GENERATION OF SIMPLE, TYPE-SAFE MESSAGES FOR INTER-TASK COMMUNICATIONS

R. Neswold, C. King
FNAL[†], Batavia, IL 60510, U.S.A.

Abstract

We present a development tool that generates source code to marshal and unmarshal messages. The code generator creates modules for differing processor architectures and programming languages.

INTRODUCTION

The Fermilab Control System has dozens of protocols carried by the ACNET transport layer. These protocols are responsible for acquiring real-time data, reporting alarm conditions, reporting changes to the central device database, broadcasting Tevatron clock events, and announcing state changes, among others.

The network representation of these protocols has always been based upon the layout of a C-language structure targeted for a little-endian machine. Given the historical choices of hardware at Fermilab, this has worked well for many years.

As the control system continues to evolve, however, this approach becomes error prone. Modules supporting various processor types must make sure bytes are ordered correctly. Also, with the inclusion of Java to our control system, we lose the ability to incorporate the raw C structures that describe the protocol and instead need to process the data piece by piece.

This methodology can, clearly, be improved.

REQUIREMENTS

The development process for new protocols should meet the following criteria:

- It must remove the tedious and fallible aspects of encoding/decoding messages; programmers should no longer be concerned with endianness and packet layout.
- It should provide message validation. When sending a message, it always generates a well-formed packet. When receiving, it performs sanity checks before accepting and decoding the message.
- The generated code should allow the messages to be manipulated naturally in the target language.
- Support the strongest type-safety available in the target language.
- Message encoding and decoding needs to be fast, since it is competing with access of raw C structures.
- Must generate code for all primary programming languages used at Fermilab (i.e. currently C++, Java and Python).
- The resulting binary message must be self describing thus having the ability to be manually decoded “on the wire”.

INVESTIGATIONS

We investigated several current technologies to see if an adequate solution was already available.

XML (eXtensible Markup Language) was considered, since it has document validators and supports multiple target languages. Unfortunately, XML manipulation libraries aren't known for their speed. Plus, document validation requires us to provide a DTD (Document Type Definition) file to be available – typically accessed via HTTP.

JSON (JavaScript Object Notation)[1] was also investigated. It's available for many programming languages through third-party libraries and is reasonably efficient to parse and generate. Our greatest concern, however, is that, since floating point values are converted to and from ASCII representations, we could lose some least significant digits during the translation.

One of the last projects we researched, Google's Protocol Buffers[2], looked very promising. It meets nearly all of our requirements except where we need to easily decode packets using network diagnostic utilities. This requisite comes from the “Project X Control System Requirements” document. Protocol Buffers encode the message in a binary form that is only decodable by the client. Despite this limitation, this project sparked the idea of using a code generator to create the encoder/decoder (previously we were considering using a dynamic API, similar what DOM libraries do.)

THE COMPILER

To meet our requirements, we created the “protocol compiler”. The compiler is a command line tool that can be invoked by a makefile to convert a protocol source file into a source file of the target language. By convention, the source file uses a `.proto` extension. The generated files have the same base name, but with the appropriate extension used by the target language.

[†]Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

Grammer

The grammar used by the protocol compiler is given in Extended Backus-Naur Form in Figure 1.

```
<protocol> ::= <struct>*, <request-
     message>+, <reply-message>*

<request-message> ::= “request”, <message-
     name>, “{“, <field>*, “}”

<reply-message> ::= “reply”, <message-
     name>, “{“ <field>* “}”

<struct> ::= “struct”, <struct-name>, “{“
     <field>* “}”

<field> ::= [“optional”], <type>, <field-
     name>, [“[]”], “;”

<type> ::= “bool” | “int16” | “int32” |
     “int64” | “double” | “string” |
     “binary” | “struct”, <struct-name>

<message-name> ::= [_a-zA-Z], [_a-zA-Z0-9]*

<struct-name> ::= [_a-zA-Z], [_a-zA-Z0-9]*

<field-name> ::= [_a-zA-Z], [_a-zA-Z0-9]*

<reply-type> ::= “single” | “multiple”,
     <reply-message>

<request-message>+ ::= “->”, <reply-type>+,
     “|”, <replytype>* | “nothing”
```

Figure 1: Protocol Compiler Grammar

Field Types

Messages need a payload to transfer information between processes, so we needed to pick a set of useful data types that clients could exchange. In choosing which subset of data types we should allow, we tried to find a common set available among the languages we eventually want to support. Since most languages don't support unsigned integers, they were left out. Some languages, like Python, only support 64-bit floating point, so there's only one floating point form used.

Based on these constraints, we end up with the set of primitive types consisting of boolean; signed 16-bit, 32-bit, and 64-bit integers; 64-bit floating point; strings; and binaries. Table 1 shows how these types are mapped to two target languages.

Strings are simply a sequence of octets. They get mapped to std::string and String in the C++ and Java mapping, respectively, so they can include NUL characters. A protocol designer may decide that strings contain a simple ASCII string or a UTF-8 encoded value.

A new type can be defined using the `struct` keyword, which closely follows the syntax of a structure in the C language. In the body of the structure, all the predefined primitive types, as well as previously defined structures, may be used.

Fields can be prefixed with the keyword “optional” to specify the field may be left out. How a optional value is expressed depends on the target language's facilities. An optional value that isn't present requires no network traffic. An optional value that's specified uses no more network bandwidth than a required field.

An array of types is denoted by empty square brackets following the field name. Arrays are homogeneous, although they may be arrays of structures.

Table 1: Primitive types mapped to C++ and Java

Type	C++ mapping	Java mapping
bool	bool	boolean
int16	int16_t	short
int32	int32_t	int
int64	int64_t	long
double	double	double
string	string	String
binary	vector<uint8_t>	byte[ ]
struct N	struct N	class N
T[ ]	vector<T>	T[ ]
optional T	auto_ptr<T>	Uses object references

Message Types

There are two message types exchanged between processes: requests and replies. In the protocol file, a message is marked as one of the two using the `request` or `reply` keywords.

When the compiler generates the source files, it uses the target languages facilities to group requests together and group replies together. For example, the C++ and Java generators create a class hierarchy for request objects and a separate hierarchy for reply objects. By tightly binding each message type in their own hierarchy, they become a data type that the compiler can verify and enforce.

Functions that send a message should specify the base class of the message as a parameter. The compiler can then guarantee at compile time that only proper messages are sent. The verification of received messages is done at run-time. The decoding function either returns a fully decoded message or reports an error if the message is badly formed (usually done by throwing an exception.)

Message Flow

Message flow statements follow after the last message definition in a protocol file. It is defined using the “returns” operator “->” as shown is Figure 1. All request messages require a flow statement which details which replies, if any, can be expected. Code generation is not affected by the “returns” statement, so it merely a way of forcing minimal protocol documentation since the protocol file will not compile in its absence.

Encoding

Messages are encoded using a self describing binary tagged format. It only takes a single pass through the data for both encoding and decoding packets.

CONCLUSIONS

The protocol compiler is a useful tool when designing new protocols. We've had the chance to use it recently for new protocols we've created. In each case, we no longer had to deal with byte ordering bugs or field alignment errors. C++ and Java clients were easily able to communicate with each other and we could focus our efforts on the applications themselves, rather than the low-level details.

Although the protocol compiler only generates C++ and Java source files, we plan on adding more target languages. Fermilab has a fairly large community of Python programmers, so Python is a priority and will be added next.

Lastly, although we use the protocol compiler for developing ACNET applications, there is nothing in the protocol compiler that depends on ACNET. We encode and decode messages into a buffer that is sent to ACNET for delivery. The generated classes could just as easily be used to send messages across a TCP socket, or to read and write data in a file.

REFERENCES

[1] Crockford, D. "RFC 4627 – JSON" The Internet Engineering Task Force. The Internet Engineering Task Force. 29 Sep. 2009 <http://www.ietf.org/rfc/rfc4627.txt?number=4627>

[2] "protobuf." Project Hosting on Google Code. Google, Inc. 29 Sep. 2009 <http://code.google.com/p/protobuf/>