High Performance Working Group
Google Protocol Buffers for FIX Encoding
User Guide

April 16, 2013

Version 0.2

DISCLAIMER

THE INFORMATION CONTAINED HEREIN AND THE FINANCIAL INFORMATION EXCHANGE PROTOCOL (COLLECTIVELY, THE "FIX PROTOCOL") ARE PROVIDED "AS IS" AND NO PERSON OR ENTITY ASSOCIATED WITH THE FIX PROTOCOL MAKES ANY REPRESENTATION OR WARRANTY, EXPRESS OR IMPLIED, AS TO THE FIX PROTOCOL (OR THE RESULTS TO BE OBTAINED BY THE USE THEREOF) OR ANY OTHER MATTER AND EACH SUCH PERSON AND ENTITY SPECIFICALLY DISCLAIMS ANY WARRANTY OF ORIGINALITY, ACCURACY, COMPLETENESS, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. SUCH PERSONS AND ENTITIES DO NOT WARRANT THAT THE FIX PROTOCOL WILL CONFORM TO ANY DESCRIPTION THEREOF OR BE FREE OF ERRORS. THE ENTIRE RISK OF ANY USE OF THE FIX PROTOCOL IS ASSUMED BY THE USER.

NO PERSON OR ENTITY ASSOCIATED WITH THE FIX PROTOCOL SHALL HAVE ANY LIABILITY FOR DAMAGES OF ANY KIND ARISING IN ANY MANNER OUT OF OR IN CONNECTION WITH ANY USER'S USE OF (OR ANY INABILITY TO USE) THE FIX PROTOCOL, WHETHER DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL (INCLUDING, WITHOUT LIMITATION, LOSS OF DATA, LOSS OF USE, CLAIMS OF THIRD PARTIES OR LOST PROFITS OR REVENUES OR OTHER ECONOMIC LOSS), WHETHER IN TORT (INCLUDING NEGLIGENCE AND STRICT LIABILITY), CONTRACT OR OTHERWISE, WHETHER OR NOT ANY SUCH PERSON OR ENTITY HAS BEEN ADVISED OF, OR OTHERWISE MIGHT HAVE ANTICIPATED THE POSSIBILITY OF, SUCH DAMAGES.

DRAFT OR NOT RATIFIED PROPOSALS (REFER TO PROPOSAL STATUS AND/OR SUBMISSION STATUS ON COVER PAGE) ARE PROVIDED "AS IS" TO INTERESTED PARTIES FOR DISCUSSION ONLY. PARTIES THAT CHOOSE TO IMPLEMENT THIS DRAFT PROPOSAL DO SO AT THEIR OWN RISK. IT IS A DRAFT DOCUMENT AND MAY BE UPDATED, REPLACED, OR MADE OBSOLETE BY OTHER DOCUMENTS AT ANY TIME. THE FPL GLOBAL TECHNICAL COMMITTEE WILL NOT ALLOW EARLY IMPLEMENTATION TO CONSTRAIN ITS ABILITY TO MAKE CHANGES TO THIS SPECIFICATION PRIOR TO FINAL RELEASE. IT IS INAPPROPRIATE TO USE FPL WORKING DRAFTS AS REFERENCE MATERIAL OR TO CITE THEM AS OTHER THAN “WORKS IN PROGRESS”. THE FPL GLOBAL TECHNICAL COMMITTEE WILL ISSUE, UPON COMPLETION OF REVIEW AND RATIFICATION, AN OFFICIAL STATUS ("APPROVED") OF/FOR THE PROPOSAL AND A RELEASE NUMBER.

No proprietary or ownership interest of any kind is granted with respect to the FIX Protocol (or any rights therein).

Document History

Revision	Date	Author	Revision Comments
V0.1	2013-04-10	Greg Malatestinic, Jordan & Jordan Sara Rosen, EBS Alessandro Triglia, OSS Nokalva	Initial draft.
V0.2	2013-04-16	Greg Malatestinic, Jordan & Jordan Sara Rosen, EBS Alessandro Triglia, OSS Nokalva	Changes due to feedback from HPWG review.

1 Introduction

The High Performance working group (HPWG) was created by the FIX Protocol Limited (FPL) Global Steering Committee in July 2012. Fred Malabre and Mark Reece are co-chairing the working group. The charter of the working group is to identify opportunities to enrich high performance financial messaging and propose specific enhancements to FIX to address these identified opportunities, including application level, session level (recovery), and encoding.

Several encoding subcommittees were formed. Among them is The Google Protocol Buffers encoding subgroup, led by Sara Rosen and Greg Malatestinic.

The mandate of the Google Protocol Buffer encoding subgroup was to define a mechanism for mapping FIX to Protocol Buffers which can support the rich semantics of the FIX language while meeting the performance goals of the High Performance trading community.

The Google Protocol Buffer encoding utilizes existing industry standard encodings and applies them to the FIX domain. Google Protocol Buffers is a mature binary encoding which can be applied "out of the box" to support FIX language semantics. However, the richness of this technology supports multiple solutions to mapping the existent FIX data types to a binary protocol. In the interest of standardization, this document will specify a normative encoding of the FIX data types in protocol buffers for interoperable exchange of financial data over FIX. Additionally, this document will spell out best practice guidelines for maximizing the efficiency of protocol buffer encodings.

1.1 Google Protocol Buffers for FIX

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data.

Protocol buffers are an attractive encoding technology for FIX messages for a number of reasons:

- Open – The complete GPB specification has been published and Google has clearly stated that GPB is available for use by anyone. Protobuf is language and platform independent, supporting C++, Java, Python and other languages on most common operating systems. Clients do not need to write their own encoders/decoders, but can rather incorporate the Google provided API's within their software.
- Flexible - Like FIX, protobuf messages are tagged. This supports optional fields, which can be leveraged to support multiple use cases via a single message definition.
- Versioning - Protobuf messages support backward compatibility across older versions of message specifications. Messages can be modified without breaking applications that utilize prior message definitions, thus reducing deployment dependencies between message producers and consumers.
- Performance - Protobuf produces compact binary messages to maximize network bandwidth and reduce serialization delays. Protobuf serialization is CPU efficient, with a low encoding and decoding overhead and a minimal footprint.

- Ease of Use - Protobuf is well documented and easy to learn. Encoders and decoders can be embedded directly in the target applications, for "do it yourself" FIX implementations.
- Structured - Protobuf supports repeating fields and nested messages, similar to FIX repeating groups. Unlike FIX, nested messages can have the same field names at any level of nesting, so there is no need for artificial constructs like Parties, NestedParties, NestedParties2, etc.
- Templates - Similar to FAST[SM] (http://www.fixprotocol.org/fast), protobuf message structures are defined in message templates, which are used by message recipients to decode the data. Default values can be populated based on template definitions, further reducing messaging bandwidth for the most commonly used values.
- Adoption - Protobuf has a large and active user community. It is also a mature technology, in use since 2001. It also serves as the underlying messaging technology for a number of open source frameworks.

1.2 References

[1] https://developers.google.com/protocol-buffers/docs/overview
GPB project home – Language Guide and Encoding Specification.

[2] Add hyperlink to FIX Repository Mapping document
Specification for mapping the FIX Unified Repository to GPB.

[3] http://www.fixprotocol.org/fastspec
FIX Adapted for Streaming[SM] (FAST[SM]) specifications.

2 Google Protocol Buffers Overview

2.1 Templates

GPB data structures are defined as "messages" in a ".proto" template file. The template provides a machine-readable interface definition which is input to language-specific code generators to produce message encoders and decoders. These expose getters and setters for each field as well as methods to serialize/parse the whole structure to/from raw bytes. Senders and receivers must each have a copy of the proto file to encode or parse the message. These are typically exchanged out-of-band. Dynamic transmission is also possible, though less performant.

The template defines the structure and properties of the message and its fields. Messages may be nested or contain repeating groups of messages. Fields may be specified as required or optional, and may contain default values. Templates may also contain human-readable comments and namespace identifiers to prevent message collisions between packages. A single proto file may define multiple message types, and can also reference messages defined in external proto files, via the "import" command.

Here is a sample template for a FIX Logout message:

```
message Logout { // FIX MSG_ID = 5
   optional Header header = 1;
   optional int32 sessionStatus = 2 [default = 4];  //Session status at time of logout
   optional string text = 3;
   optional bytes encodedText = 4;
   optional Trailer trailer = 5;
};
```

2.2 Field Properties - Names and Tags

Every protobuf field has an ASCII field name and an integer field tag. The field name is the data structure's human-readable symbolic name. Field names are not present in the binary payload of the message. Rather, the field name is mapped to the field's accessor methods to encode/decode the binary message from/to the data structures specified in the message's proto template. For FIX encoding, the protobuf field names are based on the FIX field and component names.

The field tag is the machine-readable integer which is embedded in the binary message, and which enables the GPB decoder to map the value from the wire-stream value to its associated data structure.

Tags with a value of 1-15 occupy one byte. Tags in the range 16-2047 take two bytes, and larger values take more. A highly optimized schema will determine the most frequently occurring message elements and assign these to the lower-value tags.

For this reason, and also to support mapping a single FIX field to multiple protobuf fields, the GPB Encoding Subgroup has decided not to utilize the FIX tags for the protobuf tag ids. Rather, tag numbers

were sequentially auto-generated based on field order in the FIX repository. Clients who wish to optimize bandwidth utilization may re-sequence the fields based on the relative field utilization in their context. Unlike FIX tag numbers, protobuf field tags need not be globally unique, but must only be unique per message. The standard convention is to utilize the lowest tag numbers as possible so as to minimize the size of an encoded message.

2.3 Message Encoding

A top level message has no envelope structure. The message is just a sequence of its member key-value fields. Embedded messages are encoded similar to strings, e.g. they contain a length-delimited sequence of bytes. Like all fields, this embedded message is preceded by its field tag, which associates it with its message type. This allows the protobuf decoder to recursively invoke the appropriate parser for the byte string containing the embedded message.

Example of a simplified Market Data Snapshot message:

```
Message Instrument {
        string symbol = 1;
        string securityId = 2;
}

Message MarketDataSnapshotFullRefresh {
        string mdRequestId = 1;
        Instrument  instrument =2;
        MDFullGrp mdFullGrp =3;
}
```

When parsing the MarketDataSnapshotFullRefresh message, presence of tag=2 causes the parser to extract the length-delimited sequence of bytes representing the instrument field. It then invokes the parser of the Instrument class to decode the nested message.

2.4 Varint Encoding

One significant optimization of the GPB encoding is the use of Varints. Varints are “variable length integers”, where the length of the field’s wire representation depends on the value of the field, rather than its data definition. The sizes of the varint types int32, int64, uint32, uint64, sint32, and sint64 are used to determine the size of the source and target fields, rather than the size of the wire encoding. There is therefore no loss of wire efficiency in defining a data field as a 64 bit rather than as a 32 bit integer.

2.5 Optional Fields

Protobuf is a tagged value encoding. This means that the presence or absence of a field can be detected by the presence of the field tag in the output stream. As a result, Protobuf is especially efficient for encoding /decoding messages with sparse optional fields, i.e. messages where a large number of fields are empty. Optional fields which are not set by the encoder carry no bandwidth overhead.

There are no “null” values in protobuf. When a message is parsed, if an optional field is not present in the message, the GPB decoder sets the value of the associated field to its default value. Default values

may be specified explicitly in the proto template. Otherwise, a default value is determined implicitly, based on field type.

It is occasionally useful to determine whether the value of a decoded field was explicitly set by the message producer or whether it was set to a default value by the decoder. For each optional field, XXX, the GPB pre-compiler generates a Boolean accessor method, HasXXX(), which can be utilized to determine whether or not the field was present on the wire.

2.6 Repeating groups

The GPB encoding supports repeating groups. Repeating groups may appear any number of times in the message, including zero times. The order of the repeated group will be preserved in the protocol buffer. A repeating group with a recurrence of zero does not consume message bandwidth.

GPB also supports a packed encoding of repeating groups. Only repeating groups which contain only primitive numeric types can be declared packed. Rather than having the tag and type appear for each element of the repeating group, the entire list is encoded as a single length-delimited key-value pair containing the concatenated values of all the data elements.

Source: https://developers.google.com/protocol-buffers/docs/encoding#optional

3 FIX Mapping

The following is a brief discussion of a few of the salient points of the mapping. A full description of the procedure to map the full contents of the FIX Unified Repository to GPB can be found at hyperlink.

3.1 Naming conventions

The names of the fields in the FIX Repository were used as the basis for the name attribute of the GPB fields, with the following modifications.

GPB fields are strictly camel case, beginning with lower case. Upper-case acronyms embedded in FIX field names are replaced with their with their camel-case equivalent. The first character of the field name is then set to lower case. Examples: onBehalfOfCompId, ioiNaturalFlag, mdEntryPx.

GPB message names are camel case, beginning with an upper case. This includes both stand-alone messages and FIX Components which are mapped to embedded GPB messages. Upper-case acronyms embedded in FIX field names are replaced with their with their camel-case equivalent. The first character of the field name is then set to upper case. Examples: NewOrderSingle, MdIncGrp.

Instances of the embedded messages are themselves GPB fields, and follow the lower-case convention.

Example: The FIX component “PtysSubGrp” is mapped to a GPB message type of the same name.

```
message PtysSubGrp {
        optional string partySubId = 1;
        optional PartySubIdTypeUnion partySubIdType = 2;
}
```

Instances of this message type are embedded in other GPB messages as “ptysSubGrp”, e.g.

```
message Parties {
        optional string partyId = 1;
        optional PartyIdSourceEnum partyIdSource = 2;
        optional PartyRoleEnum partyRole = 3;
        repeated PtysSubGrp ptysSubGrp = 4;
}
```

FIX enumeration types are mapped to a GPB enums. These follow the same convention as other GPB message types (camel-case beginning with an upper case), concatenated with the string “Enum”.

Enumerated-value names are mapped to upper-case, and prepended with the name of the GPB enum (minus the “Enum”) and the underscore character.

Example:

```
enum HandlInstEnum {
        HandlInst_AUTOMATED_EXECUTION_NO_INTERVENTION = 0;
        HandlInst_AUTOMATED_EXECUTION_INTERVENTION_OK = 1;
        HandlInst_MANUAL_ORDER = 2;
}
```


The reason for this compound naming convention is the GPB requirement that all enumerated values in a single proto file must have unique names. For example, both the ExecType and OrdStatus enumerations have a value of "Cancelled", which would collide. Following the convention above, the two enumerated values are ExecType_CANCELLED, and OrdStatus_CANCELLED, which preserve their uniqueness.

3.2 Timestamps

ASCII timestamps are particularly inefficient. For example, a millisecond timestamp in the format YYYYMMDD-HH:MM:SS.sss consumes 21 bytes, and must be converted to a numeric format for comparisons or calculations.

In mapping FIX to GPB,timestamps are represented by an integer offset, based on an epoch and a timeunit. Timeunits may be days, hours, minutes, seconds, milliseconds, microseconds, nanosecond, or picoseconds. The epoch is typically Jan 01, 1970, although other epochs may be defined for special purposes. The size of the integer (32 or 64 bits) will depend on the choice of timeunit and epoch.

Timeunit and epoch are not sent over the wire, but are rather specified as a comment of the proto file or in an organization's Rules of Engagement document. The FIX repository may also be annotated with the encoding attributes "epoch" and "timeUnit", which can be used to automatically generate an appropriately sized integer for the timestamp offset.

3.3 Decimal Prices

A special challenge exists for mapping FIX decimal prices to binary data. Although all binary encodings support floating point binary numbers, the general convention in the financial industry is not to use floating point representation for decimal prices due to rounding issues. The other alternative, in use by the existing FIX tag-value encoding, which represents decimal prices as ASCII strings, is inefficient both in terms of bandwidth utilization and in the processing overhead incurred when comparing or sorting ASCII prices.

The solution implemented for FIX over GPB is to represent decimal prices by a pair of integers – a mantissa and an exponent, where the exponent determines placement of the decimal point.

$$\textit{Decimal value} = \textit{Mantissa} \times 10^{\textit{exponent}}$$

For example, to represent the decimal price of 12.3456, mantissa=123456 & exponent=-4.

Alternative 1: When all price instances of a given price field can be handled with a uniform exponent, it is recommended that the exponent not be included on the wire, but rather communicated out-of-band, either as a comment on the price field in the proto file or specified in the Rules of Engagement document. In that case only a single integer field is needed to map the price field, for example:

```
optional uint32 lastPx = 1; // exponent = -6
```

Organizations are encouraged to carefully consider current and future usage scenarios before deciding on a default exponent. For example, mid-prices, reference rates, or other derived market data fields will often require extra digits of precision. It is better to err on the side of allocating extra digits of precision as a change to an implied exponent will require notification to all consumers of the protocol.

Alternative 2: When not all prices can be handled with a uniform exponent, it is recommended to create a GPB data type comprised of two fields, mantissa and exponent, where a default value of the exponent is provided in the proto template. Then prices which conform to the default need only have the mantissa sent on the wire, but flexibility is still maintained to override the default exponent when needed. Examples of such datatypes can be found in the FIX Repository to GPB mapping document. For example, the message definition below defines a decimal data type with a uint64 mantissa and a default exponent of -4 which can be overridden on the wire, when necessary.

```
message UDecimal64EMinus4 {
        optional uint64 mantissa = 1;
        optional uint32 exponent = 2 [default = -4];
}
```

This datatype could then be used to define a FIX price field, for example:

```
optional UDecimal64EMinus4 lastPx = 1;
```

To transmit a price of 12.3456, which conforms to the default exponent, it is only necessary to set the value of the mantissa, eg.

```
lastPx.mantissa = 123456;
```

To transmit a price of 12.34567, which does not conform to the default exponent, both fields can be set:

```
lastPx.mantissa = 1234567;
lastPx.exponent = -5;
```

Just as in alternative one, organizations are advised to carefully consider future needs when determining a default exponent precision for decimal prices. GPB default values are not forward compatible, since the default value is applied by the decoder. Data corruption would occur if the sender and receiver do not share the same default exponent values.

When no default precision can be determined for a given use case, the default exponent should be set to zero. This makes it clear that there is no pre-defined decimal offset for this field, and any decimal price will need to be sent with explicit values for both the mantissa and the exponent.

Alternative 3: Occasionally, fractional prices may best suit the domain model, for example when quoting in thirds, e.g. $123\ ^{2}/_{3}$. Then the price should be declared as a binary type float:

```
optional float lastPx =1;
```

The FIX Repository may be annotated with encoding attributes to drive the automated mapping of FIXPrice fields to the desired GPB encoding. The relevant encoding attributes are: “isBinaryFloat” “isFixedPoint “, “numBits”, “minValue”, and “maxValue”. See section 3.5 below for more information on the usage of encoding attributes to auto-generate GPB proto files from a FIX Repository.

3.4 MultipleCharValue

FIX fields of type MutilpleCharValue (or MultipleStringValue) are mapped to a sequence of enumerated values, using a GPB “repeated group.” Inclusion of each element of the repeating group indicates selection of its associated optional value.

For example, the FIX field, ExecInst, is a MultipleCharValue field where the choice of values is selected from a pool of enumerated values. This maps to the GPB field definition:

```
repeated ExecInstEnum execInst = 21 [packed = true];
```

where the enumerated values are defined in the following GPB enum:

```
enum ExecInstEnum {
        ExecInst_STAY_ON_OFFER_SIDE = 0;
        ExecInst_NOT_HELD = 1;
        ExecInst_WORK = 2;
        ExecInst_GO_ALONG = 3;
        // ...
}
```

Note that repeating groups of enumerations can take advantage of the optimized "packed" encoding option for repeating groups, since they contain no complex data structures.

3.5 Automated Mapping from FIX Repository

The FIX to GPB Mapping Specification [2] describes how to take as input an XML element, whose syntax and semantics comply with those of the <fix> element of the FIX Unified Repository, and generate a Google Protocol Buffers schema, consisting of a set of GPB message and enumeration types.

This specification provides rules for mapping FIX messages and component blocks to GPB messages, FIX fields to GPB field definitions, and enumerated field values to GPB enum definitions. It shows that for each FIX message or component, the mapping of its field items will depend on their effective types which include but are not limited to the following:

- Component block
- Repeating block
- Datatype restricted by the enumeration specified in <enum> child elements
- Datatype with no restricted values
- Multiple-value type with values restricted by the enumeration specified in <enum> child elements
- Multiple-value type with no restricted values

The specification also shows that the mapping utilizes encoding attributes to annotate various elements of the repository such as <datatype>, <field> and <fieldRef>. These attributes are used as the basis to determine the most favorable encoding when there are several options. They include:

- minVaue - minimum permitted value of the integer datatype
- maxValue - maximum permitted value of the integer datatype
- minLength - minimum permitted length of the string datatype
- maxLength - maximum permitted length of the string datatype
- numBits - maximum size in bits of the data element
- isFixedPoint

- exponent - a fixed exponent for a fixed-point decimal number or the default exponent for a floating-point decimal number

To illustrate how the effective datatype influences the mapping, let's consider the FIX field "Side". Since it is a char type one may think it should be mapped to the GPB bytes type which is used to encode char values. But since "Side" is restricted by enumerated values "Buy", "Sell", "Sell Short", etc., it's effective datatype causes the following GPB enum type to be generated:

```
enum SideEnum {
        Side_BUY = 0;
        Side_SELL = 1;
        Side_BUY_MINUS = 2;
        Side_SELL_PLUS = 3;
        Side_SELL_SHORT = 4;
        Side_SELL_SHORT_EXEMPT = 5;
        // etc.
}
```

Any FIX message or component containing the Side field will generate a GPB message containing a field definition of field type "SideEnum". For example, the FIX message NewOrderSingle will cause the following to be generated:

```
message NewOrderSingle {
        // …
        optional SideEnum side = n;
        // …
}
```

To illustrate how encoding attributes may influence the mapping, let's consider the FIX integer field, LiquidityNumSecurities, with the encoding attributes *minValue* equal to 0 and *maxValue* equal to 1000. We can conclude that all values of this field are positive and only two bytes are needed to store its value. So this is mapped to the GPB type uint32. Had *maxValue* been a number larger than 2^32 we would map to the GPB type uint64.

One can refer to the specification, [2], for complete details of the mapping.

4 Usage Guidelines

4.1 Optional/Required Fields

Google recommends that all fields be marked as optional. This is especially important to support message evolution.

> **"Required Is Forever** -You should be very careful about marking fields as required. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some engineers at Google have come to the conclusion that using required does more harm than good; they prefer to use only optional and repeated. However, this view is not universal."
> https://developers.google.com/protocol-buffers/docs/proto

Consequently, the GPB Encoding subgroup recommends that all FIX fields in the protobuf message definition be marked as optional. Semantically, fields can still be required, but enforcement of required message fields should be done on the application level, rather than by the GPB parser. Rules of engagement documents should be used to specify which fields are required in a given context. When such a "required" field is missing, the message should be rejected with an application or session level rejection message.

4.2 Versioning

Protobuf supports both forward and backward compatibility of message specifications, so that message senders and receivers do not have to utilize the same version of the encoding template. This is accomplished as follows:

- Backward compatibility – In this case, the message recipient has an older version of the template which contains fewer messages and fields than a newer version used by the sender. The recipient GPB decoder will drop the unknown fields.
- Forward compatibility – The message recipient has a newer version of the template which contains more messages and fields than an older version used by the sender. The recipient GPB decoder will set the values of the missing fields to their default values.

Protobuf has strong support for message evolution. Fields may be added. Optional fields may be upgraded to repeating fields. Fields may also be deleted, as long as their tags are not reused. For a full description of the do's and don'ts of protobuf message evolution, see "Updating A Message Type" in the Protobuf Language Guide.
One important consideration relates to default values. Since default values are applied by the decoder based on its version of the proto file, changes to default values by the message sender will not be known to recipients with an earlier version of the proto file. When such a change is necessary, the actual [new] value should be set by the sender in all cases, thus overriding the recipient's [old] default setting.

A similar rule applies to enum values. Enumerations may be extended, but if the value of the enumeration is unknown to the decoder, it will drop the enumeration as an unknown field.

4.3 Framing for Message Streams

When multiple messages are serialized in a stream, it is necessary to detect where one message ends and the next begins. The Protocol Buffer wire format is not self-delimiting, so protocol buffer parsers cannot itself determine where a message ends. This is typically the role of the session layer, and is outside the scope of the GPB encoding rules.

4.4 Message Type

In order to decode a GPB message it is necessary to use invoke one of the parsing methods (e.g.ParseFromString or ParseFromIstream) of the class to which the message will be deserialized. Decoding a FIX message to extract the message type therefore presents a “chicken and egg” type conundrum.

The recommended solution is that the FIX message type should be specified outside the protobuf payload, using the same mechanism as delivery of message size. Various session implementations will have their own mechanisms for transmitting this type of information, for example as JMS custom properties of a JMS message stream. Alternatively, for bare-boned delivery of GPB FIX messages over TCP, the message size and message type may be provided in a fixed-length header preceding the GPB message body.

4.5 Direct Access

High-performance applications may wish to implement “direct access” to selective fields in a GPB message, without incurring the processing overhead of message parsing.

This can be implemented with the following restrictions:

1. Direct access fields must be present in all messages.
2. All “direct access” fields should appear at the front of the GPB message, before any optional fields. Field numbers of direct access fields should be in incremental order and should be lower in value than the field numbers of any non-direct-access fields.
3. Direct access fields should utilize the fixed-length GPB data types, such as fixed32 and fixed64, rather than the variable length encoding data types such as uint32 and uint64.
4. Repeating groups with a variable number of repetitions may not be direct access fields.
5. Variable length strings may not be direct access fields.

This approach supports a hybrid approach to the tradeoffs between message flexibility and performance optimization. A single message definition may contain both critical high-performance fields which support direct-access, while less frequently occurring fields can utilize the full flexibility afforded by GPB.

Note that while the Google-provided protobuf implementations serialize fields in tag number order, this is not a requirement of the specification. Users of non-Google provided implementations who depend on consistent field ordering should verify that their implementations honor with this ordering policy.

4.6 Routing

Protobuf messages can be routed through middlemen who may be aware of only a subset of the template with no loss of information. Intermediate servers that don't need to inspect the data can pass through the data without needing to know about all the fields.

For example, a simplified proto definition can be created containing only the FIX header, or only select routing fields such as MarketId or SecurityType. An intermediary application using this proto file would parse only a subset of the message, extracting only the fields necessary for routing, but could still forward the entire message payload to the appropriate target application with no loss of data.

This mechanism contributes to the speed of the routing application, and also makes the intermediary system resilient to changes in application-level features in which it has no interest.

4.7 Utilities

Google provides proto parsers for Java, C++, and Python, but decoders for many other languages have been developed by the open-source community, with varying levels of maturity.

Many open-source GPB utilities are also available, including tools for re-formatting GPB messages as XML, JSON, or human-readable, text-based formats.

See: http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns

4.8 Self-describing Messages

See https://developers.google.com/protocol-buffers/docs/techniques#self-description

5 Sample Messages

The following listing shows a GPB message definition for a FIX OrderCancelRequest message, its nested components and enumeration types. Many of the optional FIX fields have been omitted.

```
message OrderCancelRequest {
        optional StandardHeader standardHeader = 1;
        optional string origClOrdId = 2;
        optional string orderId = 3;
        optional string clOrdId = 4;
        optional string clOrdLinkId = 5;
        optional string account = 6;
        optional Instrument instrument = 7;
        optional SideEnum side = 8;
        optional uint64 transactTime = 9; // epoch=19700101 timeunit=milliseconds
        optional OrderQtyData orderQtyData = 10;
        optional string text = 11;
        optional StandardTrailer standardTrailer = 12;
}

message StandardHeader {
        optional string senderCompId = 1;
        optional string targetCompId = 2;
        optional string onBehalfOfCompId = 3;
        optional string deliverToCompId = 10;
        optional uint32 msgSeqNum = 13;
        optional PossDupFlagEnum possDupFlag = 22;
        optional PossResendEnum possResend = 23;
        optional uint64 sendingTime = 24; // epoch=19700101 timeunit=milliseconds
        optional uint32 lastMsgSeqNumProcessed = 29;
        repeated HopGrp hopGrp = 30;
}

enum PossDupFlagEnum {
        PossDupFlag_ORIGINAL_TRANSMISSION = 0;
        PossDupFlag_POSSIBLE_DUPLICATE = 1;
}

enum PossResendEnum {
        PossResend_ORIGINAL_TRANSMISSION = 0;
        PossResend_POSSIBLE_RESEND = 1;
}

message HopGrp {
        optional string hopCompId = 1;
        optional uint64 hopSendingTime = 2;
        optional uint32 hopRefId = 3;
}

message Instrument {
        optional string symbol = 1;
        optional string securityId = 3;
        optional SecurityIdSourceEnum securityIdSource = 4;
}

enum SecurityIdSourceEnum {
        SecurityIdSource_CUSIP = 0;
```



```
        SecurityIdSource_SEDOL = 1;
        SecurityIdSource_QUIK = 2;
        SecurityIdSource_ISIN_NUMBER = 3;
        SecurityIdSource_RIC_CODE = 4;
}

enum SideEnum {
        Side_BUY = 0;
        Side_SELL = 1;
        Side_BUY_MINUS = 2;
        Side_SELL_PLUS = 3;
        Side_SELL_SHORT = 4;
        Side_SELL_SHORT_EXEMPT = 5;
        Side_UNDISCLOSED = 6;
        Side_CROSS = 7;
        Side_CROSS_SHORT = 8;
        Side_CROSS_SHORT_EXEMPT = 9;
}

message OrderQtyData {
        optional UDecimal64E0 orderQty = 1;
        optional UDecimal64E0 cashOrderQty = 2;
        optional UDecimal64EMinus2 orderPercent = 3;
}

message UDecimal64E0 {
        optional sint64 mantissa = 1;
        optional uint32 exponent = 2 [default = 0];
}

message UDecimal64EMinus2 {
        optional sint64 mantissa = 1;
        optional uint32 exponent = 2 [default = -2];
}
```

