The Protocol Compiler

Charlie King,
Rich Neswold

The Old Way

- Use a C struct to define packet layout
  - Occasionally, compiler options were needed to properly pack
- Endianess typically followed VAX/VMS practices
- Client code responsible for message validation

Why Change?

- Tedious and error prone
  - Although C targets can cut-n-paste the structures, each target still needs to verify the packing and endianness of the fields
  - Other targets (i.e. Java) need to process these packets field by field
    - Message validation was never enforced and rarely done adequately
- Current industry standards specify network byte order
- Protocols are hard to modify
- Difficult to debug on the wire

Alternate Technologies

- XML
  - Too much overhead
- JSON (Javascript Object Notation)
  - Loss of numeric precision
- Google Protocol Buffers
  - Didn't meet all our requirements
- ASN.1 (Abstract Syntax Notation)
  - This is what the protocol compiler output is based on (using DER -- Distinguished Encoding Rules)

The Protocol Compiler

- Protocol formally specified in a protocol source file
- Provisions for easily extending the protocol with backward compatibility
- Generates C++ and Java source code (with other languages planned)
  - Binary tagged format efficient to encode and decode
  - Complete message validation guarantees type-safety
  - Message manipulation using native language constructs and host byte order

What it isn't

- CORBA or RMI
  - We don't try to model remote method invocation
  - We don't require ancillary tasks running, like Name Services, ORBs, etc.
  - We're trying very hard to make sure targeted languages won't require linking with support libraries.

Fields

- Types
  - bool
  - int16, int32, int64
  - double
  - string
  - binary
- Groups of types (i.e. structures)
  - Including arrays or nested groups
- Arrays of types
  - Including groups or nested arrays
- Any field can be denoted as optional

Messages

- Types
  - Request
  - Reply
  - Message (i.e. Datagrams)
- Code generation will group each message type into a class hierarchy or data type

Example

```
//
// ACNET Chat Server Protocol
//

request Connect {
   string name;
}

reply ReceivedId {
   int16 id;
}

reply ReceivedMessage {
   string sender;
   string text;
}

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

Connect -> single ReceivedId | multiple ReceivedMessage;
```

Compiling to Source

- pc -l c++ Chat.proto
  - Creates Chat.h and Chat.cpp
- pc -l java -p gov.fnal Chat.proto
  - Creates Chat.java
  - -p option generates a package statement

C++ Mapping

Protocol Type	C++ Type
bool	bool
int16	int16_t
int32	int32_t
int64	int64_t
double	double
string	std::string
binary	std::vector<uint8_t>
struct N	struct N
T [ ]	std::vector<T>
optional T	std::auto_ptr<T>

C++ Example

```
#include <Chat.h>

uint16_t id;

...

// Setup a message
protocol::Chat::message::Say say;

say.id = id;
say.name.reset(new std::string(“Watson”));
say.text = “Mr. Watson. Come Here. I need you.”;

std::ostringstream os;
say.marshal(os);

// Send the message
```

C++ Example Cont'd

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

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

Handler callback;
std::istringstream is;

// Receive message into variable 'is'
...

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

ptr->deliverTo(callback);
```

Java Mapping

Protocol Type	Java Type
bool	boolean
int16	short
int32	int
int64	long
double	double
string	String
binary	byte[ ]
struct N	class N
T [ ]	T [ ]
optional T	object reference

Java Example

```
import gov.fnal.Chat;
import java.io.ByteArrayOutputStream;

Chat.ReceivedId id;

...

// Setup message

Chat.Message.Say say = new Chat.Message.Say();
say.id = id;
say.name = “Watson”;
say.text = “Mr. Watson. Come here. I need you.”;

ByteArrayOutputStream os = new ByteArrayOutputStream();
say.marshal(os);

// Send message
```

Java Example Cont'd

```
import gov.fnal.Chat;
import java.io.InputStream;

class Handler implements Chat.Reply.Receiver {
   void handle(Chat.Reply.ReceivedId msg) {}
   void handle(Chat.Reply.ReceivedMessage msg) {}
}

Handler handler = new Handler();
ByteArrayInputStream is = new ByteArrayInputStream();

// Receive message into variable 'is'
...

Chat.Reply.unmarshal(is).deliverTo(handler);
```

Future Plans

- Document (man page, wiki would have more details)
- Add other language generators
  - C
  - python
  - Objective-C
  - OCaml
- Change C++ code to use iterators instead of streams
- Add a dictionary field type