Protobuf : Protocol Buffers

ADSCCD : Administration et Développement Système pour les Centres de Calcul et de Donnée

M2-CNS, parcours SA et SR
Université Paris-Saclay, site d'Evry, UEVE

Patrice LUCAS
CEA-DAM, patrice.lucas@cea.fr
PAST, département informatique de l'UEVE

Protocol Buffers

- Développé par Google
  - “Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.” https://developers.google.com/protocol-buffers
- Langages principaux supportés par Google :
  - C++, C#, Dart, Go, Java, Python (mais aussi : Ruby, Objective C, JavaScript, PHP)
- De nombreuses autres implémentations sont disponibles : https://github.com/protocolbuffers/protobuf/blob/master/docs/third_party.md
  - Protobuf-c : https://github.com/protobuf-c/protobuf-c

Protocol Buffers

- Définition des types de messages par une syntaxe propre à protobuf
- Génération automatique du code de sérialisation / dé-sérialisation dans le langage souhaité à inclure dans les outils développés

Protocol Buffers : définir un message

- Fichier texte d'extension ".proto", avec rappel de la génération utilisé de protobuf
- Chaque type de message est défini par la liste de ses champs

```
syntax = "proto2";
message UnPost {
  Required int loginID = 1;
  Optional string pseudo = 2;
  Required bytes myData = 3;
  Optional double latitude = 4;
  Optional double longitude = 5;
  Optional string comment = 6;
}
```

- Chaque champ :
  - Un numéro unique
  - Un type
  - Une règle de présence

Protocol Buffers : numéro et type des champs

- Numéro unique :
  - 1 à 15 à réserver pour les éléments fréquemment utilisé car il faut un unique octet pour les encoder, 16 à 2047 2 octets, ...de 1 à (2^29 – 1)
- Type
  - Scalaire : double, float, int64, uint32, sint64, fixed32, string, bytes …
  - Enum : toujours la valeur 0 (le défaut), avec ou sans alias

```
syntax = “proto3”;
enum MsgType {
  option allow_alias = true;
  PUT = 0;
  SET = 0;
  GET = 1;
  DELETE = 2;
}
message Request {
  MsgType type = 1;
  string key = 2;
  string value = 3;
}
```

  - Inclure des types de message dans un type de message
- Une règle de présence

Protocol Buffers : type des champs

- Numéro unique
- Type
  - Scalaire : double, float, int64, uint32, sint64, fixed32, string, bytes …
  - Enum : toujours la valeur 0 (le défaut), avec ou sans alias
  - Inclure des types de message dans un type de message

```
syntax = “proto3”;
message Address {
  uint32 numero = 1;
  string rue = 2;
  string ville = 3;
  uint32 code_postal = 4;
}
message Who {
  string nom = 1;
  repeated string prenom = 2;
  Address adresse = 3;
}
```

- Une règle de présence

Protocol Buffers : règle de présence des champs

- Numéro unique
- Type
- Une règle de présence
  - Proto2 : required (Attention : pour toujours ...), optional, repeated
  - Proto3 : singular (par défaut), repeated
  - reserved : numero ou nom du champ

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

  - Oneof (à partir de libprotoc 2.6)

Protocol Buffers : règle de présence des champs

- Numéro unique
- Type
- Une règle de présence
  - Proto2 : required (Attention : pour toujours ...), optional, repeated
  - Proto3 : singular (par défaut), repeated
  - reserved : numero ou nom du champ
  - oneof (à partir de libprotoc 2.6) : plusieurs champs peuvent être positionnés dans le même message mais seul le dernier est pris en compte, cela ressemble par exemple dans l'esprit à une union en C.
    - "no repeated, optional or required inside", si vous avez par exemple besoin d'un champ "repeated" comme champ d'un oneof, alors il faut utiliser un type de message que vous définissez contenant un champ repeated.

```
syntax = "proto3";
enum Cafe {
    COURT = 0;
    ALLONGE = 1;
}
message menu_fromage_ou_dessert {
    string entree = 1;
    string plat = 2;
    oneof cloture {
        Cafe cafe = 3;
        string dessert = 4;
    }
}
```

Protocol Buffers : en C, génération

- Protoc-c :
  - “`protoc-c --c_out=OUT_DIR my_protobuf_file.proto`”
  - Un header `my_protobuf_file.pb-c.h` et un fichier source `my_protobuf_file.pb-c.c` sont générés.
- Pour l’édition de lien : -lprotobuf-c

Protocol Buffers : en C, structure de données générées

- Un type “struct” par type de message :
  - Même nom que le message protobuf défini
  - Champs de même nom que les champs protobuf définis
- Un type “enum” par type enum défini :
  - Même nom que le type enum en protobuf
  - Valeur : NOMDUENUM__NOMDEVALEUR

Protocol Buffers : en C, structure de données générées

- Repeated :
  - Pour un champ “`nom_du_champ`” de type `repeated` dans le message “`NomMessage`”, un champ `n_nom_du_champ` est ajouté dans la structure `NomMessage`.
  - Le champ `nom_du_champ` de la structure `NomMessage` devient un tableau (pointeur sur une allocation de `n_nom_du_champ` éléments).

Protocol Buffers : en C, structure de données générées

- repeated, exemple :
  - Dans le “`.proto`” :

```
message NomMessage {
    repeated uint32 nom_du_champ = 1;
}
```

  - Généré dans le “`.pb-c.h`” :

```
typedef struct _NomMessage NomMessage;

struct  _NomMessage
{
  ProtobufCMessage base;
  size_t n_nom_du_champ;
  uint32_t *nom_du_champ;
};
```

Protocol Buffers : en C, structure de données générées

- oneof :
  - Pour un champ `oneof` nommé “`type`” dans un message nommé “`Message`” un enum de type `Message__TypeCase` est créé.
  - Pour ce `oneof` qui contient les champs `champ1` et `champ2`, l’enum `Message__TypeCase` contient les constantes `MESSAGE__TYPE__NOT_SET` et `MESSAGE__TYPE_CHAMP1` et `MESSAGE__TYPE_CHAMP2`.
  - La structure `Message` contient un champ du type `Message__TypeCase` nommé `type_case`, suivi d’une `union` contenant les champs `champ1` et `champ2`.

Protocol Buffers : en C, structure de données générées

- oneof, exemple :
  - message.proto :

```
message Message {
    oneof type {
        uint32 champ1 = 1;
        uint32 champ2 = 2;
    }
}
```

  - message.pb-c.h :

```
typedef struct _Message Message;

typedef enum {
  MESSAGE__TYPE__NOT_SET = 0,
  MESSAGE__TYPE_CHAMP1 = 1,
  MESSAGE__TYPE_CHAMP2 = 2
    PROTOBUF_C__FORCE_ENUM_TO_BE_INT_SIZE(MESSAGE__TYPE)
} Message__TypeCase;

struct  _Message
{
  ProtobufCMessage base;
  Message__TypeCase type_case;
  union {
    uint32_t champ1;
    uint32_t champ2;
  };
};
```

Protocol Buffers : en C, packing

- Packing :
  - Préparer le message en initialisant correctement une variable de la structure correspondante,

```
void montype__init(Montype *message);
```

  - Allouer un buffer `uint8_t` de la bonne taille pour recevoir la donnée sérialisée

```
size_t montype__get_packed_size(const Montype *message);
```

  - Ecrire les données sérialisées dans le buffer

```
size_t montype__pack(const Montype *message, uint8_t
*out);
```

  - Il ne reste plus qu'à écrire le buf dans la socket à envoyer ...

Protocol Buffers : en C, unpacking

- Unpacking :
  - Préparer un buffer de réception d'une taille suffisante et lire la donnée sérialisée reçue
    - Deux possibilités pour la taille du buffer :
      - Une taille fixe toujours suffisante,
      - Un message uniquement pour transmettre la taille avant le message protobuf.
  - Obtenir les valeurs désérialisées :

```
Montype * montype__unpack(ProtobufCAllocator *allocator, size_t
len, const uint8_t *buffer);
```

    - allocator : peut-être NULL pour avoir l'allocateur par défaut
    - len : obtenue à la réception
    - buffer : stockant les données reçues
  - Libérer la structure désérialisée une fois utilisée

```
void montype__free_unpacked(Montype *message, ProtobufCAllocator
*allocator);
```

Protocol Buffers : en C, exemples

- protobuf_c_survey (proto2, a simple message with two fields including one enum)
- protobuf_c_shipment (proto3, oneof, repeated)

Protocol Buffers : en python

https://developers.google.com/protocol-buffers/docs/reference/python-generated

- Génération de code en python :

```
protoc --python_out=. mon_proto.proto
```

  - Avec une version de syntaxe “proto2” par exemple, un fichier “`mon_proto_pb2.py`” est généré et peut être importé dans le reste du code “`import mon_proto_pb2`”

Protocol Buffers : en python, structures générées

- Chaque type de message donne lieu à une classe dont on peut obtenir des objets.
  - Le code protobuf suivant au sein du fichier `mon_proto.proto`:

```
syntax = “proto2”;
message SousMessage {
  require uint32 id = 1;
}
message MonMessage {
  required string champ_scalaire = 1;
  required SousMessage champ_message = 2;
}
```

  - Permet de générer en python le fichier mon_proto_pb2 qui contient les déclarations pour générer un objet de la classe MonMessage possédant comme champ les champs du message défini

```
import mon_proto_pb2

mon_message = mon_proto_pb2.MonMessage()
```

Protocol Buffers : en python, structures générées

- On peut affecter des valeurs uniquement aux champs scalaires :

```
mon_message.champ_scalaire = “Alphonse”
```

- On ne peut affecter directement un message à un champ message
  - On peut soit utiliser CopyFrom() à partir d’un sous-message déjà existant pour affecter le champ sous-message correspondant.

```
sous_message = SousMessage()
sous_message.id = 23
mon_message.sous_message.CopyFrom(sous_message)
```

  - On peut également affecter directement des valeurs aux champs scalaire du sous-message, ce qui a pour effet de le créer.

```
Message = Message()
Message.sous_message.id = 77
```

Protocol Buffers : en python, enum

- Chaque enum, génère une constante par champ listé.
  - La déclaration suivante dans un fichier mon_proto.proto :

```
syntax = “proto2”;
enum Cafe {
  COURT = 0;
  ALLONGE = 1;
}
message MaCommande {
  required Cafe cafe = 1;
}
```

  - Permettra d’utiliser à travers le fichier généré mon_proto_pb2 la constante COURT et ALLONGE :

```
import mon_proto_pb2
ma_commande = mon_proto_pb2.MaCommande()
ma_commande.cafe = mon_proto_pb2.COURT
```

Protocol Buffers : en python, enum

- Pour un enum défini à l'intérieur d'un message:

```
syntax = "proto2";

message MaCommande {
  enum Cafe {
    COURT = 0;
    ALLONGE = 1;
  }
  required Cafe cafe = 1;
}
```

- La constante est définie comme un élément de la classe, MaCommande.COURT ou MaCommande.ALLONGE :

```
import mon_proto_pb2
ma_commande = mon_proto_pb2.MaCommande()
ma_commande.cafe = mon_proto_pb2.MaCommande.COURT
```

Protocol Buffers : en python, oneof

- Chaque champ d'un `oneof` est un champ existant de la classe générée pour le message.
  - Un seul champ parmi les champs du `oneof` peut être positionné. Le dernier positionné est toujours celui qui existe et il efface les précédents.
- Les différents champs possibles du `oneof` peuvent être chacun testés avec `HasField()`. Le champ `oneof` lui-même peut être testé avec la méthode `HasField()` .
- On peut également appelé `ClearField()` sur le champ `oneof`.
- Une méthode `WhichOneof` est ajoutée à la classe message et prend en paramètre le nom du champ `oneof` et renvoie le nom du champ positionné ou bien `None` si rien n'a été positionné.

Protocol Buffers : en python, oneof

- Pour le “.proto”:

```
message Bateau {
  oneof caracteristiques {
    message Multicoque multicoque = 1;
    message Deriveur deriveur = 2;
  }

  uint32 prix = 3;
}
```

- On utilise dans le code python:

```
bateau = Bateau()
multicoque = Multicoque()

bateau.multicoque = multicoque # Set the “multicoque” field
bateau.HasField(“multicoque”) # Test if “multicoque” field is set

# Test if any field of global “caracteristiques” field is set
bateau.HasField(“caracteristiques”)
# Clear any field from global “caracteriqtiques” field
bateau.ClearField(“caracteristiques”)

# To get None, “multicoque” or “deriveur”
bateau.WhichOneof(“caracteristiques”)
```

Protocol Buffers : en python, repeated

- Chaque champ repeated est géré comme une séquence que l’on ne peut pas positionner directement mais que l’on peut manipuler avec les fonctions de séquence.

Protocol Buffers : en python, repeated

- Repeated pour un scalaire
  - Dans le “.proto”:

```
message NomMessage {
    repeated uint32 nom_du_champ = 1;
}
```

  - Dans le code python:

```
msg = NomMessage()
msg.nom_du_champ.append(13) # Appends one value
msg.nom_du_champ.extend([45, 67]) # Appends an entire list
len(msg.nom_du_champ) # Gets the length of the sequence
msg.nom_du_champ[2] # Access one value
msg.nom_du_champ[1:2] # Access part of the sequence
msg.nom_du_champ[0] = 27 # Changing one value
msg.nom_du_champ[:] = [25, 17, 23, 55] # Change all the seq
```

Protocol Buffers : en python, repeated

- Repeated pour un message
  - Dans le “.proto”:

```
message SousMessage {
    Uint32 sous_champ = 1;
}

message NomMessage {
    repeated SousMessage nom_du_champ = 1;
}
```

  - Dans le code python:

```
msg = NomMessage()
sous_msg = SousMessage()
sous_msg.sous_champ = 72;
msg.nom_du_champ.append(sous_msg) # Append a copy of sous_msg

# Append a cpy of each elt of the seq
msg.nom_du_champ.append([sous_msg, sous_msg])

# Add a new elt
new_sous_msg = msg.nom_du_champ.add()
new_sous_msg.sous_champ = 23
```

Protocol Buffers : en python, sérialisation / désérialisation

- Sérialisation : fonction “`SerializeToString`” sur l’objet message qui produit un byte string

```
ma_commande = mon_proto_pb2.MaCommande()
ma_commande.cafe = mon_proto_pb2.COURT
byte_message_a_envoyer = ma_commande.SerializeToString()
```

- Désérialisation : fonction “ParseFromString” qui initialise l’objet en fonction du “byte string”

```
ma_commande = mon_proto_pb2.MaCommande()
ma_commande.ParseFromString(byte_message_recu)
```

Protocol Buffers : en python, exemple

- protobuf_python_survey

Protocol Buffers : Exercice

- Escape mars
- Pizza