U.S. DEPARTMENT OF
ENERGY

PNNL-22585

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Data Serialization Protocol

DES-0002

Revision 2

Charlie Hubbard

August 2012

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

Data Serialization Protocol

Charlie Hubbard

DES-0002
Revision 2
August 2012

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

There are many situations in which it is useful to encode a formal C/C++ data structure like a `struct` or a `class`, or some other set of related data into a text-based stream (a process called *serialization*). This is useful, for instance, when writing structure data to disk or transmitting it to another computer that uses a different hardware architecture.

The need for data serialization occurs again and again in the code base, almost always in connection with passing messages between processes or between computers. A simple text-based protocol for serializing/deserializing data into/out of strings is described here. We will first examine the format of the protocol itself, then we'll look at the `MessageParser` class. This class implements a simple parser for protocol compliant strings.

2 Protocol Format

This section describes the protocol format in detail. Because this protocol has its origin in solving the problem of inter-process and inter-computer communication, the verbage in the source code speaks a lot about *messaging*, even though technically this protocol is much more general than that. Nevertheless, to be consistent with the source code, we'll continue to use the same message-oriented nomenclature throughout this document.

2.1 Format Definition

Protocol compliant text strings have a very simple format that is most easily described with a few simple rules:

1. All *messages* take the form of an opening parenthesis followed by an *element list* followed by a closing parenthesis.
2. An *element list* is either a single *element*, or it is two or more *elements* separated by commas.
3. *Elements* are either *simple elements* (string literals consisting of any printable ASCII text character or the zero-byte empty string), or they are *compound elements*. Compound elements are complete messages in their own right. They just happen to be embedded (nested) inside another message.

2.2 Format Examples

Some examples will make this more clear. The simplest message is the empty message. It consists of just an opening parenthesis immediately followed by a closing parenthesis.

```
()
```

A slightly more complicated and more useful example is a message containing simple elements. In the following example, the message contains a list of names.

```
(Tom,Dick,Harry,Sally,Fred,Wally,Barney)
```

Quite frequently, messages contain some combination of simple elements and compound elements. As an example, suppose that a particular server maintains a set of temperature sensors. Each sensor has a name, and it reports back a temperature as well as the units the temperature is reported in. Assume that the server supports a total of three sensors called *temp01*, *temp02* and *temp03*. If the server is asked to report the values back for all temperature sensors that it maintains, the response message might look something like the following.

```
(getalltemps,(temp01,27.3,C),(temp02,38.4,C),(temp03,110.1,F))
```

In this example, the first (and only) simple element is the string *getalltemps*, which we'll pretend is the server simply reporting back the name of the command to which it is responding. What follows are three compound elements (or sub-messages) that provide the name, temperature and units for each of the server's three temperature sensors.

2.3 Special Characters and the Escape Character

The characters that are allowed to make up a simple element consist of all the printable ASCII characters. This presents a bit of a problem because there are three printable ASCII characters — the opening parenthesis, the closing parenthesis, and the comma — that serve as message delineators and element separators. To use any of these characters, the characters must be escaped with a special character to indicate that, in this particular instance, the character should be treated literally and not as a delineator or separator. We use the tilde for this purpose.

To see the escape character in action, consider this example. A server is reporting back the following error message.

```
Unable to read data from sensor 'temp07' (unknown sensor name)
```

Clearly, in this case, the parentheses around "unknown sensor name" are actually part of the error message and not message formatting characters. In order to successfully package this message, the tilde escape character is required as shown in the following example.

```
(Unable to read data from sensor 'temp07' ~(unknown sensor name~))
```

Because the tilde character is itself a printable ASCII character that can either be used literally or used as a format escape character, it too can be escaped. If you want to use a tilde as a literal in a message, simply precede it with another tilde. The first tilde will then be treated as an escape character indicating that the second tilde should be treated literally. For example, the following string

```
The tilde character (~) is used as an escape character
```

would be encoded as follows.

```
(The tilde character ~(~~~) is used as an escape character)
```

2.4 Arbitrary Nesting

In the examples above, we have seen that messages can contain *simple elements*, *compound elements* or a combination of both. As mentioned before, compound elements are themselves complete format messages that just happen to be embedded inside other messages. In the *getalltemps* example above, we saw a message that contained three embedded sub-messages. Although the embedded messages in that example did not themselves contain embedded messages, they could have. In fact, nesting messages inside messages inside messages can be done to any arbitrary depth, and this property makes our chosen message format extremely suitable to encoding data structures which themselves contain embedded data structures, which may themselves also contain embedded data structures (and so on). More will be said about this in section 4.

3 Rationale

Any number of encodings could be used to stream and store data structures and other related collections of data. Standard encoding formats already exist (raw binary, XML, ASN.1, etc.), so why not use one of those rather than re-inventing the wheel? Here are some of the reasons why the encoding protocol described in this document is particularly suitable for our specific code base and our intended applications (inter-process and inter-computer communication and, to a lesser extent, data storage).

- Our chosen format is text-based rather than binary. That means it can be used across different computer architectures without having to worry about issues of word size, structure padding or byte ordering. This is an advantage for both communication and data storage applications.

- One disadvantage of text-based formats is they are often not as space-efficient as their binary counterparts. Certainly some text-based formats, those based on XML in particular, are extremely space inefficient. Our format, on the other hand, has low encoding overhead, making it reasonably space efficient. Depending on the type of data to be encoded, it can even be *more* efficient than raw binary formats.
- Simplicity is key. Our format is very easy to construct and very easy to parse, certainly when compared to something like XML or the abomination that is ASN.1.
- Our format is reasonably human-readable. Messages can easily be written by hand, on the fly (typed into a command terminal for example), and can be relatively easily interpreted through casual visual inspection (extremely difficult with binary protocols). This is very useful during development, debugging and testing.

4 The `MessageParser` Class

Building protocol-conformant strings is extremely simple. In the code base, this is typically done one of two ways. Either the messages are just built "by hand" or, in the case of data structures that need to be encoded, a special `struct` or `class` method called `Serialize()` is written that returns a string containing a protocol encoded version of the contents of the structure. (NOTE: structures that provide `Serialize()` methods also always offer `Deserialize()` counterparts that perform the opposite action.)

The more difficult task is to parse the strings back into their constituent parts. This is done with the `MessageParser` class, defined in *messageParser.h* and implemented in *messageParser.cpp*. It will be helpful to the reader to at least skim through the *messageParser.h* file before continuing on with this section.

A slightly abbreviated version of the `MessageParser` class definition is shown below, with detailed descriptions of its various methods following.

```
class MessageParser {
public:
   int  NumFields() const;
   void Parse(string const &s);
   void Dump();
   const char* operator[](int index) const;
   const string & operator()(int index) const;
   static string Escape(const string &s);

private:
   ...
   FieldVector fieldVector;
};
```

4.1 `Parse()`

The `Parse()` method is the heart of the `MessageParser` class, so it will be discussed first. It takes as its one and only input, a reference to an STL string containing the message to be parsed. Internally, the string is processed character by character using a simple four-state state machine (see figure 1).

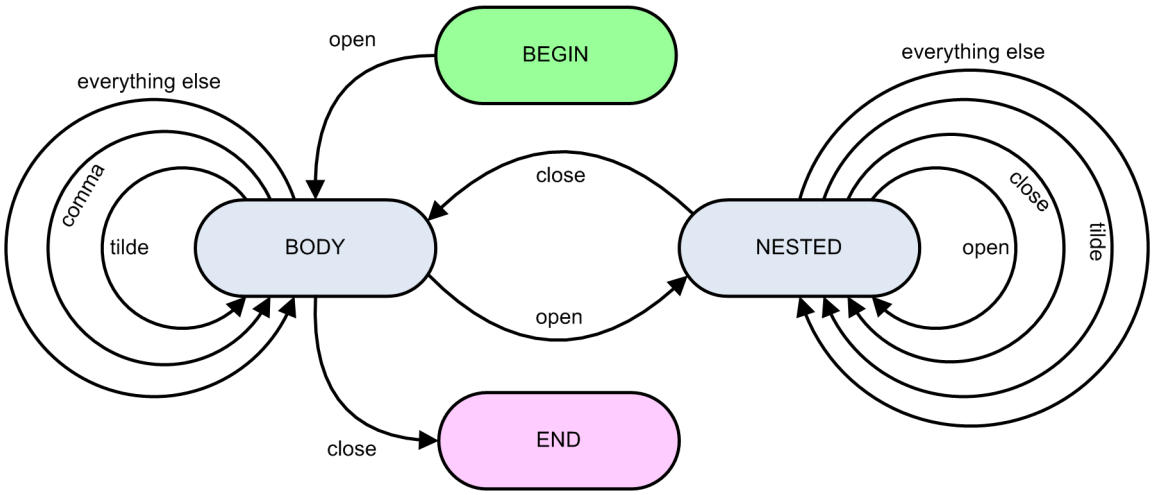

Figure 1: The `MessageParser::Parse()` state machine

As characters are consumed by the state machine, elements (either simple or compound) are recognized and stripped off, in order, from left to right, and pushed to the end of the class' internal `fieldVector` (an STL vector of STL strings). Other class methods, to be described shortly, allow the user to access the individual elements. A detailed description of the various state transitions is given in table 1.

It is important to note that one call to the `Parse()` method parses down only a single nesting level. So, if `Parse()` was called on the message from our earlier *getalltemps* example, four elements would be returned — The simple element *getalltemps*, and three compound elements containing the name, temperature and reporting units for each of the three temperature sensors. To get at the individual values in the compound elements, `Parse()` would need to be called again on each of those compound elements.

Table 1: `MessageParser::Parse()` state machine transitions

From	To	Char	Description
BEGIN	BODY	(	Consume the (character.
BODY	NESTED	(	Add the (character to the end of the current element. Increment the nesting level.
BODY	BODY	,	Consume the comma. Push the current element (now complete) onto the end of the vector. Clear the current element.
BODY	BODY	~	Consume the tilde. Add the next character to the end of the current element.
BODY	BODY	everything else	Add the current character to the end of the current element
BODY	END	)	Consume the). Push the current element (now complete) ont the end of the vector.
NESTED	NESTED	(	Add the (to the end of the current element. Increment the nesting level.
NESTED	NESTED	)	((if nesting level is greater than 1) Add the) to the end of the current element. Decrement the nesting level.
NESTED	NESTED	~	DO NOT consume the tilde! Add it and the next character to the end of the current element. That's because at this point we are parsing a nested sub-message. This message will need to be parsed at least one more time to pull out its individual fields. For that reason, we need to leave the escape character in place. It's only during the top-level (BODY) parse that it gets removed from the stream.
NESTED	NESTED	everything else	Add the current character to the end of the current element.
NESTED	BODY	)	(if the nesting level is equal to 1) Add the) to the end of the current element. Decrement the nesting level.

It should be mentioned that table 1 only shows those transitions that exist for dealing with properly formatted protocol strings. In the actual code (see *messageParser.cpp*), checks are made to ensure, for example, that the string begins with a opening parenthesis, that parenthesis pairs are not mismatched, and that strings terminate with a closing parenthesis at the correct nesting level. Malformed strings thusly detected, cause the parser to throw an exception containing an appropriate error message.

4.2 `Numfields()`

The `NumFields()` method just returns the number of elements (both simple and compound) that were parsed out the last time `Parse()` was called.

4.3 `Dump()`

The `Dump()` method is intended only for debugging and testing. All it does is print a list of all the fields that were parsed the last time `Parse()` was called.

4.4 The `[]` and `()` Operators

Users of the `MessageParser` class access the individual elements parsed out by the `Parse()` method using either the `[]` or `()` operators. Both of these take an index to the element the user is interested in and return that element in response. The difference is, the `()` operator returns the requested element as an STL string, whereas the `[]` operator returns the requested element as a pointer to a traditional C-style character array string. The `[]` operator really only exists as a convenience. There are many instances in the code where a parsed element is being passed along to a function that requires C-style strings. Using `[]` instead of `()` just makes that possible without having to append the `c_str()` call onto the end of the operator. In other words, `parser[2]` is the same as `parser(2).c_str()`.

4.5 `Escape()`

The `Escape()` method is a static method that takes a normal STL string and returns a version of that string in which any special characters (opening and closing parentheses, commas and tildes) are escaped with tilde characters.

This is very useful when arbitrary text messages (like error messages and other event log messages) need to be packed into a protocol message. See section 2.3 for more details.

4.6 Some Words on Parse Efficiency

On the face of it, the parsing scheme described in section 4.1 is not particularly efficient, since, for each additional level of nesting, an additional pass through the characters that make up the compound elements is required. Would a parser that did a single pass on a protocol string and returned a completely parsed tree be better? Actually, no. We happily live with the inefficiency inherent in a single level parse for a few reasons:

- The current parser is dead simple. It took little code to implement it, it is easy to understand and it is easy to maintain. Simplicity and clarity are worth a lot of inef-

ficiency provided the inefficiency isn't causing unacceptable bottlenecks in the code's performance.

- As used in the code base, most protocol encoded strings *do not* contain compound elements anyway. Why devote a lot of coding effort to develop a full tree parser, when few protocol strings would actually benefit from the additional parsing?
- Long use of the encoding protocol described in this document and the parser class described in this section have shown that quite often, a single nesting level parse is exactly what is needed. Compound elements pulled out of a single nesting level parse are often most appropriately parsed and examined elsewhere in the code, perhaps by a different function (for example, handed over to the `Deserialize()` method of an embedded data structure), and don't need to be decoded any further by the code that performed the initial parse.

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov