema. Typical use would be one called to init_packedobjects at startup and then multiple calls to encode/decode based on your protocol. The interface to the packedobjects_encode function requires a libxml2 doc type. The packedobjects_decode function returns a libxml2 doc type.

If during runtime your schema changed you must call the init function again with the new file. The library is designed to do preprocessing of the schema during the init function which then allows efficient encoding and decoding plus validation to take place. Therefore, do not call init_packedobjects more than once if you do not plan on supporting dynamically changing protocols at runtime.

To build an application with the software you must link with the library. Using autoconf you can add `PKG_CHECK_MODULES([LIBPACKEDOBJECTS], [libpackedobjects])` to your configure.ac file and then use the variables `$(LIBPACKEDOBJECTS_CFLAGS)` and `$(LIBPACKEDOBJECTS_LIBS)` in your Makefile.am file.

3.3 Writing a schema

libpackedobjects uses a subset of XML Schema. This provides a focus similar to the concept of a Domain Specific Language. You are provided with suitable data types to be able to create functional network protocols. We will use the canonical "Hello World" example. We first write a schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:element name="foo" type="string"/>
</xs:schema>
```

We then create the corresponding data:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>Hello World!</foo>
```

This simple example only defines one data type which happens to be a string. However it provides a template for the basic structure of all schemas. You first must include the libpackedobjects data types and then specify a root element. In this case the root element is called 'foo'.

The easiest way to learn how to write a schema is by looking at some of the examples available here.

3.4 Schema styles: flat vs nested

You have the choice of writing your schema in a flat style:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:complexType name="foobar">
    <xs:sequence>
      <xs:element name="bar" type="integer"/>
      <xs:element name="baz" type="integer"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="foo" type="foobar"/>
</xs:schema>
```

You will see the use of a 'type' attribute to refer to the previously defined sequence. Or you can write your schema in a nested style:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:element name="foo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bar" type="integer"/>
        <xs:element name="baz" type="integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Either schema could be used with the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<foo>
  <bar>1</bar>
  <baz>2</baz>
</foo>
```

3.4.1 Tips for writing an efficient schema

If your schema is dominated by strings you may find that you are overusing this type. Are you sure the data could not be represented by another type? For example, the data you are modelling may be a string but only varies between a few different values. In such cases it is much more efficient to use an enumerated data type. For example we could represent the following data as a single string or we could use an enumeration as follows:

```
<xs:element name="Name">
  <xs:simpleType>
    <xs:restriction base="enumerated">
      <xs:enumeration value="Binary 1" />
      <xs:enumeration value="Binary 2" />
      <xs:enumeration value="Binary 3" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

When we encode `<Name>Binary 2</Name>` we only need a couple of bits instead of a number of bytes.

Something else you should always consider doing is restricting the length of strings and limiting the number of times a sequence might repeat. This will avoid the use of length values which reduces the number of bits but more importantly will provide some safety on the values returned by a decoder. Leaving these types unbounded is asking for a trouble on an embedded system!

4 Data types

Writing a packedobjects schema involves using a set of predefined data types. These data types provide convenient syntax for representing information such as an IP address or currency etc. Please note, the list of valid simple types is likely to change.

4.1 Simple types

An up-to-date list of simple data types can be found here. All string types and integers can have additional constraints added to them. This not only controls the size of the encoded data but can act as an extra form of validation.

4.1.1 String constraints

All string types can be constrained with size constraints: `xs:minLength`, `xs:maxLength` and `xs:length`. For example you might have:

```
<xs:element name="givenName">
  <xs:simpleType>
    <xs:restriction base="string">
      <xs:minLength value="1" />
      <xs:maxLength value="64" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

This will restrict givenName to be between 1 and 64 characters in length (inclusive). Or you might use something like:

```
<xs:element name="initial">
  <xs:simpleType>
    <xs:restriction base="string">
      <xs:length value="1" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

This only allows a single character to be present.

4.1.2 Integer constraints

Integers can be constrained with range constrants: `xs:minInclusive` and `xs:maxInclusive`. So you could use both constraints:

```
<xs:element name="foo">
  <xs:simpleType>
    <xs:restriction base="integer">
      <xs:minInclusive value="0" />
      <xs:maxInclusive value="100" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Or you could just one of the constraints:

```
<xs:element name="foo">
  <xs:simpleType>
    <xs:restriction base="integer">
      <xs:maxInclusive value="100" />
    </xs:restriction>
```

```
    </xs:simpleType>
  </xs:element>
```

In this case negative integers would be valid.

4.2 Complex types

There are only two complex data types and these allow you to represent sequences and choices.

4.2.1 Sequence

A sequence is a way of logically grouping data within its own unique namespace. Every item in the sequence must be present and be in the correct order. So given the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:complexType name="foobar">
    <xs:sequence>
      <xs:element name="boz" type="integer"/>
      <xs:element name="baz" type="integer"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="foo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bar" type="foobar"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

We could supply the following data:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>
    <boz>1</boz>
    <baz>2</baz>
  </bar>
</foo>
```

4.2.2 Sequence with optionality

Sometimes we don't want to include everything in the sequence. In this case you can use a variant of the sequence type to express this optionality. You just need to add the attribute `minOccurs="0"` to the items which are optional. Using the previous example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:complexType name="foobar">
    <xs:sequence>
      <xs:element name="boz" type="integer"/>
      <xs:element name="baz" type="integer" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="foo">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element name="bar" type="foobar"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

So in this case we saying you must include a boz item but baz items are optional. Therefore the following is now valid:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>
    <boz>1</boz>
  </bar>
</foo>
```

4.2.3 Sequences with data that may repeat

You will often work with data that has a repeating structure. You can handle this with a special form of sequence by adding the `maxOccurs` attribute to element that may repeat. Using the previous example we could do the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:complexType name="foobar">
    <xs:sequence>
      <xs:element name="boz" type="integer"/>
      <xs:element name="baz" type="integer" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="foo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bar" type="foobar" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In the above we have said that `bar` can repeat and there is no limit on how many times this might happen. Therefore, the following is valid:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>
    <boz>1</boz>
    <baz>2</baz>
  </bar>
  <bar>
    <boz>1</boz>
  </bar>
</foo>
```

It is a good idea to specify an integer instead of “unbounded” when you can. We can also supply a `minOccurs` attribute which is useful, for example, when you want to say that there might be no items.

4.2.4 Choice

The ability to structure data selectively is very powerful. You can easily implement simple network protocols where there is a choice between the different application messages. You could, for example, have a state machine which maps to different nested levels of choices. As a simple example we could implement a ping type protocol using:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="http://zedstar.org/xml/schema/packedobjectsDataTypes.xsd"/>
  <xs:element name="pingpong">
    <xs:complexType>
      <xs:choice>
        <xs:element name="ping" type="null" />
        <xs:element name="pong" type="null" />
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Our client application might supply:

```
<?xml version="1.0" encoding="UTF-8"?>
<pingpong>
  <ping/>
</pingpong
```

And our server application might respond with:

```
<?xml version="1.0" encoding="UTF-8"?>
<pingpong>
  <pong/>
</pingpong
```

But both applications use the same network protocol or schema as they must have knowledge of both messages.

Index