Protocol Buffers

http://code.google.com/p/protobuf/

Overview

- What are Protocol Buffers
- Structure of a .proto file
- How to use a message
- How messages are encoded
- Important points to remember
- More Stuff

What are Protocol Buffers?

- Serialization format by Google
- used by Google for almost all internal RPC protocols and file formats

  (currently 48,162 different message types defined in the Google code tree across 12,183 .proto files. They're used both in RPC systems and for persistent storage of data in a variety of storage systems.)
- Goals:
  - Simplicity
  - Compatibility
  - Performance

Comparison XML - Protobuf

- Readable by humans ↔ binary format
- Self-describing ↔ Garbage without .proto file
- Big files ↔ small files (3-10 times)
- Slow to serialize/parse ↔ fast (20-100 times)
- .xsd (complex) ↔ .proto (simple, less ambiguous)
- Complex access ↔ easy access

Comparison XML – Protobuf (cntd)

```
<person>
   <name>John Doe</name>
   <email>jdoe@example.com</email>
 </person>
```

(== 69 bytes, 5-10'000ns to parse)

```
Person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

(== 28 bytes, 100-200ns to parse)

```
cout << "Name: "
       <<
person.getElementsByTagName("nam
e")->item(0)->innerText()
       << endl;
cout << "E-mail: "
       <<
person.getElementsByTagName("emai
l")->item(0)->innerText()
       << endl;
```

```
cout << "Name: " << person.name() <<
endl;
cout << "E-mail: " << person.email() <<
endl;
```

Example

```
message Person {
  required string name  = 1; // name of person
  required int32  id    = 2; // id of person
  optional string email = 3; // email address

  enum PhoneType {
    MOBILE = 0;
    HOME   = 1;
    WORK   = 2;
  }

  message PhoneNumber {
    required string    number = 1;
    optional PhoneType type   = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

From .proto to runtime

- Messages defined in .proto file(s)
- Compiled into source code with protoc
  - C++
  - Java
  - Python
  - More languages via AddOns (C#, PHP, Perl, ObjC, etc)
- Usage in code
- Passed via network / files

Message Definition

- Messages defined in .proto files
- Syntax: `Message [MessageName] { ... }`
- Can be nested
- Will be converted to e.g. a C++ class

Message Contents

- Each message may have
  - Messages
  - Enums:

```
enum <name> {
  valuename = value;
}
```

  - Fields
- Each field is defined as

```
<rule> <type> <name> = <id> {[<options>]};
```

Field rules

- **Required**
  - exactly once (`msg.`**`fieldname`**`()`)
- **Optional**
  - None or one
  - Query existence (`msg.has_`**`fieldname`**`()`)
- **Repeated**
  - None to infinite (ordered array)
  - Query count (`msg.`**`fieldname`**`_size()`)
  - Use option `packed=true` for efficient encoding

Required is **required**

- Field rule *required* is a tough decision
- Once a field is *required*, it must stay *required* forever unless compatibility between versions is to be broken (not such a good idea)
- Some engineers at Google advise to never use *required*

Field types

.proto type	Note	C++ type
float / double		float / double
int32 / int64	Variable-length, primarily suited for pos. numbers	int32 / int64
uint32 / sint32 (dto. ...64)	Variable-length, un/signed	(u)int32 / (u)int64
(s)fixed32, (s)fixed64	Fixed length (un/signed), better suited for $>2^{28 / 56}$	(u)int32 / (u)int64
bool		bool
string	UTF-8 **or** 7-bit ASCII	std::string
bytes	Arbitrary sequence of bytes	std::string
Message or Enum type		Corresponding class

Field id (tag)

- Each field has a unique tag (id) (1 .. 2^{29}-1)

  (Unique per message definition)
- Variable length encoded – 1..15 == one byte
- Identifies the field within the binary format
  - i.e. field names are **NOT** used in the encoded data
- Assigned for life

Options, namespaces and importing

- Options:
  - [default = value] → sets a default value (beware: default values are not encoded!)
  - [packed = false/true] → better encoding of *repeated*
  - [deprecated = false/true] → marks a field as obsolete
  - [optimize_for = SPEED/CODE/LITE_RUNTIME]
  - Java package and outer classname
- Namespaces/packages can be defined via e.g. `package com.example.message`
- Importing of messages defined in other files via `import „filename.proto“`

Example (again)

```
message Person {
  required string name  = 1; // name of person
  required int32  id    = 2; // id of person
  optional string email = 3; // email address

  enum PhoneType {
    MOBILE = 0;
    HOME   = 1;
    WORK   = 2;
  }

  message PhoneNumber {
    required string    number = 1;
    optional PhoneType type   = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Overview – Where are we

- What are Protocol Buffers
- Structure of a .proto file
- **How to use a message**
- How messages are encoded
- Important points to remember
- More Stuff

From .proto to code

- protoc compiler creates classes in desired language
- Example: `protoc -cpp_out=. person.proto` will create person.pb.cc and person.pb.h

Generated code

```
// name
bool has_name() const;
const;
void clear_name();
const string& name() const;
void set_name(const string& value);
)
void set_name(const char* value);
string* mutable_name();

// phone
inline int phone_size() const;
inline void clear_phone();
inline const RepeatedPtrField<Person_PhoneNumber>&
  phone() const;
inline RepeatedPtrField<Person_PhoneNumber>*
  mutable_phone();
inline const Person_PhoneNumber& phone(int index) const;
inline Person_PhoneNumber* mutable_phone(int index);
```

```
// id
bool has_id()

void clear_id();
int32_t id() const;
void set_id(int32_t
```

Setting values in a message

```
#include "person.pb.h"

Person person;
person.set_name("Hans Mustermann");
person.set_email("hans@muster.mann");

// std::string *name = person.mutable_name();
// *name = "Hans Mustermann";

Person::PhoneNumber *phone;
phone = person.add_phone();
phone->set_number("030 12345678");
phone->set_type(Person::WORK);
phone = person.add_phone();
phone->set_number("0170 987654321");
phone->set_type(Person::MOBILE);

// check for validity: person.IsInitialized() == true ?
```

Serializing

- Serialize data via
  - `std::string person.SerializeAsString()`
  - `person.SerializeToString(std::string*)`
  - `person.SerializeToFileDescriptor(int)`
  - `person.SerializeToOstream(std::ostream*)`
  - `person.SerializeToArray(char*, int size)`
- Example

```
std::ofstream file(filename,
        std::ios::out | std::ios::binary);
if (false == file.fail()) {
  person.SerializeToOstream(&file);
}
```

Parsing

- Parse via
  - `person.ParseFromIstream(std::istream*)`
  - `person.ParseFromString(std::string)`
  - `person.ParseFromFileDescriptor(int)`
  - `person.ParseFromArray(const char*, int)`
- Example:

```
std::ifstream file(filename,
        std::ios::in | std::ios::binary);
if (false == file.fail()) {
  person.ParseFromIstream(&file);
}
```

Retrieving values from a message

```
#include "person.pb.h"

Person person;
person.ParseFromIstream(file);
if (person.IsInitialized()) {
  cout << "Name: " << person.name() << endl;
  if (person.has_email()) {
     cout << "Email: " << person.email() << endl;
  }
  for (int i=0; i < person.phone_size(); i++) {
    cout << "Phone: " << person.phone(i).number()
         << endl;
  }
}
```

Overview – Where are we

- What are Protocol Buffers
- Structure of a .proto file
- How to use a message
- **How messages are encoded**
- Important points to remember
- More Stuff

Message encoding

- Full description at
  code.google.com/intl/apis/protocolbuffers/docs/encoding.html
- Messages are encoded in binary format, many key/value pairs
- Key = (id << 3) | wire_type
  - 0 = Varint (u/s/int32/64, bool, enum)
  - 1 = 64 bit (fixed64, sfixed64, double)
  - 2 = Length-delimited (string, bytes, messages, packed repeated fields)
  - 5 = 32 bit (fixed32, sfixed32, float)
- Little endian

Message encoding - **Varints**

- lower 7 bits per byte are used to store data ; if MSB is set, the next byte belongs to this value as well.
- Example: 1 → **0**000 0001
  300 (100101100) → **1**010 1100 **0**000 0010
- Example: `message Test1 { required int32 a = 1; }` and setting *a* to 150 (0x96) is encoded as `08 96 01`:
  - 08 = **0**000 1000, so wire type = 0 (varint) and id = 1
  - 96 01 = **1**001 0110 000 0001 → 1001 0110 → 150
- Generic/unsigned integer types use varint encoding

Message encoding - **ZigZag**

- int32 stores negative values in full length
- signed integer types (e.g. sint32) use ZigZag
- Mapping small positive AND negative values to small sizes:

  0 → 0
  - 1 → 1
  +1 → 2
  - 2 → 3
  2 → 4
  ...

- i.e. n → (n << 1) ^ (n >> 31)

Message encoding – The rest

- string, byte: varint-encoded length + raw data
- float, double: as-is (little endian)
- repeated fields:
  - packed=false: tag/id occurs multiple times
  - packed=true: tag + size + elements
- Unused fields are not part of the message
- strings

Overview – Where are we

- What are Protocol Buffers
- Structure of a .proto file
- How to use a message
- How messages are encoded
- **Important points to remember**
- More Stuff

Important points to remember

- Always remember that backward **and** forward **compatibility** is goal #1 with protobuf
- Be absolutely sure about a field's long-term necessity when using *required*
- Choose id numbers 1-15 for often used values (more efficiently encoded)
- Choose appropriate data types, based on expected values signed/unsigned/generic may result in better encoding

Updating a message

To update a message

- Define new fields as *repeated* or *optional* and set sensible *default* values (for backwards compatibility)
- Do **not** change tags/ids and do **not** recycle tags/ids (when e.g. removing optional fields in an update, make sure that the id will not be used again, preferably by prefixing the name of the obsolete field with e.g. OBSOLETE_)
- Some data type changes (e.g. between ints) possible
- When changing defaults, remember that default values are not encoded but always used as defined in .proto

More stuff

- Extensions
  - Define ranges of tags/ids that can be defined in another .proto file

```
message OneMessage {
  extensions 100 to max;
}

// Elsewhere...
extend OneMessage {
  optional Foo foo_ext = 100;
  optional Bar bar_ext = 101;
  optional Baz baz_ext = 102;
}
```

More stuff (cntd)

- Services
  - Possible to create stubs for RPC services using protobuf, e.g.

```
service SearchService {
 rpc Search (SearchRequest) returns (SearchResponse
);
}
```

- Self-describing messages, Reflection
- Custom options

Questions?