Medii de proiectare și programare

2017-2018
Curs 7

Conținut

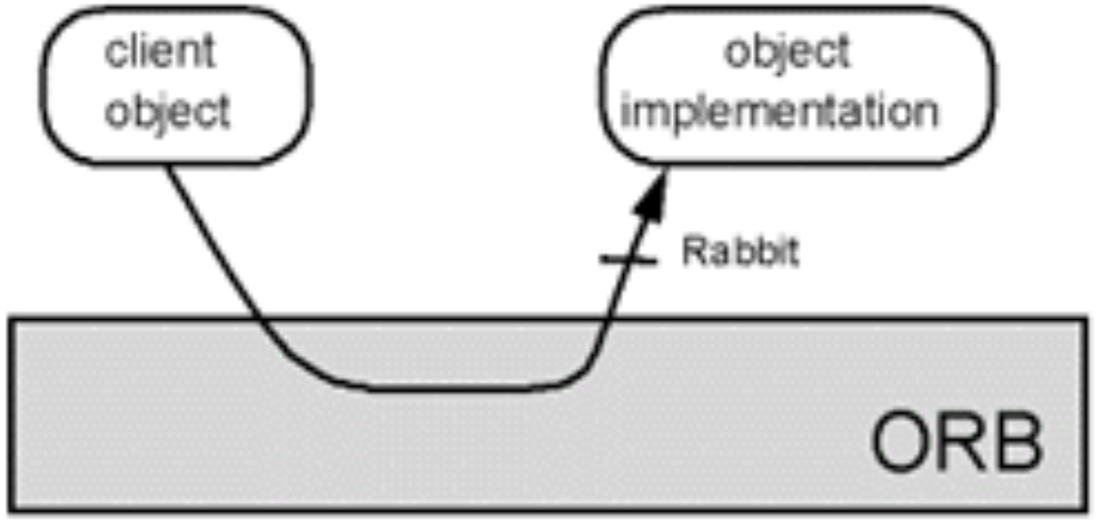
- Aplicații distribuite cross-platform
- CORBA
- Google Protobuf, gRPC
- Apache Thrift

CORBA

- CORBA (Common Object Request Broker Architecture) este o arhitectură standard pentru sisteme distribuite orientate pe obiect.
- Permite unei colecții de obiecte distribuite *heterogene* să interacționeze.
- A fost propusă de OMG (Object Management Group).
- Produsele CORBA oferă un framework pentru dezvoltarea și execuția aplicațiilor distribuite.
- CORBA definește arhitectura obiectelor distribuite.
- Paradigma de bază CORBA este de cerere a unor servicii oferite de un obiect distribuit (remote).
- Serviciile oferite de un obiect sunt specificate de interfața acestuia.
- Interfețele sunt definite folosind limbajul Interface Definition Language (IDL) definit de OMG.
- Obiectele distribuite (remote) sunt identificate prin referințe, de tipul interfețelor IDL.

Arhitectura CORBA

- Un client păstrează o referință către un obiect distribuit (remote).
- Un proces numit Object Request Broker (ORB), trimite cererea obiectului și returnează rezultatul primit clientului.




```
interface Rabbit {
        :
    AnotherObject jump(
        in long how_high);
        :
};
```

ORB

- ORB este un serviciu distribuit care transmite cererile obiectelor remote.
- Localizează obiectul remote în rețea, comunică cererea primită de la client, așteaptă rezultatul și când acesta este disponibil comunică rezultatul obținut clientului.
- ORB promovează transparența locației. Același mecanism de cerere este folosit de client și obiectul remote CORBA indiferent de locația acestuia.
- ORB transmite cererea indiferent de limbajul de programare folosit pentru construirea acesteia.
- Clientul care transmite cererea poate fi dezvoltat într-un limbaj de programare diferit de obiectul remote care va răspunde cererii.
- Serviciul ORB realizează transformările necesare între limbaje.
- Sunt definite transformări pentru limbaje de programare cunoscute: Java, C++, etc.
- CORBA definește un protocol de rețea, numit IIOP (Internet Inter-ORB Protocol), care permite clienților care folosesc un obiect remote CORBA să comunice cu alte obiecte remote CORBA. IIOP are la bază implementarea TCP/IP.

Interfețe IDL

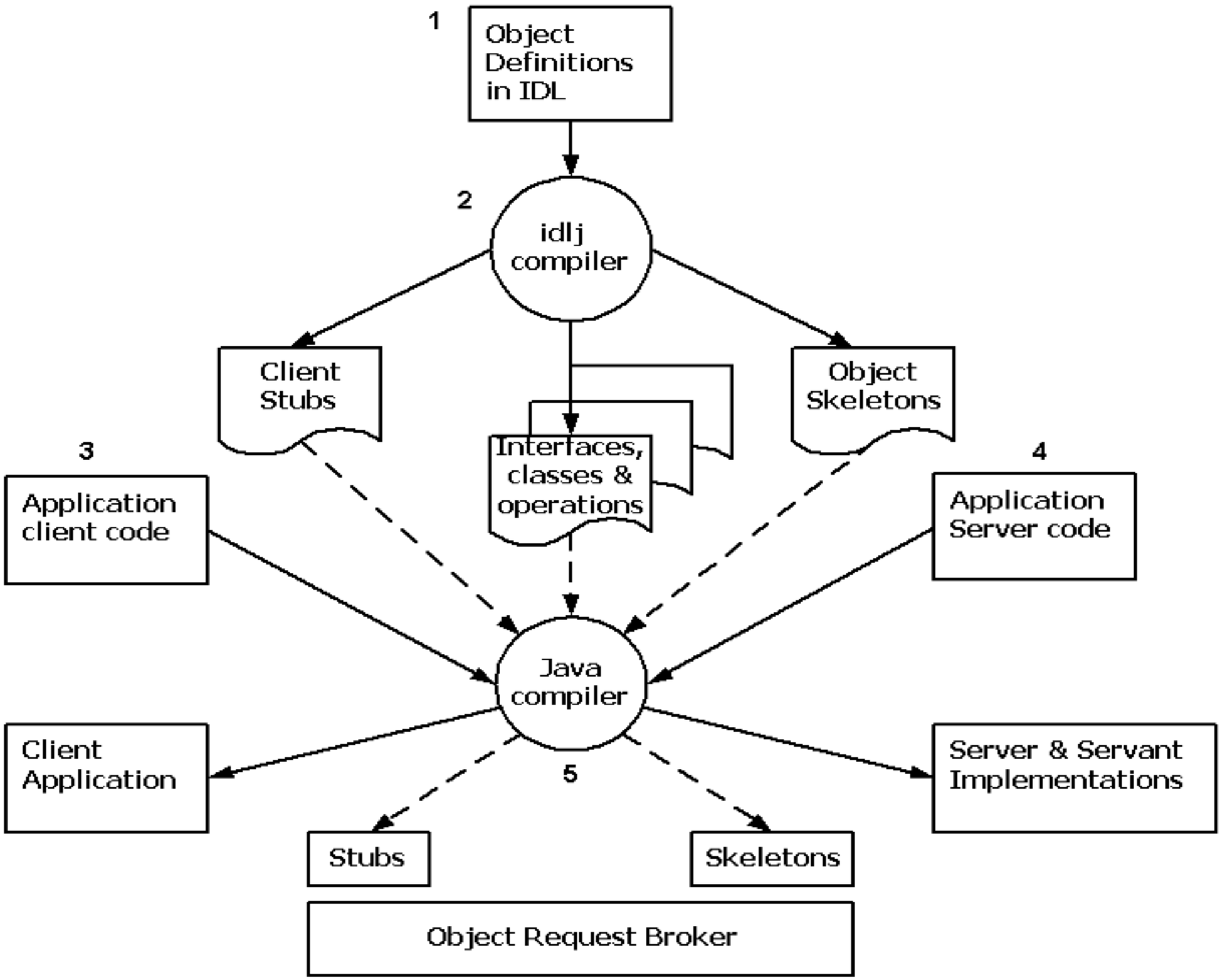
- Limbajul *Interface Definition Language* (IDL) propus de OMG permite specificarea interfețelor obiectelor remote.
- Interfața unui obiect remote indică metodele ce pot fi apelate la distanță, dar nu cum sunt implementate aceste metode.
- În IDL nu se poate declara starea unui obiect sau algoritmi.
- Implementarea unui obiect CORBA este furnizată într-un limbaj de programare (Java, C++, etc). O interfață specifică contractul dintre codul care folosește obiectul și codul care implementează obiectul. Clienții depind doar de interfață.
- Interfețele IDL nu depind de un anumit limbaj de programare. IDL definește transformări (eng. language bindings) pentru diferite limbaje de programare.
- Se permite astfel ca pentru obiectul remote să fie ales cel mai potrivit limbaj de programare, și, de asemenea, permite clienților să aleagă cel mai potrivit limbaj (care poate diferi de cel folosit pentru implementarea obiectului remote).
- Transformări pentru C, C++, Java, Ada, Smalltalk, etc.

Exemplu IDL

```
module BookShop {
  struct Book {
      double price;
      String title;
      String author
  };
  exception Unknown{};
  interface Library {
      Book getBook(in string title) raises(Unknown);
      readonly attribute string name;


  };
  interface BookFactory {
    Book create_book( in string title, in string author, in double
   price);
  };
};
```

CORBA și Java



Pașii dezvoltării unui sistem cu CORBA

1. Scrierea specificației fiecărui obiect remote folosind IDL.
2. Folosirea compilatorului IDL pentru a genera codul stub client și server.
3. Scierea codului obiectului remote, respectiv a clientului.
4. Compilarea codului client și server.
5. Pornirea serverului.
6. Execuția aplicației client.

Protocol Buffers (Protobuf)

- Reprezintă o modalitate independentă de limbaj și platformă de a serializa structuri de date folosite în protocole de comunicație, stocarea datelor, etc.
- Sunt flexible, eficiente și au un mecanism automat de serializare a datelor structurate (similar cu XML), dar mecanismul este mai rapid, mai simplu și mai ușor de folosit.
- Se definește o singura dată modul de structurare a datelor (folosind un IDL), iar apoi se folosește un compilator special (protoc) care generează cod ce permite scrierea/citirea structurilor de date în/din o varietate de streamuri de date și folosind diferite limbaje de programare.
- Este permisă modificarea structurii datelor fără a provoca apariția erorilor în programele dezvoltate folosind vechea structură de date.
- Specificarea structurii datelor se definește folosind tipuri de mesaje protocol buffers și sunt salvate în fișiere `.proto`.
- Fiecare mesaj protocol buffer este o înregistrare logică mică de informații, conținând o serie de perechi cheie-valoare.

Protocol Buffers vs XML

- Protocol buffers au avantaje asupra XML în privința serializării datelor:
  - sunt mai simple
  - sunt de 3 până la de 10 ori mai mici ca și dimensiune
  - sunt de 20 până la de 100 de ori mai rapide
  - sunt mai puțin neclare (ambigue)
  - clasele generate automat pentru accesarea datelor sunt mai ușor de folosit în limbajul de programare
- Protocol buffers nu sunt totdeauna o soluție mai bună decât XML:
  - Protocol buffers nu sunt potrivite pentru modelarea unui text ce folosește markup (ex. HTML), deoarece nu permite ușor imbricarea structurii datelor cu text.
  - XML este ușor de citit de oameni (eng. *human-readable*) și de editat (eng. *human-editable*);
  - Protocol buffers nu pot fi citite de oameni și editate
  - XML se descrie pe sine.
  - Un protocol buffer are sens doar dacă avem acces și la definiția mesajului (fișierul .proto).

Protocol Buffers vs XML

//XML

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

//Reprezentarea textuala a unui Protobuf (nu octeții serializați)

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

Protocol Buffers vs XML

//Parsarea unui XML

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
  </person>
//Java
 person.getElementsByTagName("name").item(0).getNodeValue()
 person.getElementsByTagName("email").item(0).getNodeValue();
```

//Parsarea unui mesaj Protobuf

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
System.out.println("Name: " + person.getName());
System.out.println("E-mail: “+ person.getEmail());
```

Protocol Buffers – Exemplu

```
syntax="proto2"
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;
}
```

Definirea unui nou tip de mesaj

- Tipurile mesajelor sunt salvate într-un fișier **.proto**

```
syntax="proto3";
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- Fiecare câmp are un nume, tip și un tag asociat.
- Tipul poate fi:
  - scalar (double, float, int32, int64, bool, string, bytes, etc).
  - tip compus (enumerare)
  - tipul unui alt mesaj

Definirea unei enumerări

- Enumerare: un câmp de tip enumerare poate avea ca și valoare doar una dintre constantele specificate.
- Fiecare enumerare trebuie să definească o constantă care are asociată valoarea zero ca primă constantă din enumerare.
- Aceasta valoare va fi folosită ca și valoarea implicită.
- Două constante pot avea aceeași valoare (dacă se setează opțiunea allow_alias).

```
enum Corpus {
     option allow_alias = true;
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 5;
  }
  Corpus corpus = 4;
}
```

Reguli pentru specificarea câmpurilor

- Un câmp poate fi:
  - *singular*: într-un mesaj bine format acest câmp poate să apară cel mult o dată (zero sau unu).
  - **`repeated`**: acest câmp poate să apară de ori câte ori (inclusiv zero) într-un mesaj bine format. Ordinea valorilor care se repetă se va păstra.
- Dacă un mesaj nu conține nici o valoare pentru un câmp singular, la parsare câmpul va primi valoarea implicită corespunzătoare tipului său.

```
message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

Atribuirea tagurilor

- Fiecare câmp din definiția unui mesaj are un tag numeric unic asociat.
- Aceste taguri sunt folosite pentru identificarea câmpurilor în formatul binar și nu ar trebui schimbate după începerea folosirii lui.
- Tagurile cu valori între 1 și 15 ocupă un octet (incluzând numărul de identificare și tipul câmpului).
- Tagurile cu valori între 16 - 2047 ocupă 2 octeți.
- Tagurile cu valori între 1 și 15 ar trebui folosite pentru câmpurile folosite cele mai des din mesaj.
- Tagurile pot avea valori între 1 și 536,870,911.
- Numerele între 19000 și 19999 sunt rezervate pentru implementarea Protocol Buffers - compilatorul protoc va genera o eroare dacă se încearcă folosirea lor.

Tipuri multiple de mesaje

- Mai multe tipuri de mesaje pot fi definite într-un singur fișier .proto.

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}


message SearchResponse {
 ...
}
```

Câmpuri rezervate

- Dacă definiția tipului unui mesaj se modifică ulterior prin ștergerea unui câmp sau comentarea unui câmp utilizatorii pot refolosi tag-ul asociat câmpului respectiv.
- Pot apărea erori când se folosesc mesaje salvate cu versiuni mai vechi ale aceluiași tip de mesaj.
- Se poate specifica că anumite taguri sau nume de câmpuri sunt rezervate și nu mai pot fi refolosite ulterior, folosind instrucțiunea **`reserved`**.
- În aceeași instrucțiune **`reserved`** nu pot fi folosite taguri și nume de câmpuri.

```
message Foo {
    int32 foo = 2;  // câmpul foo

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

Fișiere .proto multiple

- Într-un fișier .proto se pot folosi definițiile din alte fișiere .proto prin importarea lor.
- Importarea definițiilor din alt fișier **`.proto`** se face cu instrucțiunea import la începutul fișierului (după instrucțiunea syntax):

```
import “myproject/other_protos.proto“;
```

- Compilatorul caută fișierele importate în lista de directoare specificată la linia de comandă ca și parametru al compilatorului folosind opțiunea **`-I sau --proto_path`**.
- Dacă nu a fost specificată această opțiune, compilatorul va căuta in directorul în care a fost rulat compilatorul.

Tipuri Nested

- Se pot defini tipuri de mesaje în interiorul altor tipuri de mesaje:

```
message SearchResponse {
  message Result {
     string url = 1;
     string title = 2;
     repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

- Tipul nested se poate folosi din exterior respectând formatul `Parent.Type:`

```
message SomeOtherMessage {
   SearchResponse.Result result = 1;
}
```

Tipul Any

- Tipul Any este folosit pentru a defini câmpuri pentru care nu știm/avem fișierul .proto corespunzător.
- Un câmp de tip Any va conține un mesaj necunoscut serializat într-un șir de octeți și un URL folosit ca și identificator global unic care va indica către definiția tipului mesajului.
- Pentru a folosi tipul Any trebuie importat fișierul *google/protobuf/any.proto*.

```
import "google/protobuf/any.proto";

message ErrorStatus {
  string message = 1;
  repeated google.protobuf.Any details = 2;
}
```

URL implicit asociat unui mesaj definit într-un fișier .proto are formatul

```
type.googleapis.com/packagename.messagename
```

Opțiunea oneof

- Este folosită când valoarea unui câmp poate fi cel mult una dintr-o mulțime finită de valori de tipuri diferite. Folosirea opțiunii reduce dimensiunea mesajului.
- Câmpurile Oneof sunt câmpuri normale, exceptând faptul că partajează aceeiași zonă de memorie și cel mult un câmp poate avea asociată o valoare la un moment dat.
- Setarea valorii unuia dintre câmpurile marcate *oneof* duce automat la ștergerea valorii celorlalte câmpuri.
- Se poate determina care câmp din mulțimea specificată oneof are asociată o valoare (dacă există un astfel de câmp). (Dependent de limbaj)
- Nu se pot folosi câmpuri repeated în mulțimea câmpurilor specificate cu oneof.

```
message SampleMessage {
  oneof test_oneof {
    string name = 4;
    SubMessage sub_message = 9;
  }
}
```

Tipul Map

- Se pot defini dicționare ca și câmp al unui nou tip de mesaj.

```
map<key_type, value_type> map_field = N;
```

- **`key_type`** poate fi orice tip *integral* (orice tip scalar exceptând tipurile reale și octeți) sau tipul string. **`value_type`** poate fi orice tip.
- Observații:
  - Câmpurile Map nu pot fi *repeated*.
  - Ordinea serializarii valorilor dintr-un dicționar este nespecificată, nu ne putem baza pe o anumită ordine a elementelor din dicționar.
  - Dacă la parsare/deserializare există mai multe chei având aceeaiași valoare se păstrează ultima valoare întâlnită.

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

Pachete

- Opțional se poate specifica pachetul corespunzător tipurilor de mesaje definite într-un fișier .proto pentru a evita coliziuni de nume.

```
package foo.bar;
message Open { ... }
```

- Numele pachetului poate fi utilizat ulterior la definirea tipului câmpurilor:

```
message Foo {
  ...
  foo.bar.Open open = 1;
  ...
}
```

- Modul în care specificarea unui pachet afectează codul generat depinde de limbaj:
  - În Java, pachetul este folosit ca și pachet Java, dacă nu se specifică explicit un alt pachet folosind opțiunea `option java_package` în fișierul .proto.
  - În C#, pachetul este folosit ca și spațiu de nume după transformarea la PascalCase, dacă nu se specifică explicit un alt pachet folosind opțiunea `csharp_namespace` în fișierul .proto. (Exemplu, câmpul Open va face parte din spațiul de nume Foo.Bar).

Definirea serviciilor RPC

- Se pot defini servicii RPC într-un fișier .proto, iar compilatorul va genera codul corespunzător interfeței și stub-urilor (proxies) în limbajul ales.

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

- gRPC - tehnologie open-source cross-platform RPC dezvoltată de Google care folosește protocol buffers și permite generarea codul RPC direct din fișierul .proto dacă se folosește un plugin adițional pentru compilatorul protoc.
  - Există și alte tehnologii RPC bazate pe Protocol Buffers care pot fi utilizate pentru RPC

Alte opțiuni

- Într-un fișier .proto se pot declara opțiuni individuale care nu modifică semnificație tipurilor de mesaje definite în fișier, dar influențează modul în care este generat codul sursă corespunzător unui anumit limbaj.
- Lista completă a opțiunilor disponibile: google/protobuf/descriptor.proto.
- Cele mai folosite opțiuni:
  - **`java_package`** (nivel fișier): Numele pachetului care va fi folosit la generarea claselor Java. Dacă nu se specifică explicit, numele implicit al pachetului va fi cel specificat folosind opțiunea package.

```
option java_package = "com.example.foo";
```

  - **`java_outer_classname`** (nivel fișier): Numele clasei folosite ca și clasă exterioară pentru tipurile mesajelor. Dacă nu se specifică explicit opțiunea **`java_outer_classname`**, numele clasei se va construi prin transformarea numelui fișierului .proto la camel-case (**`foo_bar.proto`** va deveni **`FooBar.java`**).

```
option java_outer_classname = "FooBar";
```

Alte opțiuni

- **`java_multiple_files`** (nivel fișier): Determină generarea tipurilor mesajelor, a enumerărilor și a serviciilor la nivel de pachet (nu într-o clasă exterioară).

```
option java_multiple_files = true;
```

- Opțiuni C#

```
option (google.protobuf.csharp_file_options).namespace =
"MyCompany.MyProject";
option (google.protobuf.csharp_file_options).umbrella_classname =
"ProjectProtos";
```

Generarea claselor Java/C#

- Generarea claselor Java, C#, Python, C++, etc. se face folosind compilatorul protoc asupra fișierului .proto.

```
protoc --proto_path=IMPORT_PATH --java_out=DST_DIR  path/to/
   file(s).proto
```

```
protoc --proto_path=IMPORT_PATH --csharp_out=DST_DIR  path/to/
   file(s).proto
```

- **`IMPORT_PATH`** specifică directorul în care compilatorul va căuta fișierele .proto când încearcă să rezolve opțiunile *import.*
  - Dacă nu este specificat se folosește directorul curent.
  - Se poate folosi opțiunea **`-I=IMPORT_PATH`** ca și o formă prescurtată a opțiunii **`--proto_path.`**

Clase generate

- Pentru fiecare tip de mesaj M definit într-un fișier .proto, compilatorul va genera clasa M.
- Fiecare clasa are un Builder asociat prin intermediul căruia se vor crea instanțe ale clasei respective.
- Clasa M cât și Builder-ul asociat au metode de tip get generate automat de compilatorul *protoc.* Clasa Builder are și metode de tip set care permit setarea valorilor.
- Clasele corespunzătoare mesajelor generate de compilator sunt *immutable*.
- După ce un obiect a fost construit, el nu mai poate fi modificat.
- Pentru a construi un mesaj, întâi se construiește builder-ul, se setează valorile câmpurilor folosind builder-ul, iar apoi se apelează metoda build() pentru a obține mesajul.
- Fiecare metodă din builder returnează un builder (obiectul this) care permite legarea mai multor apeluri de metode set într-o singură linie de cod.

Clase generate

- Fiecare mesaj și builder conțin metode care permit verificarea și transformarea mesajului:
  - `toByteArray(): byte[]` serializează mesajul și returnează un șir de octeți conținând mesajul serializat.
  - `parseFrom(byte[] data): Person` parsează un mesaj dintr-un șir de octeți.
  - `writeTo(OutputStream output)` serializează mesajul și îl scrie într-un OutputStream.
  - `writeDelimitedTo(OutputStream output)` serializează mesajul și scrie dimensiunea lui și mesajul serializat într-un OutputStream.
  - `parseFrom(InputStream input): Person` citește și parsează un mesaj dintr-un InputStream.
  - `parseDelimitedFrom(InputStream input): Person` citește și parsează un mesaj dintr-un InputStream (mesajul a fost scris folosind writeDelimitedTo).

Exemplu mini-chat (proto2)

```
syntax="proto2";
package chat.protocol;
option java_package = "chat.protocol.protobuf";
option java_outer_classname = "ChatProtobufs";
message User{
    required string id=1;
    optional string passwd=2;
}
message Message{
    required string receiverId=1;
    required string senderId=2;
    required string text=3;
}
```

```
message ChatRequest {
  enum Type { Login = 1; Logout = 2; SendMessage = 3; GetLoggedFriends=4 ;}
  // Identifies which request is filled in.
  required Type type = 1;
  // One of the following will be filled in, depending on the type.
  optional User user = 2;
  optional Message message = 3;
}
message ChatResponse{
   enum Type { Ok = 1; Error = 2; GetLoggedFriends=3; FriendLoggedIn = 4; FriendLoggedOut=5; NewMessage=6;  }
     // Identifies which request is filled in.
     required Type type = 1;
    // One of the following will be filled in, depending on the type.
     optional string error = 2;
     repeated User friends=3;
     optional User user=4 ;
     optional Message message = 5;
}
```

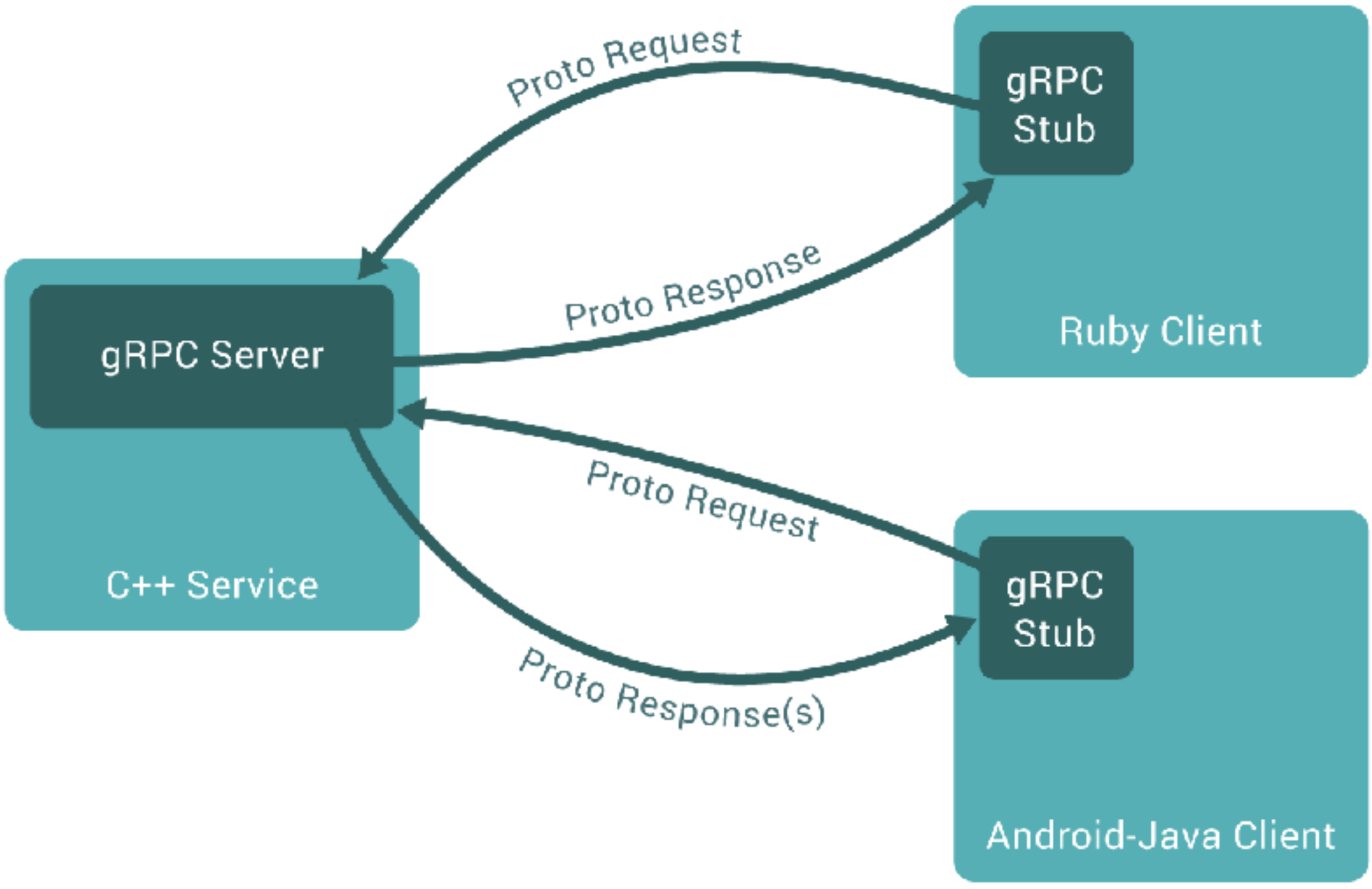
Exemplu mini-chat (proto3)

```
syntax="proto3";
package chat.protocol;
option java_package = "chat.network.protobuffprotocol";
option java_outer_classname = "ChatProtobufs";
message User{
     string id=1;
     string passwd=2;
}
message Message{
     string receiverId=1;
     string senderId=2;
     string text=3;
}
message ChatRequest {
  enum Type {Unkown=0; Login = 1; Logout = 2; SendMessage = 3; GetLoggedFriends=4 ;}
  // Identifies which request is filled in.
   Type type = 1;
  // One of the following will be filled in, depending on the type.
   oneof payload{
       User user = 2;
       Message message = 3;
  }
}
message ChatResponse{
   enum Type { Unknown=0; Ok = 1; Error = 2; GetLoggedFriends=3; FriendLoggedIn = 4; FriendLoggedOut=5; NewMessage=6;}
     // Identifies which request is filled in.
      Type type = 1;
     // One of the following will be filled in, depending on the type.
      string error = 2;
      repeated User friends=3;
      User user=4 ;
      Message message = 5;
}
```

gRPC

- Cu gRPC o aplicație client poate apela metode la distanță ca si cum ar fi metode ale unui obiect local.
- Facilitează crearea aplicațiilor distribuite și a serviciilor.
- gRPC folosește idea definirii unui serviciu (interfețe) care specifică metodele ce pot fi apelate la distanță, împreuna cu parametrii și tipul returnat al acestora.
- În aplicația server există un obiect remote care implementează interfața, iar serverul gRPC gestionează cererile clienților.
- Clientul (aplicația client) are un stub (proxy) care oferă aceleași metode ca și obiectul remote.
- Clienții și serverele gRPC pot fi scrise și pot rula în diferite limbaje de programare: Java, C#, C++, Python, Go, etc.

gRPC


Proto Request
gRPC Stub
Proto Response
gRPC Server
Ruby Client
C++ Service
Proto Request
gRPC Stub
Proto Response(s)
Android-Java Client

gRPC - Definirea unui serviciu

- Implicit, gRPC folosește Protocol Buffers ca și limbaj de definire a tipurilor de mesaje și a serviciilor (IDL).
- Se pot folosi alte IDL-uri.

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

gRPC - Definirea unui serviciu

- gRPC permite specificarea a 4 tipuri de apeluri de metode la distanță:
  - *unare* - clientul trimite o singura cerere la server și primește un singur răspuns de la server (apeluri de funcții normale).

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

  - *server streaming* - clientul trimite o cerere la server și primește un stream cu ajutorul căruia poate citi o secvență de răspunsuri de la server. Clientul citește răspunsurile până când nu mai sunt mesaje pe stream.

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

  - *client streaming* - clientul scrie o secvență de mesaje și le trimite la server folosind streamul furnizat. După ce clientul a terminat scrierea mesajelor, așteaptă ca serverul să le citească și să trimită un singur răspuns.

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

gRPC - Definirea unui serviciu

- gRPC permite specificarea a 4 tipuri de apeluri de metode la distanță:
  - *Bidirectional streaming* - atât serverul cât și clientul trimit o secvență de mesaje folosind un stream bidirecțional (read-write). Streamurile (read-write) funcționează independent; clienții și serverul pot citi și scrie în ordinea dorită de ei.

    Exemple:

    -serverul poate aștepta până primește toate mesajele de la client înainte de a trimite răspunsurile;

    -serverul poate citi un mesaj, trimite răspunsul, sau altă combinație de read-write.

    Ordinea mesajelor din fiecare stream se păstrează.

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

gRPC - Definirea unui serviciu

- Pornind de la definiția unui serviciu dintr-un fișier .proto, gRPC furnizează plugin-uri pentru compilatorul protoc care permit generarea codului corespunzător clienților și serverului.
- Clienții și serverul gRPC folosesc acest API pentru implementarea funcționalității dorite:
  - În aplicația server, serverul implementează metodele declarate de serviciu, și le face disponibile clienților folosind un server gRPC. Frameworkul gRPC decodifică cererile, execută metodele remote și codifică răspunsurile.
  - În aplicația client, clientul are un obiect local (stub, proxy) care implementează aceleași metode ca și serviciul. Client poate apela aceste metoda ca și cum ar fi apeluri locale, folosind parametrii corespunzători.

gRPC - Timeout

- gRPC permite clienților să specifice cât timp sunt dispuși să aștepte ca un apel la distanță să își încheie execuția. Dacă durata specificată a fost depășită, apelul la distanță se va încheia cu eroarea DEADLINE_EXCEEDED.
- Pe server se poate verifica dacă un anumit apel la distanță a depășit durata sau cât timp mai are la dispoziție.

- Atât clientul cât și serverul pot anula execuția unui apel la distanță în orice moment. Anularea duce la oprirea imediată a execuției apelului la distanță.
- Anularea nu este o operație de tip “undo”: modificările făcute înainte de anulare nu vor fi anulate și ele.

gRPC - Terminare, canale de comunicare

- Terminarea unui apel - clientul și serverul determină independent succesul execuției unui apel la distanță, iar concluziile lor pot să difere.
  - Exemplu 1: un apel RPC se poate termina cu succes pe server (“I have sent all my responses!”), dar poate eșua pe client (“The responses arrived after my deadline!”).
  - Exemplu 2: serverul poate decide terminarea înainte ca clientul să fi terminat de trimis toate cererile.

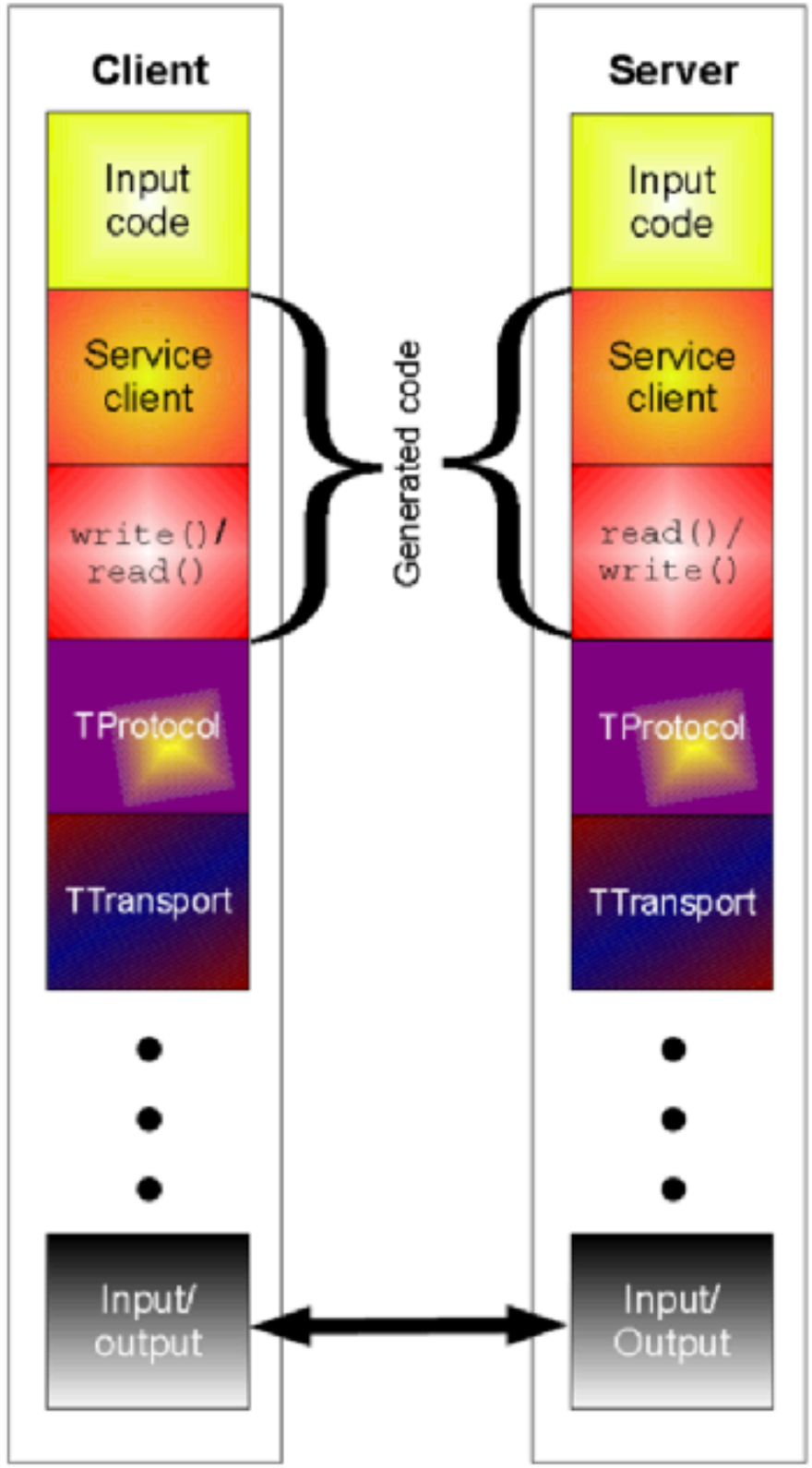
- Canal de comunicare gRPC - furnizează o conexiune la un server gRPC specificând adresa și portul și se folosește la instanțierea unui proxy (client stub). Clienții pot specifica parametrii pentru a modifica comportamentul implicit (setarea opțiunii de compresie a mesajelor, etc).
- Un canal are asociată o stare (dacă este conectat, dacă este idle, etc.)

gRPC - Exemplu

- O aplicație simplă client/server:
  - Obiectul remote are o metodă care transformă un text în text cu litere mari și adaugă data și ora la care a fost primit textul.
  - Clientul obține o referință la obiectul remote, apelează metoda și tipărește rezultatul primit.

Apache Thrift

- Thrift este un tehnologie cross-platform pentru RPC.
- Thrift oferă separare clară între nivelul transport, serializarea datelor și logica aplicației.
- Thrift a fost dezvoltat inițial de Facebook, acum este un proiect open-source găzduit de Apache.
- Apache Thrift conține un set de unelte de generare de cod care permit utilizatorilor să dezvolte clienți și servere RPC doar prin definirea tipurilor de date necesare și a interfețelor serviciilor într-un fișier IDL .thrift.
- Folosind acest fișier ca și date de intrare, se generează automat cod sursă pentru clienți și servere RPC în diferite limbaje de programare care permit aplicațiilor să comunice indiferent de limbajul folosit pentru scrierea clientului/serverului.
- Thrift supportă diferite limbaje de programare: C++, Java, C#, Python, PHP, Ruby.



Thrift Architecture.

Image from wikipedia.org

Apache Thrift

- Component cheie:
  - tipuri de date
  - transport
  - protocol
  - versioning
  - processors
- Tipuri de date Thrift
  - Sistemul de tipuri de date Thrift nu introduce tipuri speciale dinamice sau obiecte wrapper.
  - Programatorul nu trebuie să scrie cod pentru serializarea tipurilor de date sau pentru transportul datelor.
  - Conține tipuri de bază, structuri, containere.
  - Tipuri de bază:
    - *bool* - valoarea booleană: true sau false
    - *byte* - a signed byte
    - *i16* - întreg cu semn pe 16-biți
    - *i32* - întreg cu semn pe 32-biți
    - *i64* - întreg cu semn pe 64-biți
    - *double* - real pe 64-biți
    - *string*
    - *binary* - șir de octeți folosit pentru reprezentarea blob-urilor.

Tipuri Thrift -Structuri

- O structură Thrift definește un obiect comun tuturor limbajelor ce poate fi transmis prin rețea.
- O structură este echivalentul unei clase dintr-un limbaj orientat pe obiect.
- Structura conține o mulțime de câmpuri care au un tip definit și un identificator unic asociat.
- Sintaxa asemănătoare structurii C.
- Câmpurile pot fi adnotate cu un identificator numeric unic (valoare întreagă) și o valoarea implicită (opțional). Identificatorii trebuie să fie unici în cadrul structurii.
- Dacă identificatorii câmpurilor sunt omiși, li se vor atribui automat valori.

```
struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

Tipuri Thrift -Containere

- Containerele Thrift sunt containere cu tip care se vor mapa la cele mai folosite containere din limbajele de programare corespunzătoare.
- Sunt parametrizate folosind stilul Java Generics/C++ template.
- Trei tipuri de containere disponibile:
  - **`list<type>`** : o listă de elemente.
    - Se mapează la STL **`vector`**, Java **`ArrayList`**, sau orice alt tablou din limbajele scripting. Poate conține duplicate.
  - **`set<type>`** : O mulțime neordonată de elemente.
    - Se mapează la STL **`set`**, Java **`HashSet`**, **`set`** în Python, sau dicționare native în PHP/Ruby.
  - **`map<type1,type2>`** :Un dicționar cu chei unice
    - Se mapează la STL **`map`**, Java **`HashMap`**, PHP associative array, sau Python/Ruby dictionary.
- Elementele din container pot fi de orice tip valid Thrift, inclusiv alte containere sau structuri.
- În codul generat corespunzător limbajului de programare dorit, fiecare definiție va conține două metode, **`read`** și **`write`**, folosite pentru serializarea si transportul obiectelor folosind Thrift TProtocol.

Thrift - Excepții

- Excepțiile Thrift sunt sintactic și funcțional echivalente cu structurile Thrift doar că sunt declarate folosind **`exception`** în loc de **`struct`**.
- Clasele generate vor moșteni din clasele de bază corespunzătoare excepțiilor în limbajul de programare folosit, pentru a se putea folosi în mod transparent cu mecanismul de tratare a excepțiilor din limbajul respectiv.

Servicii Thrift

- Serviciile Thrift se definesc folosind tipurile Thrift
- Definirea unui serviciu este echivalentă semantic cu definirea unui interfețe într-un limbaj de programare orientat pe obiecte.
- Compilatorul Thrift va genera stub-uri client și server complet funcționale care implementează interfața.
- Lista parametrilor și lista excepțiilor sunt implementate ca și structuri Thrift.

```
service <name> {
  <returntype> <name>(<arguments>)[throws (<exceptions>)]
  ...
}
```

```
service StringCache {
   void set(1:i32 key, 2:string value),
   string get(1:i32 key) throws (1:KeyNotFound knf),
   void delete(1:i32 key)
}
```

Nivelul transport Thrift

- Nivelul transport este folosit de codul generat automat pentru a ușura transmiterea datelor între clienți și server.
- Thrift se folosește de obicei peste TCP/IP ca și nivel de bază pentru comunicare.
- Codul generat Thrift trebuie să știe doar cum să scrie și cum să citească datele.

- Originea sau destinația datelor sunt irelevante: poate fi socket, memorie partajată sau un fișier pe discul local.
- Interfața Thrift **`TTransport`** conține metodele:
  - **`open`** deschiderea transportului
  - **`close`** închiderea transportului
  - **`isOpen`** verifică dacă transportul este deschis
  - **`read`** citește date
  - **`write`** scrie date
  - **`flush`** forțează scrierea datelor păstrate în zona tampon.

Nivelul transport Thrift

- Interfață **`TServerTransport`** folosită pentru crearea și acceptarea obiectelor transport:
  - **`open`** deschidere
  - **`listen`** așteaptă conexiuni de la clienți
  - **`accept`** returnează un nou obiect transport (când s-a conectat un client nou)
  - **`close`** închidere
- Interfața transport este proiectată pentru implementare ușoară în orice limbaj de programare.
- Se pot defini noi mecanisme de transport:
  - Clasa **`TSocket`** este implementată în toate limbajele de programare suportate. Oferă o modalitate simplă de comunicare cu un socket TCP/IP.
  - Clasa **`TFileTransport`** este o abstractizare a unui stream ce reprezintă un fișier de pe discul local. Poate fi folosită pentru a salva cererile clienților într-un fișier de pe disc.

Thrift - Protocol

- Thrift cere respectarea unei anumite structurii a mesajelor când sunt transmise prin rețea, dar nu știe de protocolul efectiv folosit pentru serializarea/codificarea acestora.
- Nu contează dacă datele sunt serializate folosind XML, human-readable ASCII (stringuri) sau octeți, dacă mecanismul suportă o mulțime de operații prestabile care permit citirea și scrierea datelor.
- Interfața Thrift Protocol suportă:
  - trimiterea mesajelor bidirectional,
  - codificarea tipurilor de bază, a structurilor și a containerelor.
- Are o implementare care folosește un protocol binar.
- Scrie toate datele într-un format binar:
  - Tipurile întregi sunt convertite într-un format rețea independent de limbaj,
  - Stringurile au adăugate la început lungimea lor în număr de octeți.
  - Mesajele și antetul câmpurilor sunt scrise folosind construcții de serializare a datelor întregi.
  - Numele câmpurilor nu sunt serializate, identificatorii asociați sunt suficienți pentru reconstruirea datelor.

Thrift - Protocol

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
...
```

```
name, type, seq = readMessageBegin();
readMessageEnd()
name = readStructBegin()
readStructEnd()
name, type, id = readFieldBegin()
 readFieldEnd()
k, v, size = readMapBegin()
readMapEnd()
...
```

Thrift - Versioning

- Thrift este robust la schimbări de versiuni și de definiție a datelor.
- Versioning este implementat folosind identificatorii asociați câmpurilor:
  - Antetul unui câmp dintr-o structură Thrift este codificat folosind identificatorul unic.
  - Combinația (identificator câmp, tip câmp) este folosită pentru a identifica unic un câmp din structură.
- Limbajul IDL Thrift permite atribuirea automată de identificatori pentru câmpuri, dar se recomandă definirea explicită a acestora.
- Dacă la parsare/decodificare/deserializare se întâlnește un câmp necunoscut acesta este ignorat și distrus.
- Dacă un câmp care ar trebui să existe nu apare, programatorul poate fi notificat de lipsa acestuia (folosind structura **`isset`** definită în interiorul fiecărei obiect).
- Obiectul *isset* din interiorul unei structuri Thrift poate fi interogat pentru a determina existența unui anumit câmp. De fiecare dată când se primește o instanță a unei structuri, programatorul ar trebui să verifice existența unui câmp înainte de folosirea lui.

Thrift - Implementare RPC

- Instanțe a clasei **`TProcessor`** sunt folosite pentru a trata cererile RPC:

```
interface TProcessor {
bool process(TProtocol in, TProtocol out) throws TException
}
```

- Pentru fiecare serviciu dintr-un fișier .thrift se generează următoarele:
  - o interfață ServiceIf corespunzătoare serviciului
  - clasa ServiceClient implementează ServiceIf
    TProtocol in
    TProtocol out
  - clasa ServiceProcessor : TProcessor
    ServiceIf handler
  - clasa ServiceHandler implementează ServiceIf
  - TServer(TProcessor processor,
    TServerTransport transport,
    TTransportFactory tfactory,
    TProtocolFactory pfactory)

  serve()

Thrift - Implementare RPC

- Serverul încapsulează logica corespunzătoare conexiunii, threadurilor, etc, iar obiectul de tip TProcessor se ocupă de apelul metodelor la distanță.
- Programatorul trebuie să scrie doar codul din fișierul .thrift și implementarea corespunzătoare serviciilor (ServiceHandler)
- Există mai multe implementări posibile pentru TServer:
  - TSimpleServer: server secvențial
  - TThreadedServer: server concurent (se creează câte un thread pentru fiecare conexiune)
  - TThreadPoolServer: server concurent care folosește un container de threaduri.
- Exemplu: Text transformer

Bibliografie

- Protocol Buffers

  https://developers.google.com/protocol-buffers/

  https://github.com/google/protobuf

- gRPC

  http://www.grpc.io/

- Apache Thrift

  http://thrift-tutorial.readthedocs.io/en/latest/index.html

  https://thrift.apache.org/