Generation of Simple, Type-safe Messages for Inter-task Communications

R. Neswold, C. King
FNAL, Batavia, IL 60510

We present a development tool, the protocol compiler, that generates source code to marshal and unmarshal messages. The protocol 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.

The protocol compiler is a useful tool when designing new protocols and has been used recently for new protocols at Fermiab. 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 current 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.

Even though we use the protocol compiler for developing ACNET applications, there is nothing in the protocol compiler that depends on ACNET libraries or tools. 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.

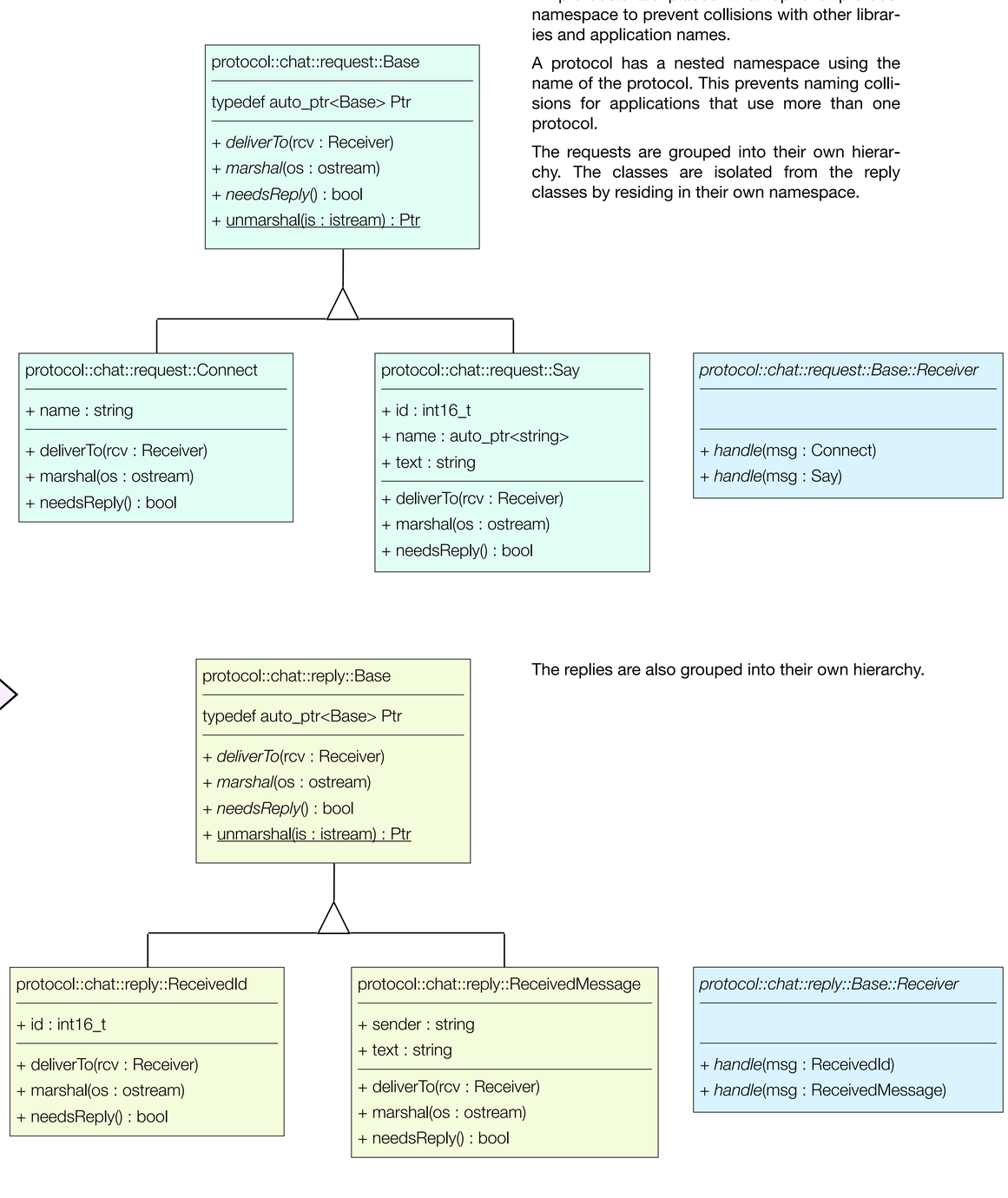
For this example, we'll define and create a protocol for a simple chat server. This example uses some ACNET conventions (i.e. a single request can receive multiple replies.)

A Connect message is sent which contains the name the client wishes to be known as. The server responds with a ReceivedId reply which contains the ID the client uses to send messages. This connection is left open to later receive ReceivedMessage replies.

While this connection is open, the client can make another request using the Say message. If the ID in the Say message is valid, the server will send the text to all listeners when the name field is not specified, or to the specified listener, if given.

```
// ACNET Chat server

request Connect {
    string name;
}

reply ReceivedId {
    int16 id;
}

reply ReceivedMessage {
    string sender;
    string text;
}

Connect -> single ReceivedId |
           multiple ReceivedMessage;

request Say {
    int16 id;
    optional string name;
    string text;
}

Say -> nothing;
```

Compiles to the C++ class hierarchy



All protocols are placed in a top-level protocol namespace to prevent collisions with other libraries and application names.

A protocol has a nested namespace using the name of the protocol. This prevents naming collisions for applications that use more than one protocol.

The requests are grouped into their own hierarchy. The classes are isolated from the reply classes by residing in their own namespace.

The replies are also grouped into their own hierarchy.

This C++ snippet shows how an application can send a Say message using the generates class interfaces. We have a global 'id' variable holding the client's ID (presumably obtained via a previously sent Connect message.)

All fields in the message are public, so it's easy to fill it with the appropriate data. One exception is the 'name' field which, since it's an optional field, uses the C++ auto_ptr<> class to maintain an optional value.

This example uses an ostringstream to act as a memory buffer to receive the encoded message, but it easily could have been a file stream or a socket.

```
#include <Chat.h>

uint16_t id;

…

protocol::Chat::request::Say say;

say.id = id;
say.name.reset(new string("Watson"));
say.text = "Mr. Watson. Come here. I need you.";

ostringstream os;

say.marshall(os);
```

This example decodes an incoming message. A callback class is defined which handles each possible message type (the compiler ensures only messages of the correct type will be handled and will complain if a message is ommitted.)

```
#include <Chat.h>
using namespace protocol::Chat:reply;

class Handler : public Base::Receiver {
    void handle(ReceivedId const&) {…}
    void handle(ReceivedMessage const&) {…}
};

Handler callback;
istringstream is;

// Place encoded message in 'is'
…

Base::Ptr ptr = Base::unmarshal(is);

ptr->deliverTo(callback);
```