

Programmation Système Répartie et concurrente

Master 1 Informatique – MU4IN400

Cours 9 : Protocoles de Communication

Yann Thierry-Mieg

Yann.Thierry-Mieg@lip6.fr

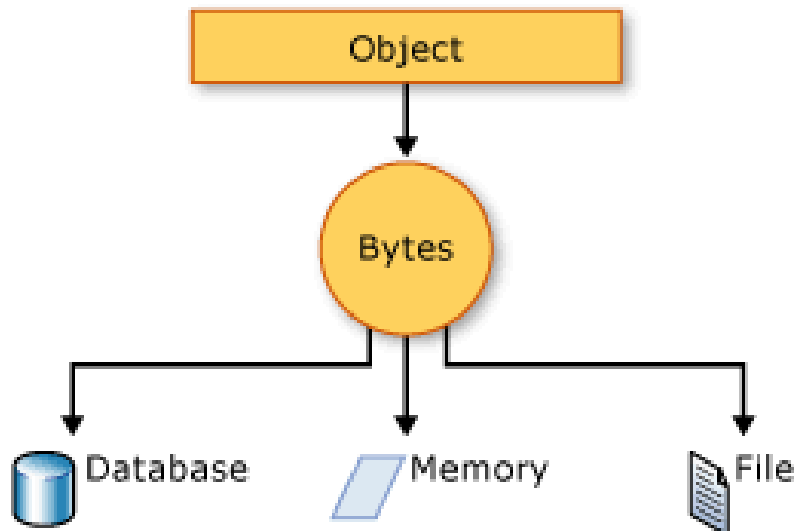
Plan

- On a vu au cours précédent :
 - Les Socket, un standard IEEE pour la communication
- Aujourd'hui : Construire un protocole sur des Sockets
 - Le design pattern « Proxy Distant »
 - Sérialisation de données
 - Protobuf
- Références :
 - « Computer Systems : A programmers Perspective. » Bryant, O'Hallaron
 - « Design Patterns », le GOF Gamma, Helm, Vlissides, Johnson
 - Google protobuf : <https://developers.google.com/protocol-buffers/>
 - Slides assemblées de plusieurs sources, citées dans les slides concernées

La sérialisation des données

Sérialisation

- ▶ Processus permettant de transformer un objet donné dans un format qui peut être stocké dans un buffer
 - ▶ Sauvegarde et persistance (fichier)
 - ▶ Transport dans un réseau
- ▶ Objectif : reconstruire l'objet en question dans l'environnement du récepteur
- ▶ Processus inverse : désérialisation



Sérialisation par défaut (exemple Java)

- ▶ Pas mal de langages (Java, C#...) proposent une sérialisation par défaut vers un format binaire
 - ▶ En C ou C++, copier dans un buffer et memcpy ?
- ▶ Mais
 - ▶ Formats langage spécifique
 - ▶ Sérialisation Java nécessite JVM + la classe (interopérabilité limitée)
 - ▶ Formats peu efficaces
 - ▶ Données supplémentaires stockées par objet
 - ▶ Opérations de (dé)sérialisation coûteuses
 - ▶ Problèmes de versions des classes
 - ▶ Utilisation en production non recommandée
 - ▶ e.g Bloch, Effective Java, items 74 et 78

Sérialisation binaire

▶ Avantages :

- ▶ Efficace en mémoire
- ▶ Rapide à construire et à parser

▶ Désavantages :

- ▶ Illisible par les humains
- ▶ Dépendant de la plateforme
- ▶ Peu extensible, versions...
- ▶ Mécanisme « unsafe » en production

Formats Textuels : CSV

- ▶ Comma Separated Values

Nom, Age, Tel

Joe, 22, 06789012

Bob, 26, 07689210

- ▶ Avantages

- ▶ Simplicité, portabilité, interactions tableurs, ligne de commande

- ▶ Désavantages

- ▶ Peu compact (encodage ascii)
- ▶ Fichiers « plats », sans structure
- ▶ Pas de validation

Stockage au format XML

```
<person>  
  <name>Bob</name>  
  <age>26</age>  
  <contacts>  
    <email>my@email.com</email>  
    <phone>999</phone>  
  </contacts>  
</person>
```

► Avantages :

- Lisibilité, portabilité, généralité
- Structure hiérarchique
- Validation vis-à-vis d'un schéma (DTD, XSD, RNG...)
- Standard bien accepté qui conserve une place importante

► Désavantages

- Verbosité excessive, taille de stockage énorme => lenteur du parse, et de la construction
- Stockage des noms de tous les champs
- Pas de typage fort (int vs float ?)
- Complexité des schémas et de l'accès aux données, de la validation

Stockage au format JSON

- ▶ Issu de « JavaScript Object Notation »
 - ▶ A depuis largement dépassé ce cadre
- ▶ Concepts simples,
 - ▶ paires clé valeur
 - ▶ Listes, imbriquées OK
- ▶ Bon support et adoption dans le web
 - ▶ Alternative « light » à XML
 - ▶ Reste assez peu compact
 - ▶ Mais lisible et éditée par des humains
- ▶ Globalement une bonne alternative à XML dans les mêmes cas d'utilisation
 - ▶ SOAP, REST, JSON est un stack populaire

```
Person {  
  name: "Bob"  
  age:26  
  contacts: {  
    email:my@email.com  
    phone:999  
  }  
}
```

Google Protobuf

- ▶ « LE » format de sérialisation et de stockage de Google

- ▶ Utilisé à travers toutes ses application, adopté par d'autres Twitter...

- ▶ Un standard émergent

(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.)

- ▶ Alternatives existent Apache Thrift, Apache Avro (e.g Facebook)...

- ▶ En pratique : un format de sérialisation (Wire)

- ▶ Défini à partir d'un fichier de description lisible : langage de définition d'interface .proto

- ▶ Compact, fortement typé, performant

- ▶ 3 à 10 fois plus compact que XML, 20 à 100 fois plus rapide

- ▶ Portable, Interopérable, multi langage

- ▶ Supportant l'évolution des versions du schéma des données

Fichier .proto

- ▶ Définit un format de « Message »
 - ▶ i.e. une donnée structurée, proche d'une classe OO
- ▶ Syntaxe simple et lisible
 - ▶ Champs fortement typés
 - ▶ Propriété des champs : **optional, required, repeated**
 - ▶ Ajout d'un indice pour chaque champ
 - ▶ Import et raffinement (héritage) de types supportés

```
syntax = "proto3";  
package bayes.bob;  
  
message UserProfile {  
    // User's email.  
    string email = 1;  
  
    // User's year of birth.  
    uint32 year_of_birth = 2;  
}
```

Exemple

```
import "google/protobuf/timestamp.proto";
import "bob_emploi/frontend/api/profile.proto";
import "bob_emploi/frontend/api/project.proto";

message User {
    // User's profile.
    UserProfile profile = 1;

    // List of current projects.
    repeated Project projects = 2;

    // The date & time we last sent an email to the user.
    google.protobuf.Timestamp last_email_sent_at = 3;
}
```

Utilisation : protoc

- ▶ On compile le fichier proto vers un langage cible
 - ▶ Génère une API facile d'emploi pour la (dé)sérialisation

Compilation

```
protoc -I . bob_emploi/frontend/api/*.proto --python_out=.
```



Une API facile d'emploi

.proto

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

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

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

  repeated PhoneNumber phone = 4;
}
```

C++

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
//Write
person.SerializeToOstream(&output);

//Read
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Syntaxe Protobuf

Définition d'un Message

- ▶ Point d'entrée du fichier Protobuf .proto
- ▶ Syntaxe : **Message [NomMessage] { ... }**
- ▶ Les définitions peuvent être imbriquées
- ▶ Conversion de *chaque* Message vers une classe e.g. C++

Contenu du Message

- ▶ **Chaque Message peut contenir**

- ▶ Des Messages

- ▶ Des Enums :

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

- ▶ Des Champs (field), chacun ayant la syntaxe

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

Champ « rule »

▶ **Required**

- ▶ **Présent exactement une fois, accès par `msg.champ()`**
 - ▶ **E.g. `person.name()`**
- ▶ **Attention à « required » on ne peut pas en ascendant compatible se débarrasser du modificateur ou du champ**

▶ **Optional**

- ▶ **Présent une fois ou absent**
- ▶ **Test de l'existence : `msg.has_champ()`**
 - ▶ **E.g. `person.has_email()`**

▶ **Repeated**

- ▶ **Champs « répétés » de 0 à N fois non borné**
- ▶ **Liste *ordonnée***
- ▶ **Interrogation de la taille `msg.champ_size()`, itération**
 - ▶ **E.g. `person.phone_size()`**

Typage

.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

Protobuf 3 ajoute un type Map :

```
map<string, Project> projects = 3;
```

Identifiant de Champ

- ▶ Chaque champ à un identifiant unique au sein de sa définition de Message
 - ▶ Les indices ne sont pas réutilisables, même si on efface des champs (versions)
 - ▶ Les indices courts (≤ 15) sont plus compacts, à privilégier
- ▶ L'identifiant sert à l'encodage des données
 - ▶ On ne répète pas le nom du champ
 - ▶ Seulement son indice sur le moins de bits possibles

Options, namespaces, import

- ▶ Les Options :
 - ▶ [default = value] : donne une valeur par défaut. Les champs qui ont leur valeur par défaut ne sont pas encodés dans la représentation
 - ▶ [packed = false/true] : force un encodage plus compact des « repeated ». Sous proto3 le défaut est déjà true.
 - ▶ [deprecated = false/true] : marque un champ comme obsolète
 - ▶ [optimize_for = SPEED/CODE/LITE_RUNTIME] : contrôler les choix de génération de code
 - ▶ SPEED par défaut, engendre une spécialisation de chaque opération
 - ▶ CODE : plus lent, mais une base de code moindre
 - ▶ LITE_RUNTIME : moins d'opérations sur Message, mais plus léger (e.g. tel portable)
- ▶ Les packages engendrent un namespace C++
 - ▶ package pr
- ▶ L'importation de Messages décrits ailleurs est supportée
 - ▶ import « pr/myproto.proto »

Utilisation pratique de Protobuf

Exemple C++ 11

Le site Protobuf : <https://developers.google.com/protocol-buffers/>
Le Tutoriel C++ : <https://developers.google.com/protocol-buffers/docs/cpp-tutorial>
La référence : <https://developers.google.com/protocol-buffers/docs/reference/cpp-generated>

Fichier addressbook.proto

```
syntax = "proto3";
```

```
package tutorial;
```

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

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

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

```
  repeated PhoneNumber phones = 4;  
} // fin Person
```

```
message AddressBook {  
  repeated Person people = 1;  
}
```

Compilation

- ▶ Commande protoc (pour tous les langages)

```
protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/addressbook.proto
```

- ▶ Flag « --cpp_out » pour C++
- ▶ Génère deux fichiers
 - ▶ addressbook.pb.h : la déclaration
 - ▶ addressbook.pb.cc : le corps d'implémentation
- ▶ Il ne reste plus qu'à include de .h pour avoir accès à l'API

Code Généré

opérations de la classe Person

// name

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

// id

```
inline bool has_id() const;  
inline void clear_id();  
inline int32_t id() const;  
inline void set_id(int32_t value);
```

// email

```
inline bool has_email() const;  
inline void clear_email();  
inline const ::std::string& email() const;  
inline void set_email(const ::std::string& value);  
inline void set_email(const char* value);  
inline ::std::string* mutable_email();
```

API attributs répétés

- On accède par indice

```
// phones
inline int phones_size() const;
inline void clear_phones();
inline const ::google::protobuf::
    RepeatedPtrField<::tutorial::Person_PhoneNumber >&
        phones() const;
inline ::google::protobuf::
    RepeatedPtrField< ::tutorial::Person_PhoneNumber >*
        mutable_phones();
inline const ::tutorial::Person_PhoneNumber& phones(int index) const;
inline ::tutorial::Person_PhoneNumber* mutable_phones(int index);
inline ::tutorial::Person_PhoneNumber* add_phones();
```

API : enum, sous-messages

- ▶ Pour les enum => enum C++
 - ▶ Person::PhoneType : enum
 - ▶ Person::MOBILE, Person::HOME... valeurs
- ▶ Pour les messages imbriqués => classes imbriquées
 - ▶ Accès via Person::PhoneNumber ou via Person_PhoneNumber
- ▶ On dispose d'une API réflexive :
 - ▶ static const Descriptor* descriptor()
 - ▶ Permet d'interroger les champs et leur description
- ▶ Globalement on a ce qu'on attend :
 - ▶ Des classes, avec des attributs, des getter/setter simples.

API : opérations du Message

- ▶ `bool IsInitialized() const`:: la validation : vérifie que tous les champs “required” sont bien présents.
- ▶ `string DebugString() const`:: utile pour debugger
- ▶ `void CopyFrom(const Person& from)`:: écrase le contenu du message par le contenu fourni. On a aussi les opérateurs du C++.
- ▶ `void Clear()`:: ramène à un état vide, aucun champ positionné

API : sérialiser

- ▶ **bool SerializeToString(string* output) const;**
 - ▶ sérialise le message et le stocke (au format binaire) dans une `std::string`.
- ▶ **bool ParseFromString(const string& data);**
 - ▶ réciproque, construit un objet depuis une string binaire
- ▶ **bool SerializeToOstream(ostream* output) const;**
 - ▶ écrit l'objet dans le flux de sortie fourni
- ▶ **bool ParseFromIstream(istream* input);**
 - ▶ parse un message à partir du flux d'entrée fourni

+Variantes : text, JSON...

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

Lecture et Ecriture avec un filedescriptor nu ZeroCopy[I/O]Stream

// Write some data to "myfile". First we write a 4-byte "magic number"
// to identify the file type, then write a length-delimited string. The
// string is composed of a varint giving the length followed by the raw
// bytes.

```
int fd = open("myfile", O_CREAT | O_WRONLY);  
ZeroCopyOutputStream* raw_output = new FileOutputStream(fd);  
CodedOutputStream* coded_output = new CodedOutputStream(raw_output);
```

```
int magic_number = 1234;  
char text[] = "Hello world!";  
coded_output->WriteLittleEndian32(magic_number);  
coded_output->WriteVarint32(strlen(text));  
coded_output->WriteRaw(text, strlen(text));  
delete coded_output;  
delete raw_output;  
close(fd);
```

Lire depuis un fichier

```
#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;
    }
}
```



Encodage : Wire



Encodage Wire

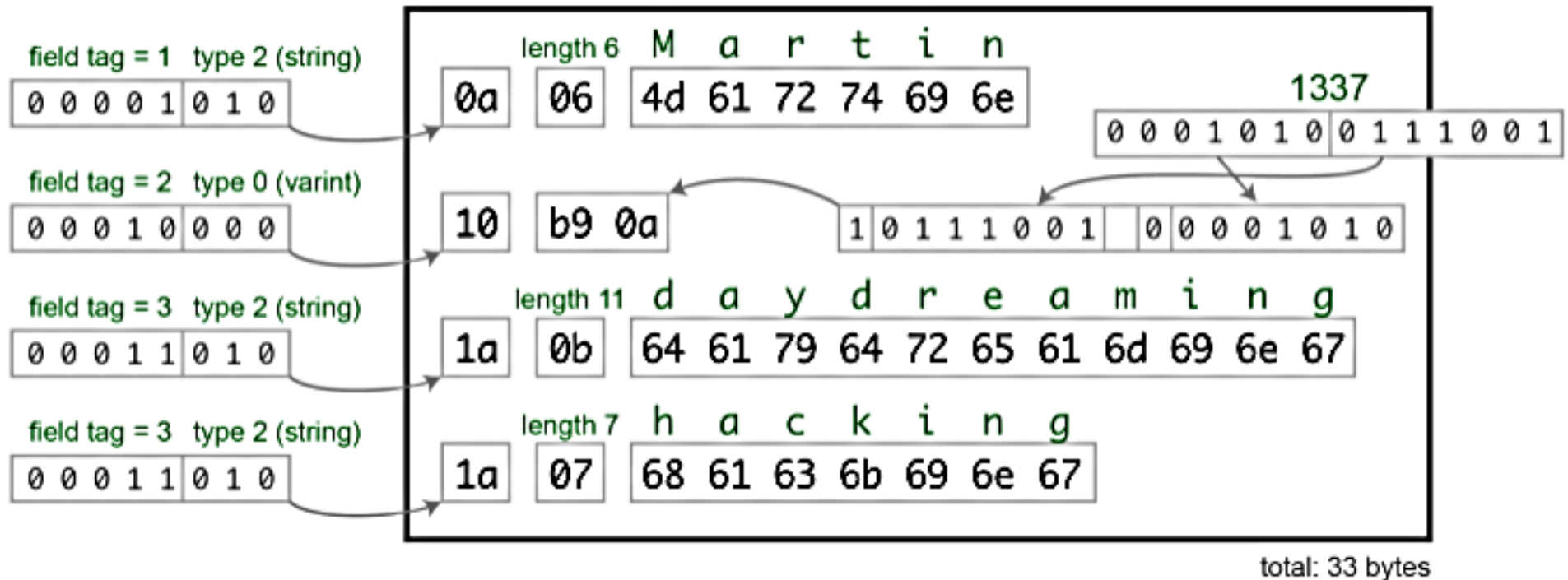
Person.proto

```
message Person {  
    required string user_name      = 1;  
    optional int64  favourite_number = 2;  
    repeated string interests       = 3;  
}
```

Person.json

```
{  
    "userName": "Martin",  
    "favouriteNumber": 1337,  
    "interests": ["daydreaming", "hacking"]  
}
```

Protocol Buffers



Message encoding

- Full description at code.google.com/intl/apis/protocolbuffers/docs/encoding.html
- Messages are encoded in binary format, many key/value pairs
- $\text{Key} = (\text{id} \ll 3) \mid \text{wire_type}$
 - 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

24/33

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 → 0000 0001
 300 (100101100) → 1010 1100 0000 0010
- Example: `message Test1 { required int32 a = 1; }`
and setting *a* to 150 (0x96) is encoded as 08 96 01:
 - 08 = 0000 1000, so wire type = 0 (varint) and id = 1
 - 96 01 = 1001 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. $n \rightarrow (n \ll 1) \wedge (n \gg 31)$

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



Conclusion

ProtoBuf et Wire




Comparison


	Parsing efficiency	Reusable	Model Update	Hierarchical	Small Size
CSV					
XML					
JSON					
PB					

Un format Compact pour Sérialiser et Désérialiser

- ▶ **Finale**ment simplement une chaîne technologique moderne et efficace
 - ▶ Portabilité nombreux langages
 - ▶ Mise à jour gestion de versions
 - ▶ Efficace en temps et en mémoire
 - ▶ Gestion de beaucoup de petits messages (<1 MB)
 - ▶ Passerelles vers JSON et autres formats
- ▶ **Protobuf** à travers son IDL apporte d'autres avantages
 - ▶ Documentation du format des données (notion de schéma)
- ▶ Avec cette API produisant une sérialisation compacte
 - ▶ Introduction des gRPC, une API de service distant



Design Pattern Proxy Distant



DP Proxy

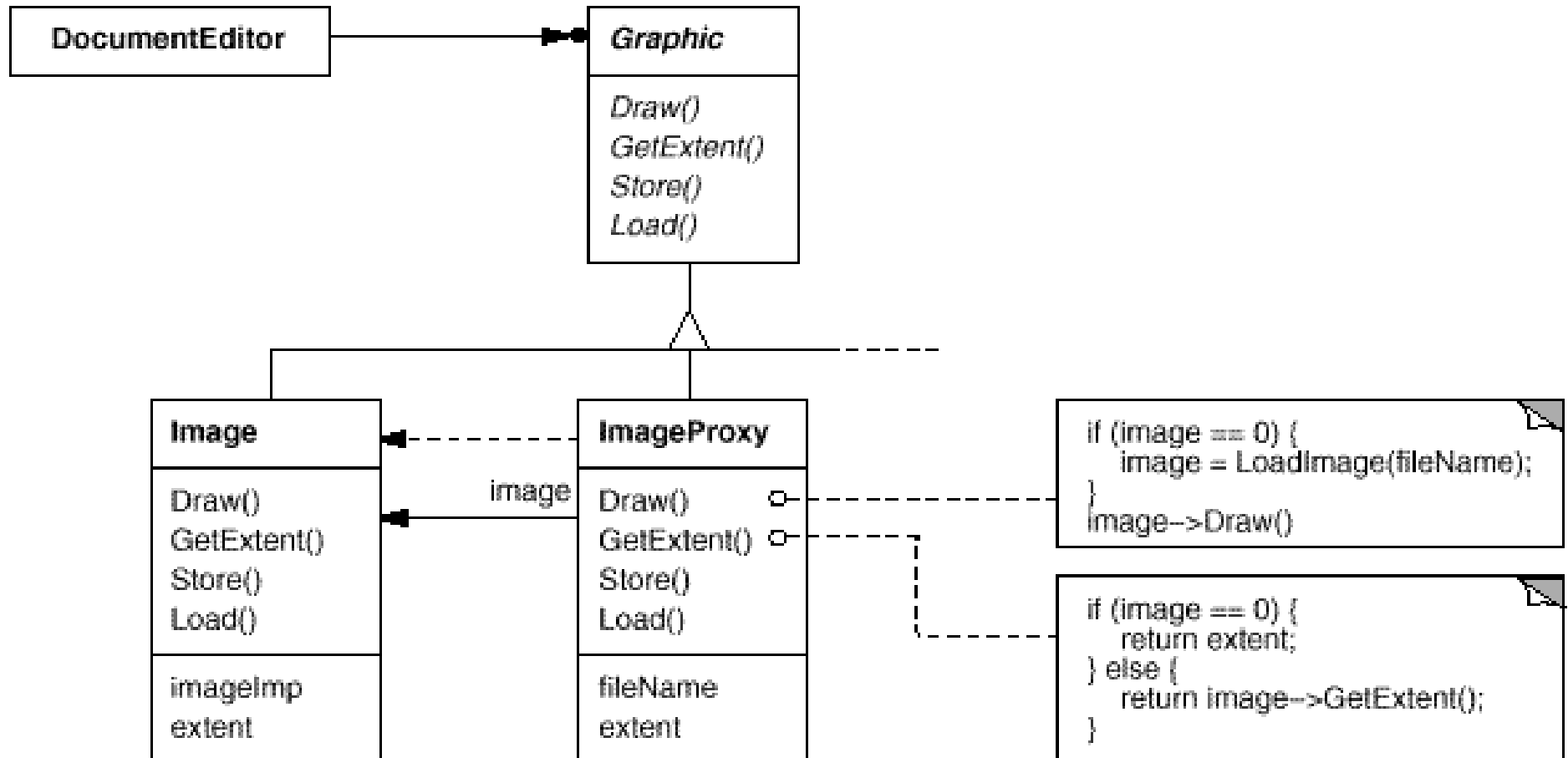
- ▶ Proxy : objet qui fait semblant d'être un autre objet
- ▶ Par exemple le proxy réseau : de votre browser se comporte comme un gateway internet (e.g. comme une box) mais rajoute des traitements (filtres, cache...)
- ▶ Pour le DP, le proxy est une classe qui implémente les mêmes opérations que l'objet qu'elle protège/contrôle.
- ▶ Plusieurs variantes de Proxy, selon la finalité :
 - ▶ Proxy Virtuel : retarder les allocations/calculs couteux
 - ▶ Proxy de Sécurité : filtre/contrôle les accès à un objet
 - ▶ Proxy Distant : objet local qui se comporte comme l'objet distant et cache le réseau
 - ▶ Smart Reference : proxy qui compte les références (C, C++)

Proxy Virtuel :

retarder les opérations coûteuses

- ▶ Soit un éditeur de texte type Word
- ▶ Le document est rempli d'images « lourdes » (plusieurs megas) stockées dans des fichiers séparés
- ▶ Quand on ouvre un document, il faut calculer le nombre de pages du document, la mise en page => connaître la taille des images (`Image.getExtent()`)
- ▶ Quand on affiche une page donnée du document, il faut faire le rendu des images présentes sur la page (`Image.draw()`)
- ▶ Comment retarder le chargement des images quand on ouvre le document ??

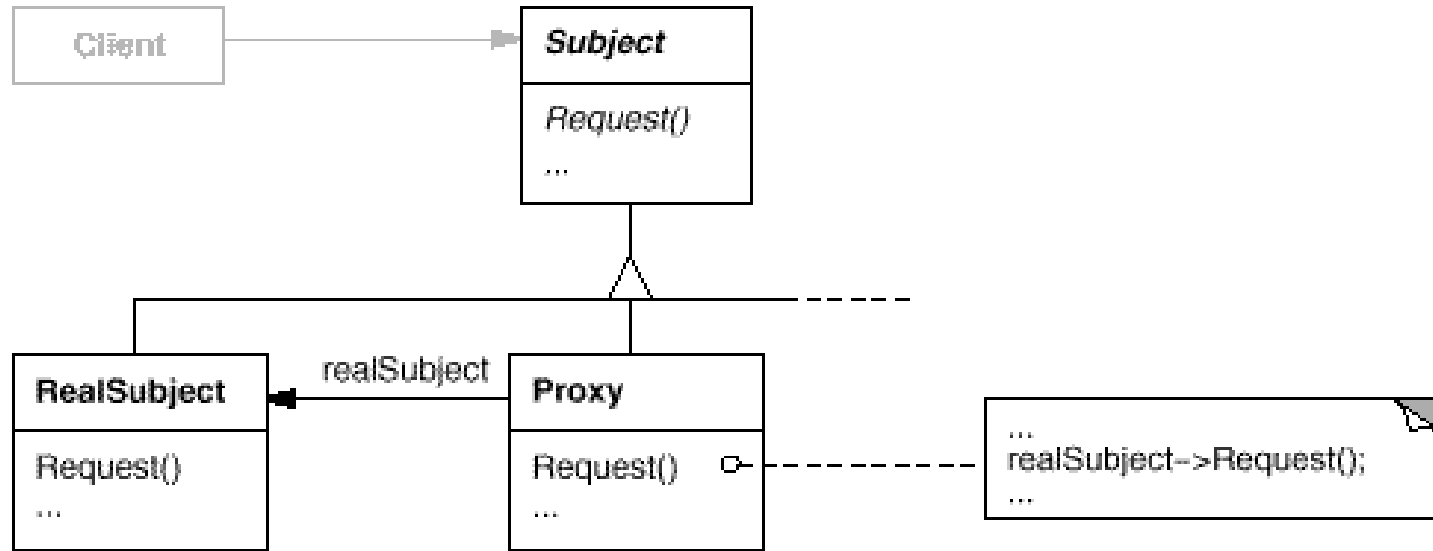
Proxy Virtual : Exemple



Principes du Proxy virtuel

- ▶ On construit initialement des ImageProxy pour chaque image
 - ▶ Par exemple le document crée les images via une ImageFactory
- ▶ Ces objets stockent et donc connaissent la taille de l'image (getExtent)
- ▶ Ce n'est qu'au moment où on affiche la page avec l'image (première invocation de draw sur le proxy) que l'image va être chargée (à la volée)
- ▶ Conclusion : le document s'ouvre rapidement, mécanisme transparent vis-à-vis de la classe Document.
 - ▶ Un proxy ne peut pas être distingué de l'objet réel par le client !

DP Proxy : Structure



- ▶ **Subject** : interface manipulée par le client ; il ne distingue pas le proxy de l'objet réel
- ▶ **RealSubject** : un objet réel sur lequel on (le client) souhaite au final faire l'interrogation
- ▶ **Proxy** : Fait semblant d'être le vrai sujet, et connaît (bien) ce dernier, en pratique intercepte les invocations au sujet => filtre, cache, comptage, etc...
- ▶ Délégation particulière, où délégat (**Proxy**) et délégué (**RealSubject**) réalisent la même interface / Proche d'un DP adapter ou contrat et existant satisfont la même interface.

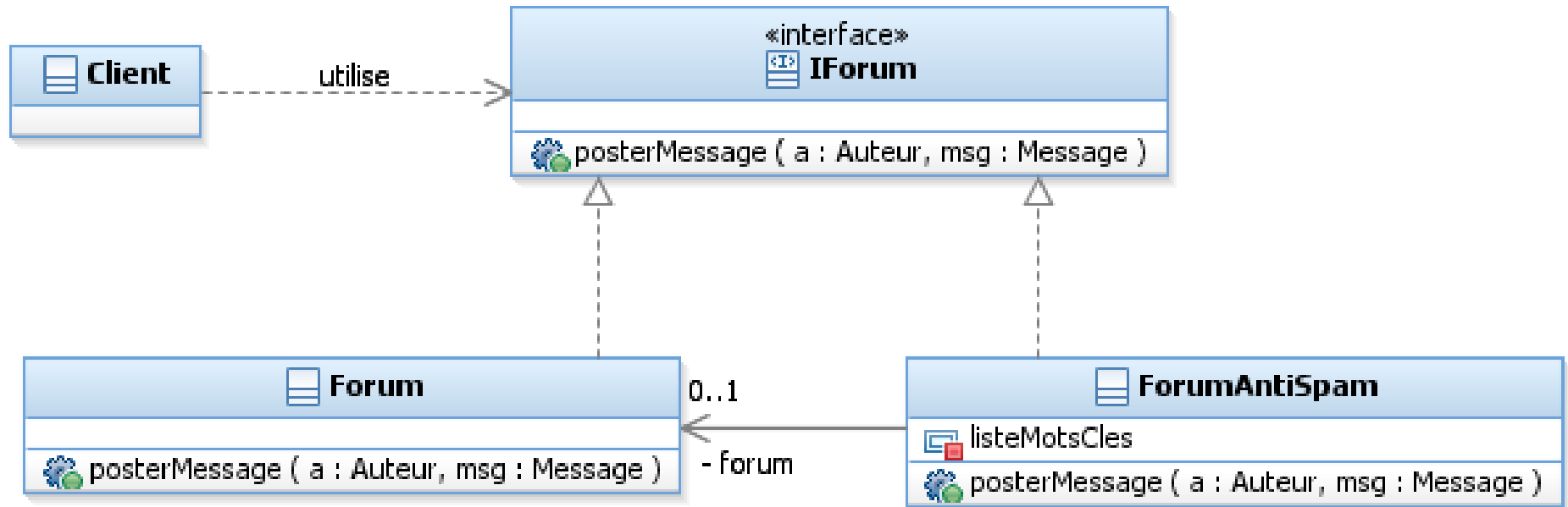
DP Proxy : Smart Reference

- ▶ Spécifique aux langages qui ne sont pas munis de gc
- ▶ Problème de décider quand désallouer (free) les objets ?
- ▶ Solution : compter les références
 - ▶ Création du proxy en lui passant un objet => compteur de refs à 1
 - ▶ Copie du proxy => incrémenter le compteur
 - ▶ Destruction de Proxy => décrémenter le compteur, si on atteint 0, désallouer l'objet concret
- ▶ Implémenté en C++ (std 2011) par `shared_ptr`

Proxy de Sécurité

- ▶ Permet de protéger ou contrôler les accès à un objet
- ▶ Exemple Forum:
 - ▶ Classe Forum : munie d'une opération posterUnMessage (Auteur a, Message m)
 - ▶ La classe Forum existe, il ne s'agit pas de la modifier
- ▶ Comment bloquer les messages contenant des mots clés interdits ? (grossièretés, langage SMS,...)

DP Proxy de Sécurité : Forum



```
for (String mot : listeMotsCles)
    if (msg.contains(mot))
        return;

forum.posterMessage(a,msg);
```

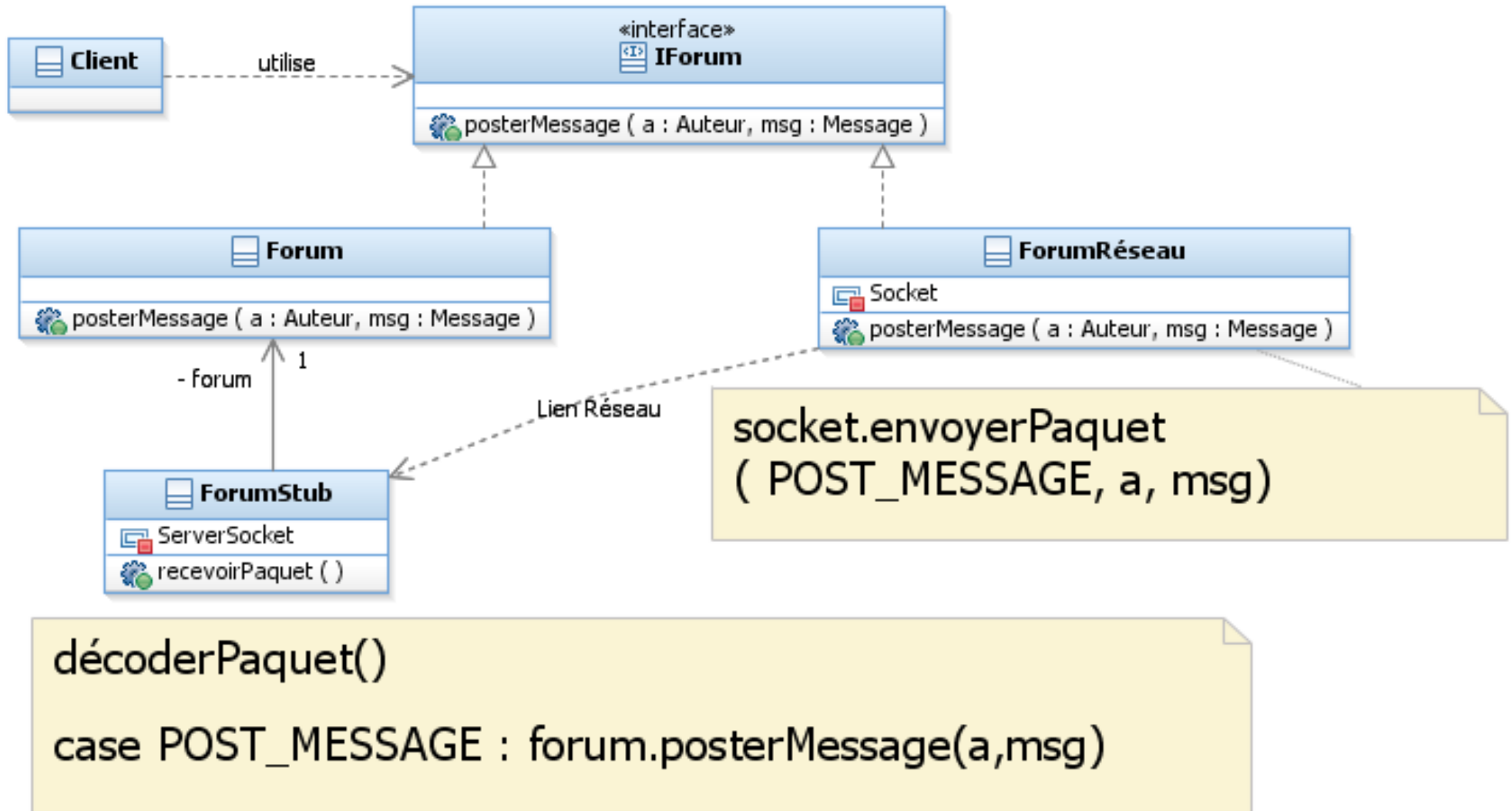
Proxy de sécurité : conclusions

- ▶ **Orthogonal au traitement protégé**
 - ▶ La sécurité est une couche supplémentaire, distincte du traitement de base
 - ▶ Transparent vis-à-vis du client et de l'implémentation (RealSubject)
 - ▶ Parfois Decorator peut jouer le même rôle.

DP Proxy Distant : Principes

- ▶ On a une application répartie sur plusieurs machines
- ▶ On voudrait développer l'application sans trop se soucier de où sont physiquement stockés les objets
- ▶ Proxy réseau : objet local à la machine, qui se comporte comme l'objet distant, mais répercute ses opérations sur l'objet distant via le réseau
 - ▶ Comportement par délégation, mais avec le réseau interposé

DP Proxy distant



Proxy distant: conclusions

- ▶ Généralise la notion de RPC (remote procedure call)
- ▶ Rends transparent la localisation de objets (on s'adresse à un *service de nommage* pour obtenir une ref à l'objet)
- ▶ La réalisation du Proxy réseau et du stub suit une ligne standard
- ▶ De nombreux frameworks offrent de générer cette glu automatiquement (et/ou de la cacher)
 - ▶ Java RMI : remote method invocation
 - ▶ Google RPC, en appui sur ProtoBuf

Google RPC gRPC

Documentation sur : <https://grpc.io/>

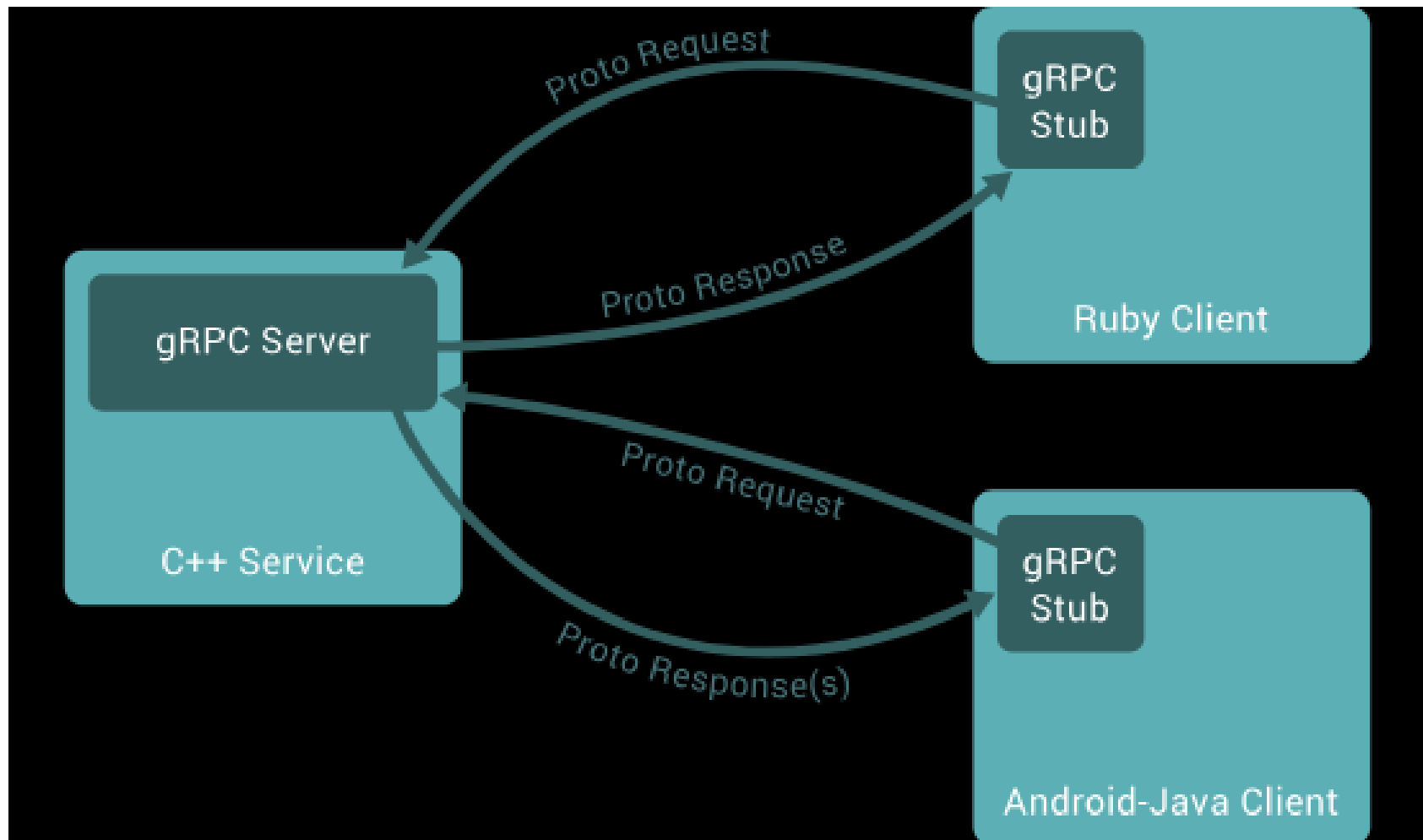
Principes

- ▶ On enrichit la définition des .proto avec des Services

```
service HelloService {  
    rpc SayHello  
        (HelloRequest) returns (HelloResponse);  
}  
message HelloRequest {  
    string greeting = 1;  
}  
message HelloResponse {  
    string reply = 1;  
}
```

Les Services engendrent des Stub/Client

- ▶ Offre le service via le réseau
 - ▶ Inter-opérable grâce à Wire/PB



Service RPC ou Streaming

- ▶ RPC simple : une requête, on attend de façon synchrone la réponse

```
rpc SayHello(HelloRequest) returns (HelloResponse){ }
```

- ▶ Mode Stream : un flux de Message dans un sens ou dans l'autre

- ▶ On attend la lecture complète de la réponse

```
rpc ManyHello(stream HelloRequest) returns (HelloResponse){ }
```

```
rpc ManyResponse(HelloRequest) returns (stream HelloResponse){ }
```

- ▶ Mode Bidirectionnel + Stream

- ▶ Echanges plus ou moins arbitraires de séquences de Message

```
rpc BiDiHello(stream HelloRequest) returns (stream HelloResponse){ }
```

Côté Client

- ▶ Création d'un channel de connexion

```
grpc::CreateChannel("localhost:50051", grpc::InsecureChannelCredentials());
```

- ▶ Utilisation du « Stub » généré par gRPC

public:

```
RouteGuideClient(std::shared_ptr<ChannelInterface> channel,  
                 const std::string& db)  
    : stub_(RouteGuide::NewStub(channel)) { ... }
```

- ▶ Invocation de Service distant (RPC)

```
bool GetOneFeature(const Point& point, Feature* feature) {  
    ClientContext context;  
    Status status = stub_->GetFeature(&context, point, feature);  
}
```

Mise en place : Serveur

- La génération de code produit un serveur prêt à l'emploi

```
void RunServer(const std::string& db_path) {  
    std::string server_address("0.0.0.0:50051");  
    RouteGuideImpl service(db_path);  
  
    ServerBuilder builder;  
    builder.AddListeningPort(server_address,  
                             grpc::InsecureServerCredentials());  
    builder.RegisterService(&service);  
    std::unique_ptr<Server> server(builder.BuildAndStart());  
    std::cout << "Server listening on " << server_address << std::endl;  
    server->Wait();  
}
```