

# Medii de proiectare și programare

2021-2022

Curs 7

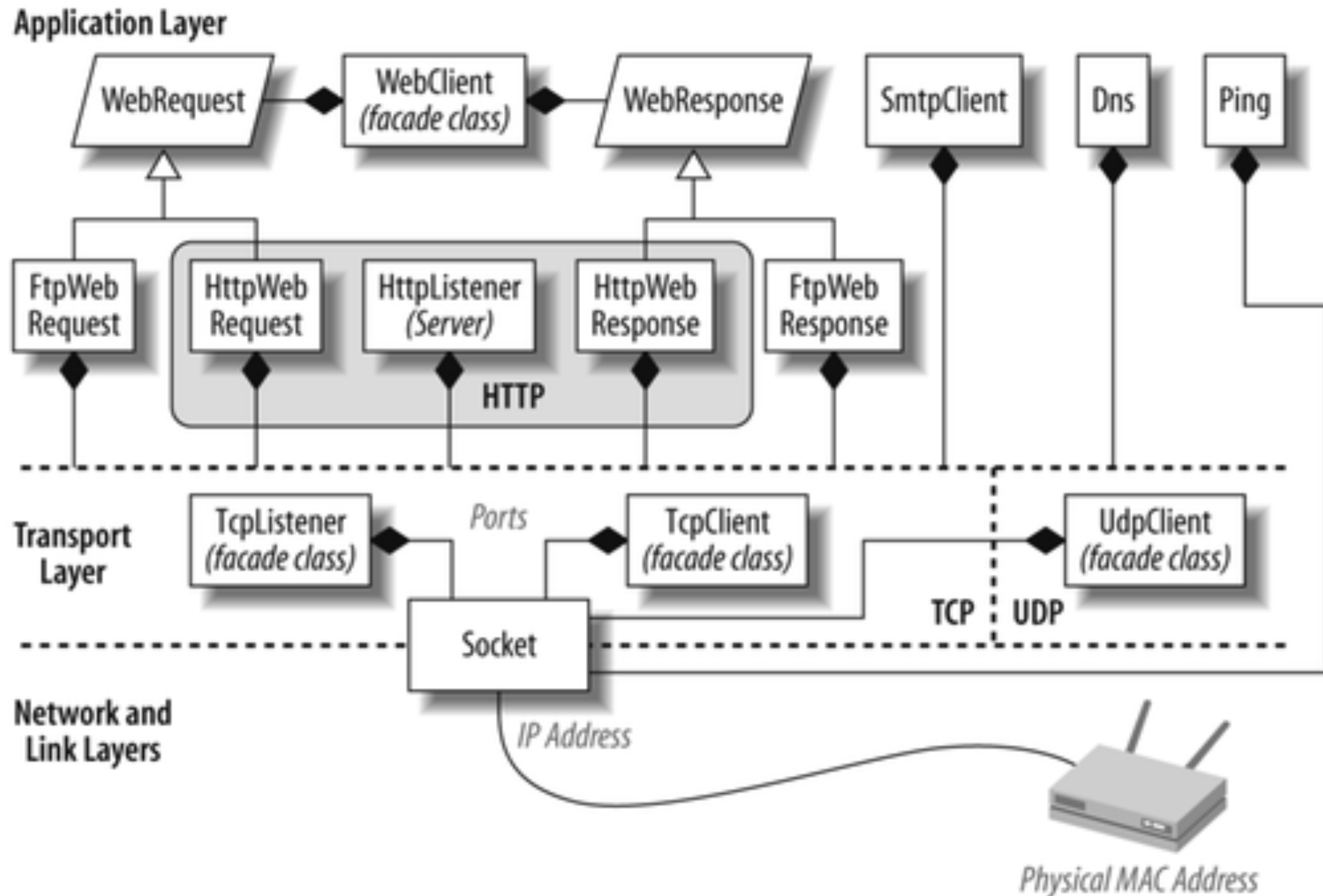
# Conținut curs 7

- Networking si threading in C#
  - Exemplu Mini-Chat (networking) în C#
- Remote Procedure Call
- Aplicații distribuite cross-platform

# Networking în C#

- .NET conține clase pentru comunicarea prin rețea folosind protocoale standard cum ar fi HTTP, TCP/IP și FTP.
- Spațiul de nume **System.Net.\***:
  - **WebClient** fațade pentru operații simple de download/upload folosind HTTP sau FTP.
  - **WebRequest** și **WebResponse** pentru operații HTTP și FTP complexe.
  - **HttpListener** pentru implementarea unui HTTP server.
  - **SmtpClient** pentru construirea și trimiterea mesajelor folosind SMTP.
  - **TcpClient**, **UdpClient**, **TcpListener** și **Socket** pentru acces direct la nivelul rețea.

# Networking in C#



# Networking în C#

- Clasa `IPAddress` din `System.Net` reprezintă o adresă IPv4 (32 bits) sau IPv6 (128 bits).

```
IPAddress a1 = new IPAddress (new byte[] { 172, 30, 106, 5 });
```

```
IPAddress a2 = IPAddress.Parse ("172.30.106.5");
```

```
IPAddress a3 = IPAddress.Parse  
    (" [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]"); //IPv6
```

- O asociere între o adresă IP și un port este reprezentată folosind clasa `IPEndPoint`:

```
IPAddress a = IPAddress.Parse ("172.30.106.5");
```

```
IPEndPoint ep = new IPEndPoint (a, 55555); // Port 55555
```

```
Console.WriteLine (ep.ToString()); // 172.30.106.5:55555
```

- Porturile: 1 – 65535.
- Porturile dintre 49152 și 65535 nu sunt rezervate oficial.

# System.Net.Sockets Namespace

- Clasele **TcpClient**, **TcpListener** și **UdpClient** încapsulează detaliile creării conexiunilor de tip TCP și UDP.
- **Socket** implementează interfața Berkeley socket.
- **SocketException** excepția aruncată când apare o eroare la comunicarea prin socket.
- **NetworkStream** streamul folosit pentru comunicarea prin rețea.

# TcpListener

- TCP server:

```
TcpListener listener = new TcpListener (<ip address>, port);  
listener.Start();  
while (keepProcessingRequests)  
    using (TcpClient c = listener.AcceptTcpClient( ))  
        using (NetworkStream n = c.GetStream( ))    {  
            // Read and write to the network stream...  
        }  
listener.Stop ();
```

- **TcpListener** necesită adresa IP la care va aștepta conexiunile clienților (dacă calculatorul are două sau mai multe plăci de rețea).
  - **IPAddress.Any** ascultă pe toate adresele IP locale (sau singura).
- **AcceptTcpClient** blochează execuția până când se conectează un client.

# TcpClient

- Client Tcp:

```
using (TcpClient client = new TcpClient (<address>, port))  
    using (NetworkStream n = client.GetStream( ))  
    {  
        // Read and write to the network stream...  
    }
```

- **TcpClient** încearcă crearea conexiunii în momentul creării obiectului folosind adresa IP și portul specificate.
- Constructorul blochează execuția până la stabilirea conexiunii.



# NetworkStream

- **NetworkStream** comunicare bidirecțională pentru transmiterea și recepționarea datelor după stabilirea unei conexiuni.
- Methods:
  - **Read**
  - **Close**
  - **Write**
  - **Seek**
  - **Flush**
- Properties:
  - **CanRead, CanWrite**
  - **Socket**
  - **DataAvailable**
  - **Length**

# Threading în C#

- Spatiul de nume `System.Threading` clase și interfețe pentru programarea concurentă:
  - Clasa `Thread`.
  - Delegate: `ThreadStart`, `ParameterizedThreadStart`.
  - Sincronizare: `lock`, `Monitor`, `Mutex`, `Semaphore`, `EventHandles`.
- Delegates: reprezintă metoda executată de un thread.

```
public delegate void ThreadStart();
```

```
public delegate void ParameterizedThreadStart(Object obj);
```

- Clasa `Thread`: crearea unui thread, setarea priorității, obținerea informațiilor despre statusul unui thread.

```
public Thread(ThreadStart start);
```

```
public Thread(ParameterizedThreadStart start);
```

# Threading in C#

```
class Program {
    static void Main(string[] args) {
        Worker worker=new Worker();
        Thread t1=new Thread(new ParameterizedThreadStart(static_run));
        Thread t2=new Thread(new ThreadStart(worker.run));
        t1.Start("a");
        t2.Start();
    }
    static void static_run(Object data) {
        for(int i=0;i<26;i++) { Console.Write("{0} ",data); }
    }
}
class Worker {
    public void run() {
        for(int i=0;i<26;i++) Console.Write("{0} ",i);
    }
}
//a a a a a a a a a a 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 a a a a a a a a a a a a a a a a a
```

# Sincronizarea threadurilor

- Diferite tipuri:
  - *Blocarea exclusivă*: doar un singur thread poate executa o porțiune de cod la un moment dat.
    - `lock`, `Mutex`, and `SpinLock`.
  - *Blocarea nonexclusivă*: limitarea concurenței.
    - `Semaphore` and `ReaderWriterLock`.
  - *Semnalizarea*: un thread poate bloca execuția până la primirea unei notificări de la unul sau mai multe threaduri.
    - `ManualResetEvent`, `AutoResetEvent`, `CountdownEvent` ȘI `Barrier`.

# Sincronizarea threadurilor –Blocarea

- Instrucțiunea `lock`:

```
lock(locker_obj) {  
    //code to execute  
}
```

- `locker_obj` - tip referință.
- Doar un singur thread poate obține accesul la un moment dat. Dacă mai multe threaduri încearcă să obțină accesul, ele sunt puse într-o coadă și primesc accesul pe baza regulii “primul venit-primul servit”.
- Dacă un alt thread a obținut deja accesul, threadul curent nu își continuă execuția până nu obține accesul.

# Sincronizarea threadurilor - Signaling

- *Event wait handles* - construcții simple pentru semnalizare:
  - **EventWaitHandle** - reprezintă un eveniment pentru sincronizarea threadurilor. Unul sau mai multe threaduri blochează execuția folosind un **EventWaitHandle** până când un alt thread apelează metoda **Set** permițând execuția unuia sau mai multor threaduri aflate în așteptare.
  - **AutoResetEvent**, **ManualResetEvent**
  - **CountdownEvent** (Framework 4.0)
- **AutoResetEvent** notifică un thread aflat în așteptare de apariția unui eveniment (doar un singur thread).
  - **Set()** - eliberează un thread aflat în așteptare
  - **WaitOne()** - threadul așteaptă apariția unui eveniment

Observații:

1. Dacă **Set** este apelată când nici un thread nu se află în așteptare, handle -ul așteaptă până când un thread apelează metoda **WaitOne**.
2. Apelarea metodei **Set** de mai multe ori când nici un thread nu este în așteptare nu va permite mai multor threaduri obținerea accesului când apelează metoda **WaitOne**.

# Sincronizarea threadurilor - Signaling

- **ManualResetEvent** (asemănător **AutoResetEvent**) – notifică toate threadurile aflate în așteptare la apariția unui eveniment.

- Crearea unui event wait handle:

- Constructori:

```
AutoResetEvent waitA=new AutoResetEvent(false);
```

```
AutoResetEvent waitA=new AutoResetEvent(true); //calls Set
```

```
ManualResetEvent waitM=new ManualResetEvent(false);
```

- Clasa **EventWaitHandle**

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

```
var manual = new EventWaitHandle (false, EventResetMode.ManualReset);
```

- Distrugerea unui wait handle:

- Apelul metodei **Close** pentru eliberarea resurselor sistemului de operare.
- Ștergerea referințelor pentru a permite garbage collector-ului distrugerea obiectului.

# Exemplu Signaling

```
class WaitHandleExample
{
    static EventWaitHandle waitHandle = new AutoResetEvent (false);
    static void Main()
    {
        new Thread (Worker).Start();
        Thread.Sleep (1000);    // Pause for a second...
        waitHandle.Set();    // Wake up the Worker.
    }
    static void Worker()
    {
        Console.WriteLine ("Waiting...");
        waitHandle.WaitOne();    // Wait for notification
        Console.WriteLine ("Notified");
    }
}
```



# Task-uri C#

- Limitările threadurilor:
  - Se pot transmite ușor date unui thread, dar nu se poate obține la fel de ușor rezultatul execuției unui thread.
  - Dacă execuția threadului aruncă o excepție, tratarea excepției și retransmiterea ei este mai dificil de implementat.
  - Nu se poate seta ca un thread să execute altceva când și-a încheiat execuția.
- Un *Task* C# reprezintă o operație concurentă care poate fi (sau nu) executată folosind threaduri.
  - Taskurile pot fi compuse.
  - Pot folosi un container de threaduri pentru a reduce timpul necesar pornirii execuției.
- Tipul **Task** a fost introdus începând cu Framework 4.0 ca făcând parte din biblioteca pentru programare paralelă.
- **System.Threading.Tasks** namespace.

# Pornirea execuției unui Task

- Framework 4.5 - Metoda statică **Task.Run** (pornirea execuției unui task folosind threaduri) - parametru de tip **Action** delegate:

```
Task.Run (() => Console.WriteLine ("Ana")) ;
```

- Framework 4.0 - Metoda statică **Task.Factory.StartNew**:

```
Task.Factory.StartNew (() => Console.WriteLine ("Ana")) ;
```

- Implicit taskurile folosesc threaduri din containere deja create.
- Folosirea metodei **Task.Run** - similar cu execuția explicită folosind threaduri:
  - **new Thread** (() => **Console.WriteLine** ("Ana")).**Start**() ;
- **Task.Run** returnează un obiect **Task** care poate fi folosit pentru monitorizarea progresului.
- Nu este necesară apelarea metodei **Start**.

# Obținerea rezultatului execuției

- **Task<TResult>** subclasă a clasei **Task** care permite returnarea rezultatului execuției.
- **Task<TResult>** poate fi obținut apelând **Task.Run** folosind un delegate de tip **Func<TResult>** (sau o expresie lambda compatibilă).
- Rezultatul poate fi obținut folosind proprietatea **Result**.
- Dacă taskul nu și-a încheiat execuția, apelul proprietății **Result** va bloca execuția threadului curent până la terminarea execuției taskului:

```
Task<int> task = Task.Run(()=>{int x=2; return 2*x; });  
int result = task.Result;    // Blocks if not already finished  
Console.WriteLine (result);    // 4
```

# Taskuri și excepții

- Taskurile propagă excepțiile (threadurile nu).
- Dacă un task aruncă o excepție, excepția este rearuncată către codul care apelează metoda `Wait()` a clasei `Task` sau care apelează proprietatea `Result` a clasei `Task<TResult>`:
- Excepția va fi inclusă într-o excepție de tip `AggregateException` de către CLR:

// Start a Task that throws a NullReferenceException:

```
Task task = Task.Run (() => { throw null; });

try{
    task.Wait();
}catch (AggregateException aex){
    if (aex.InnerException is NullReferenceException)
        Console.WriteLine ("Null!");
    else throw;
}
```

# Actualizare GUI

- Interfețele grafice de tip Windows Forms folosesc obiecte de tip Control.
- Aceste obiecte pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat.
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.
- Dacă se dorește apelarea unui membru (metoda, atribut, proprietăți) a obiectului X creat într-un thread Y, cererea trebuie transmisă threadului Y (folosind metoda **Invoke** sau **BeginInvoke** a obiectului X).

# Actualizare GUI

- **Invoke** și **BeginInvoke** au un parametru de tip delegate care referă metoda corespunzătoare obiectului de tip Control care se dorește a se executa.
- **Invoke** execuție sincronă: execuția apelantului este blocată până la actualizarea controlului.
- **BeginInvoke** execuție asincronă: execuția apelantului continuă, iar cererea este pusă într-o coadă (corespunzătoare evenimentelor de la tastatură, mouse, etc) și se va executa ulterior.

# Exemplu

//1. definirea unei metode pentru actualizarea unui ListBox

```
private void updateListBox(ListBox listBox, IList<String> newData) {  
    listBox.DataSource = null;  
    listBox.DataSource = newData;  
}
```

//2. definirea unui delegate care va fi apelat de GUI Thread

```
public delegate void UpdateListBoxCallback(ListBox list, IList<String>  
data);
```

//3. în celălalt thread se transmite metoda care actualizeaza ListBox:

```
list.Invoke(new UpdateListBoxCallback(this.updateListBox), new Object[]  
{list, data});
```

or

```
list.BeginInvoke(new UpdateListBoxCallback(this.updateListBox), new  
Object[]{list, data});
```

# Exemplu C#

- O aplicație simplă client/server:
  - Serverul așteaptă conexiuni.
  - Clientul se conectează la server și îi trimite un text.
  - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

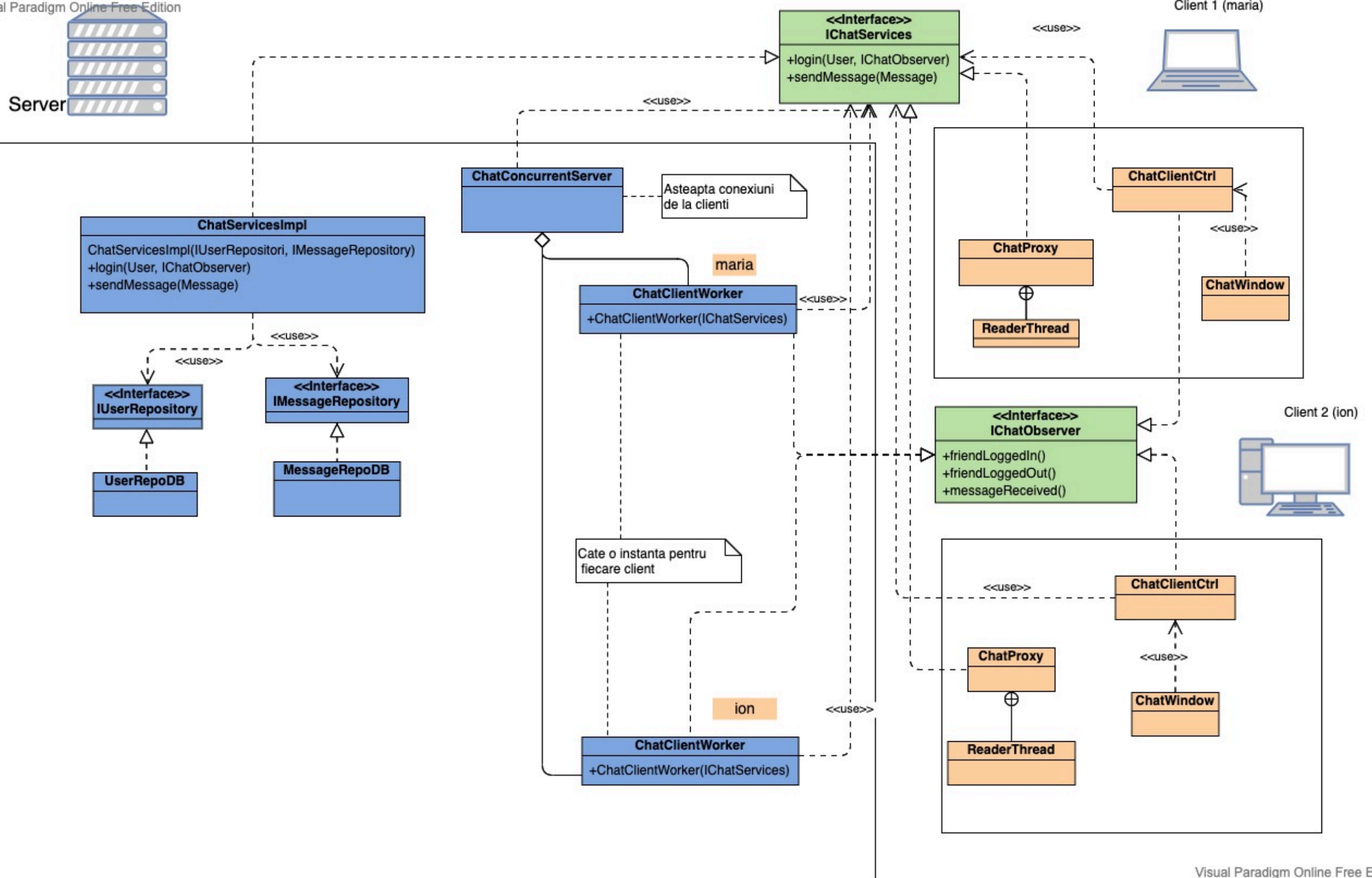


# Exemplu C#

- Mini-chat C#

# Networking

Visual Paradigm Online Free Edition



Visual Paradigm Online Free Edition

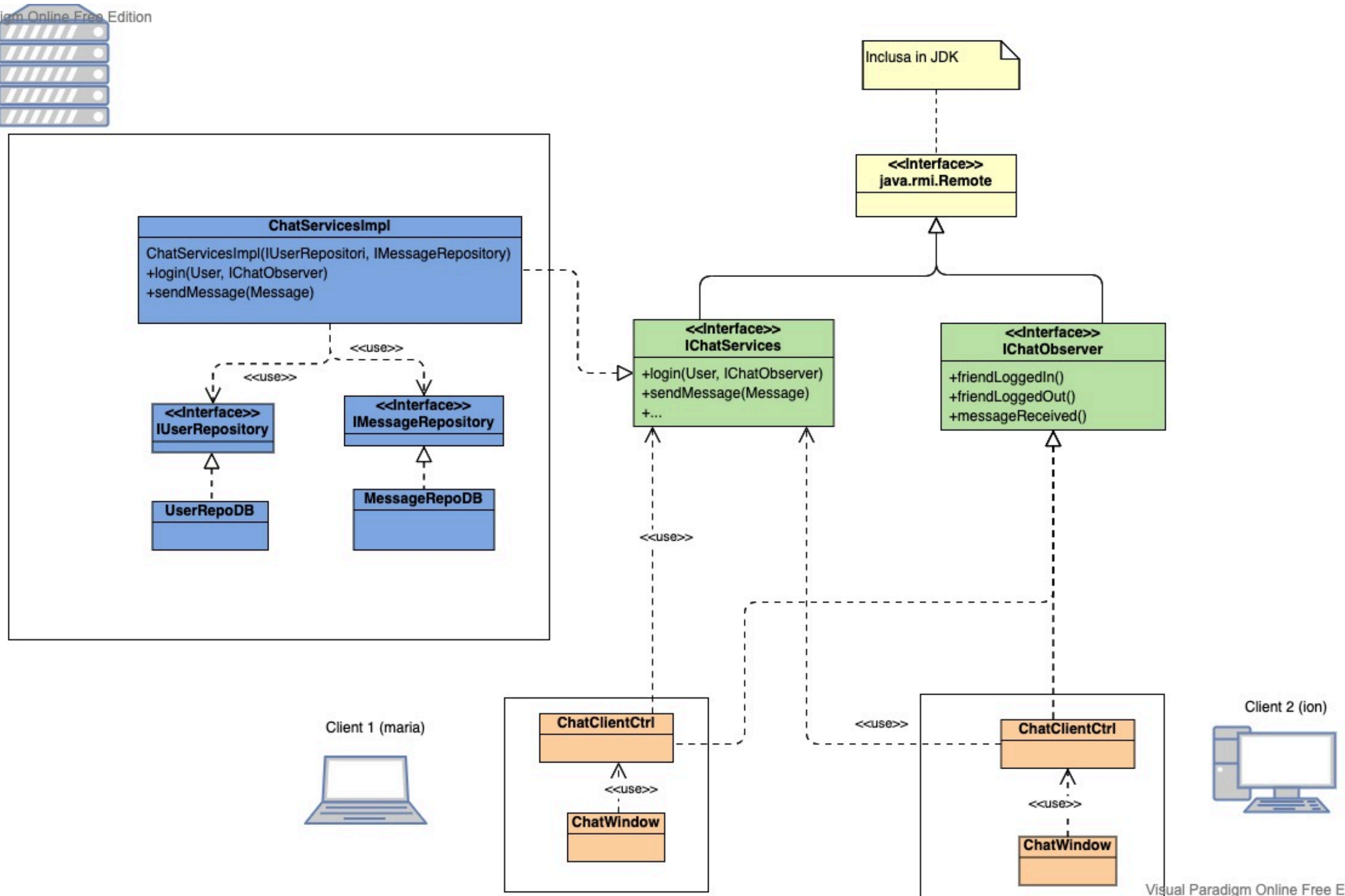
# Remote Procedure Call

- Apelul procedurilor la distanță (eng. *Remote procedure call* - *RPC*) este o tehnologie de comunicare între procese care permite unei aplicații să inițieze execuția unei subrutine sau a unei proceduri în alt spațiu de adrese, fără ca programatorul să scrie explicit codul corespunzător interacțiunii dintre procese.
- Programatorul scrie aproximativ același cod indiferent dacă apelează o rutină locală (din același proces) sau una la distanță (din alt proces, în alt spațiu de adrese).
- Când se folosește paradigma orientată pe obiecte RPC - apeluri la distanță sau apelul metodelor la distanță.
- RPC este adesea folosit pentru implementarea aplicațiilor client-server.
- Un apel la distanță este inițiat de client prin trimiterea unei cereri către un server la distanță pentru execuția unei proceduri cu parametrii dați. Serverul trimite un răspuns clientului, iar aplicația își continuă execuția.
- Cât timp serverul procesează cererea clientului, execuția clientului este blocată (așteaptă până când serverul a terminat procesarea cererii).

# Java RMI

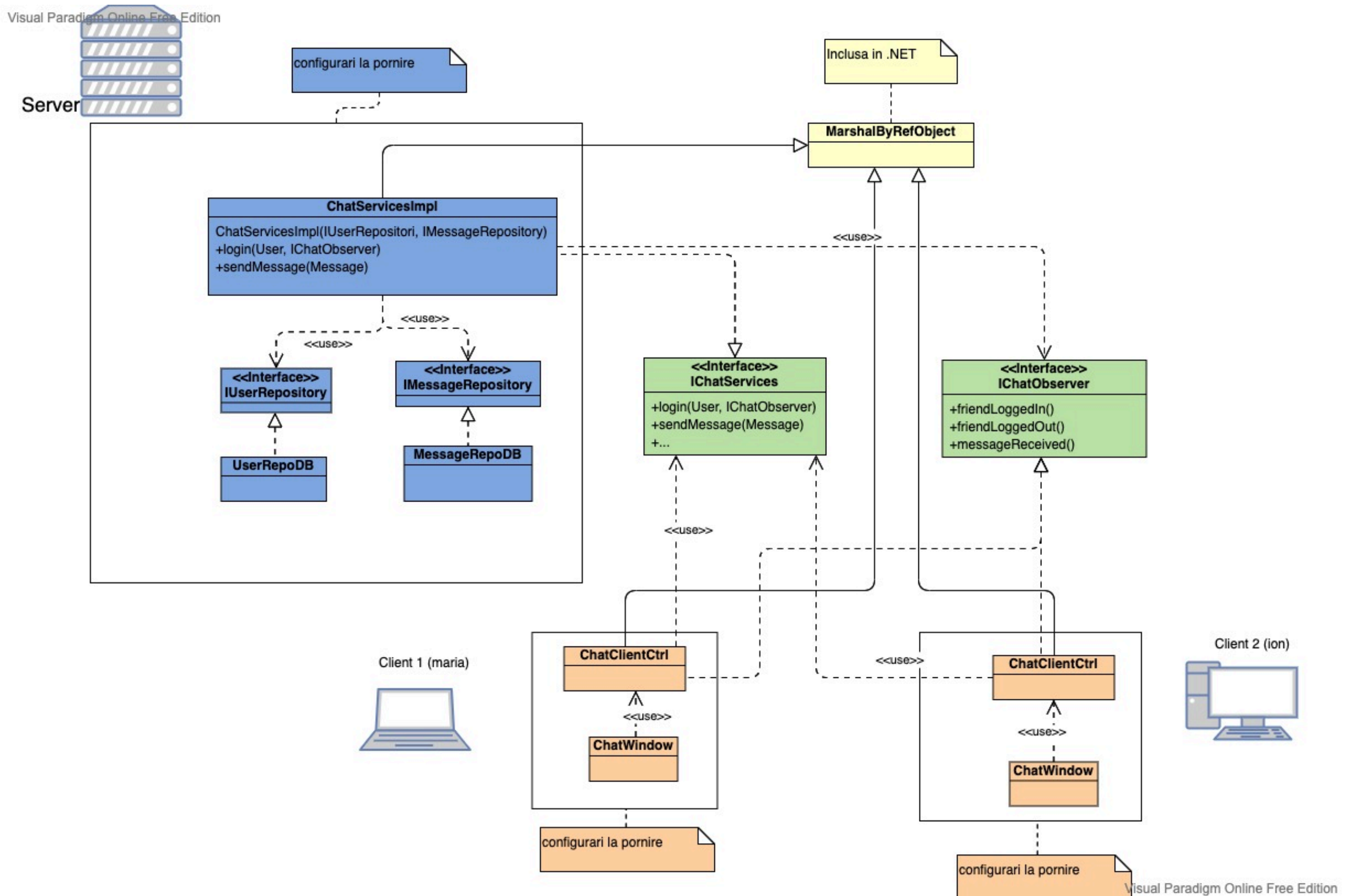
Visual Paradigm Online Free Edition

Server



Visual Paradigm Online Free Edition

# NET. Remoting



# Remote Procedure Call

## Java RMI, NET. Remoting

- Avantaje

- Dezvoltarea ușoară a aplicațiilor client-server
- Ascunderea detaliilor de implementare pentru comunicarea client-server.

- Dezavantaje

- Aplicațiile se pot dezvolta într-un singur limbaj (Java/C#/etc)
- Dacă apar versiuni noi pentru interfețe (servicii), trebuie modificați toți clienții.

# Aplicații distribuite cross-platform

- CORBA
- Protobuf
- gRPC
- Thrift

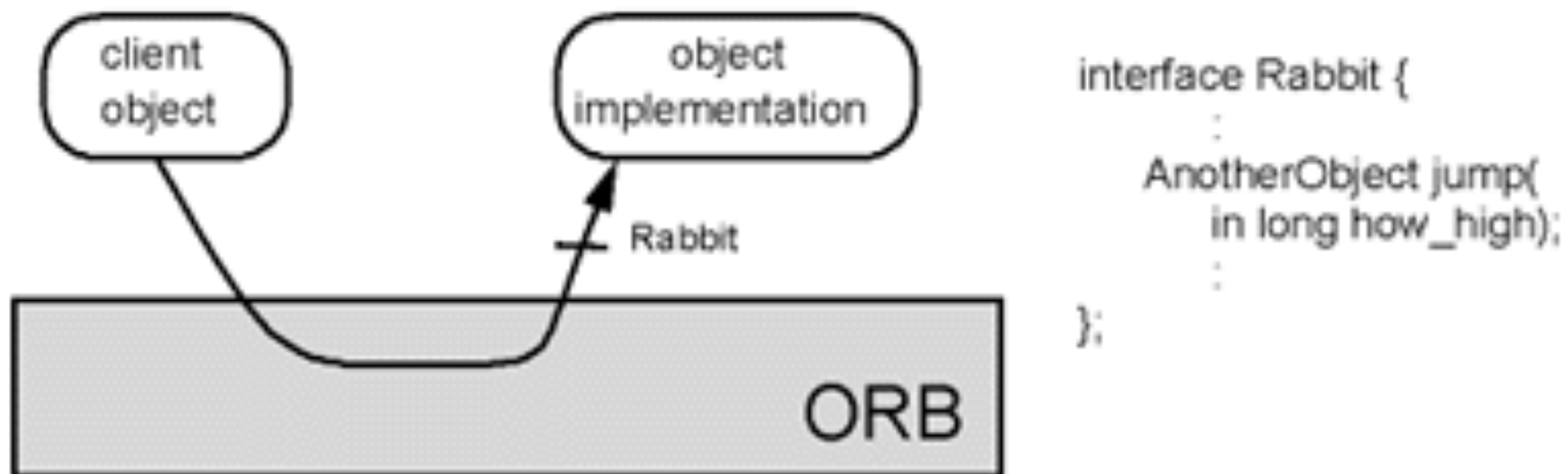
# CORBA

- A fost propusă de OMG (Object Management Group) înainte de anul 2000.
- CORBA (Common Object Request Broker Architecture) este o arhitectură standard pentru sisteme distribuite orientate pe obiect.
- Produsele CORBA oferă un framework pentru dezvoltarea și execuția aplicațiilor distribuite.
- Permite unei colecții de obiecte distribuite *heterogene* să interacționeze.
- 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.



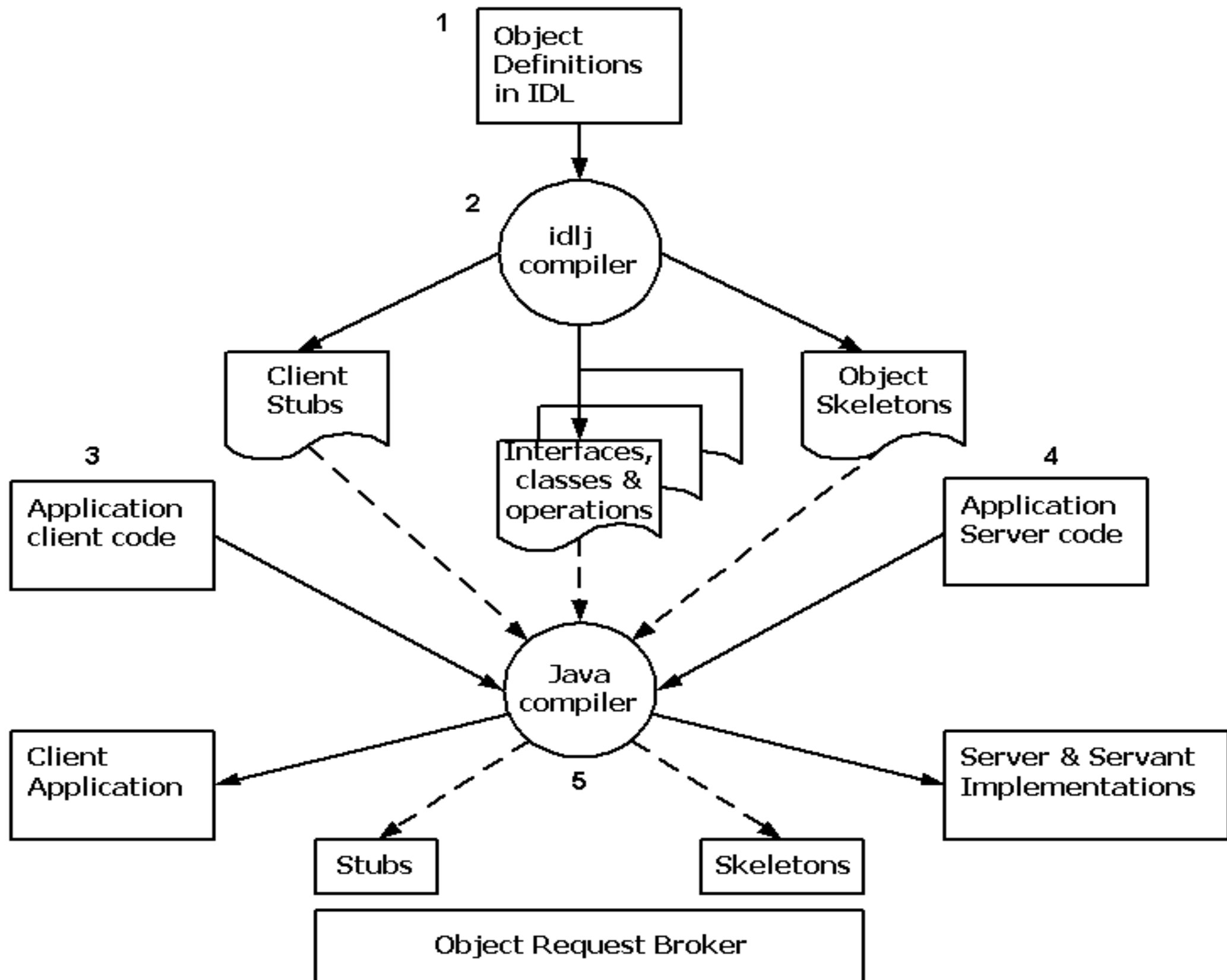
# Interfețe IDL

- 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.

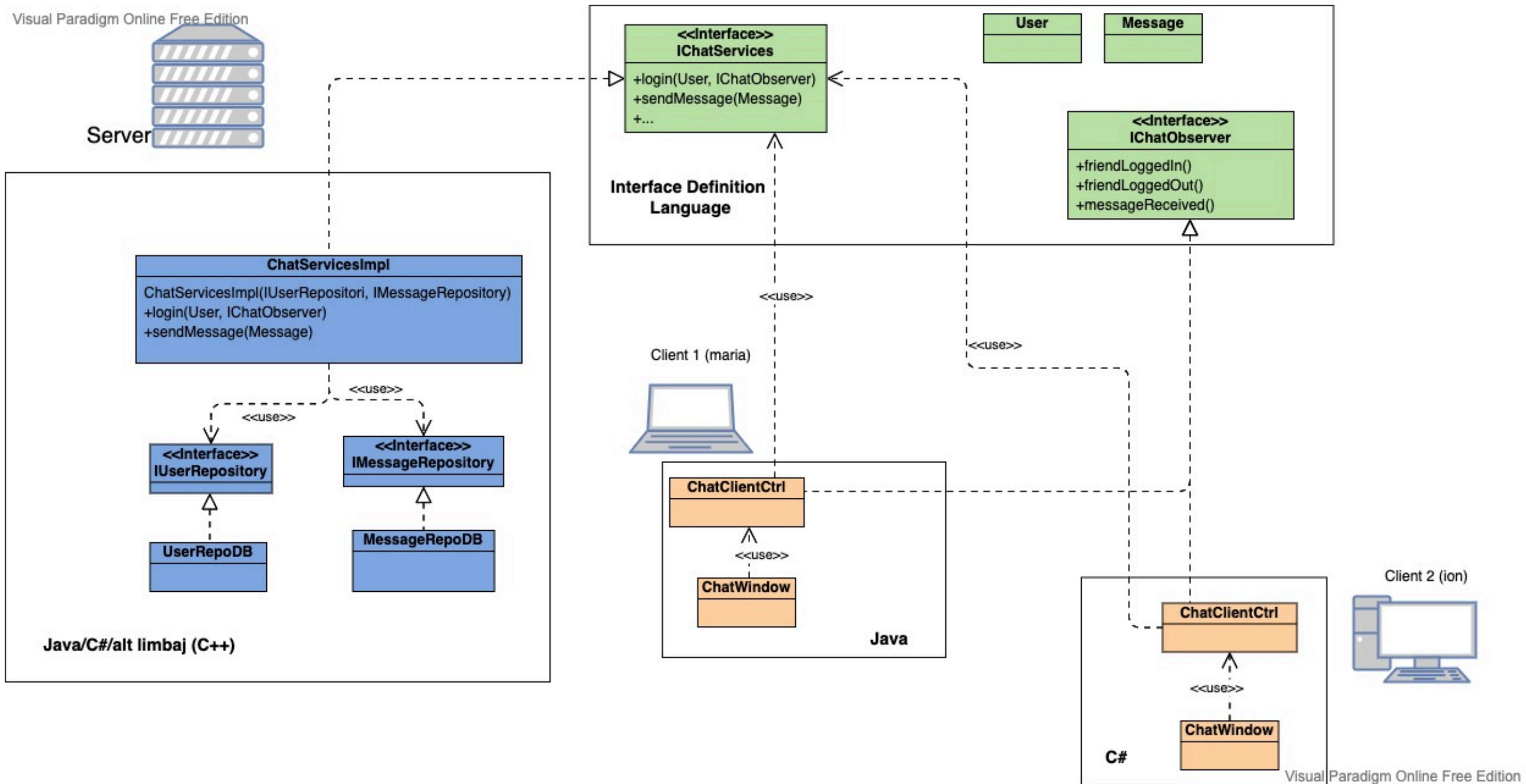
# Exemplu IDL

```
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



# CORBA și Java



# Protocol Buffers (Protobuf)

- Reprezintă o modalitate independentă de limbaj și platformă de a serializa structuri de date folosite în protocoale 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 singură 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 (ambiguous)
  - 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

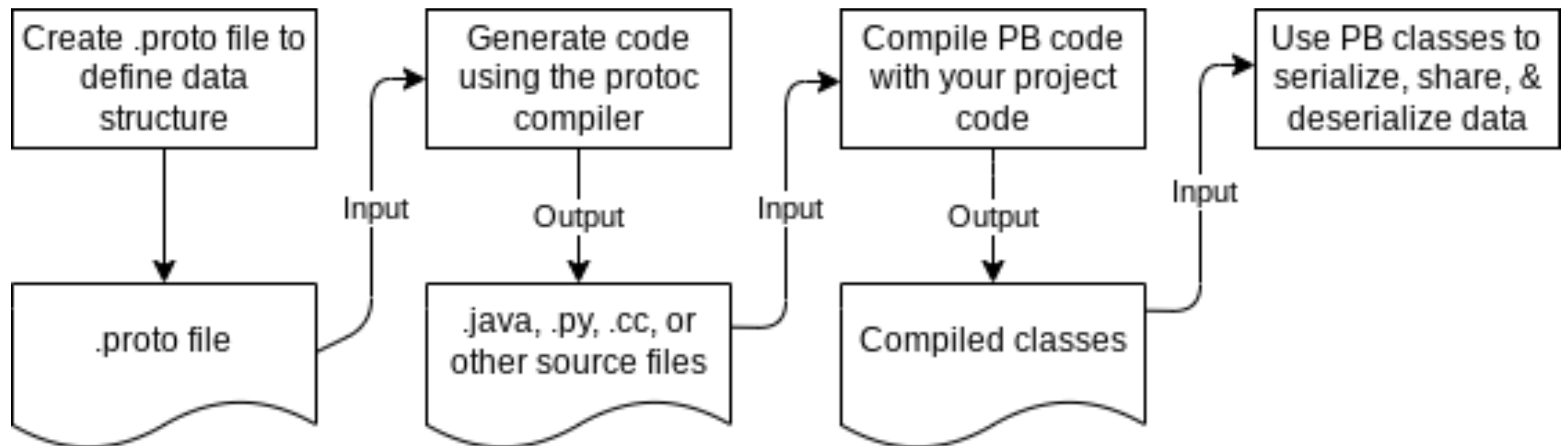
```
person.getElementsByTagName("name").item(0).getNodeValue()
person.getElementsByTagName("email").item(0).getNodeValue();
```

//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 Workflow\*



\*<https://developers.google.com/protocol-buffers/docs/overview>

# Protocol Buffers vers. 2 – 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 - vers. 3

- 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 - vers. 3

- 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.
- Această valoare va fi folosită ca și valoare 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 - vers. 3

- 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 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 în 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 aceeași 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 serializării 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 aceiași 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 semnificația 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](https://github.com/google/protobuf/blob/master/src/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.protobufprotocol";
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;
}
```

# Bibliografie

- Protocol Buffers

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

<https://github.com/google/protobuf>

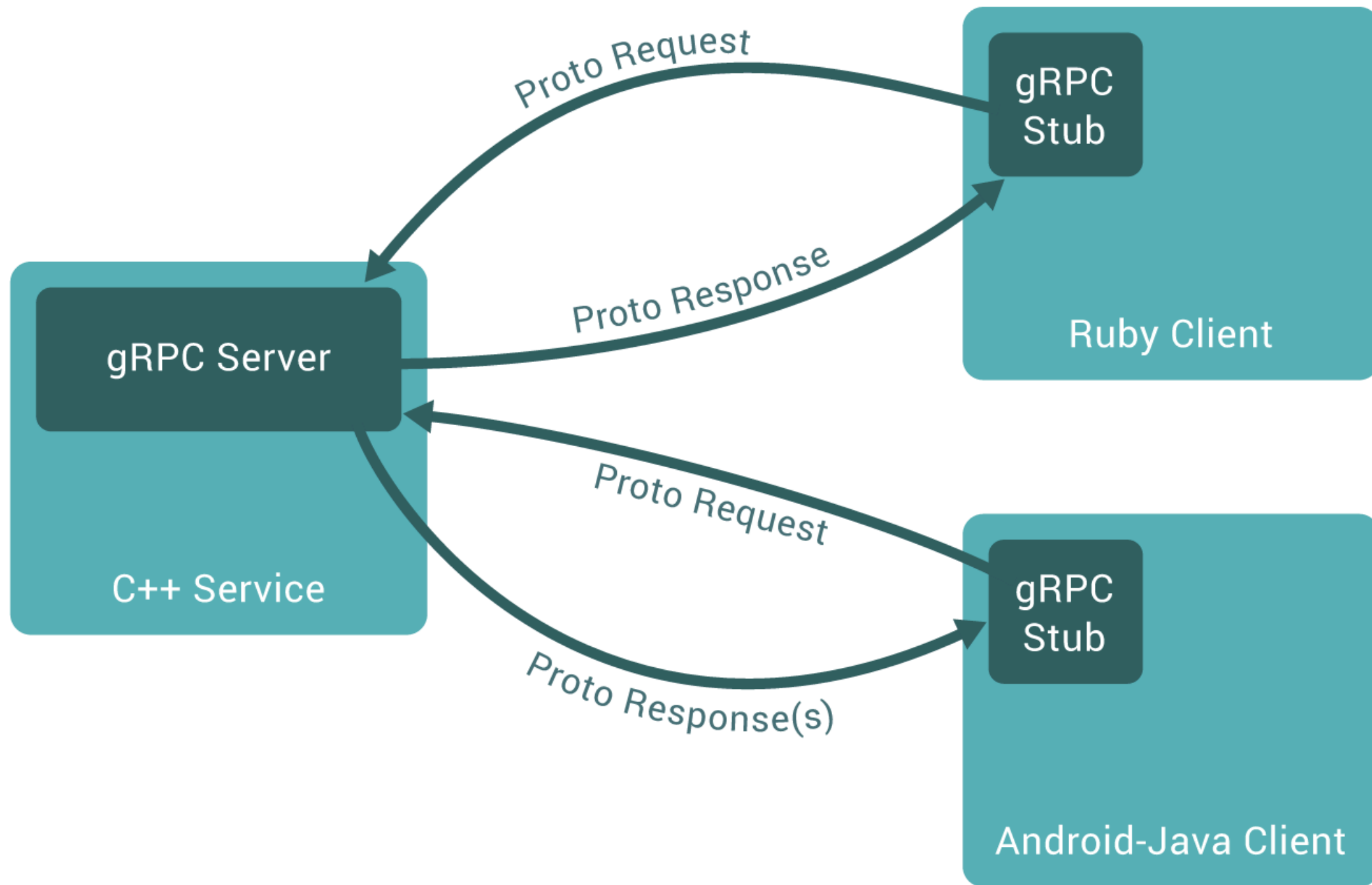
- Exem plu MiniChat



# gRPC

- Cu gRPC o aplicație client poate apela metode la distanță ca și cum ar fi metode ale unui obiect local.
- Facilitează crearea aplicațiilor distribuite și a serviciilor.
- gRPC folosește ideea definirii unui serviciu (interfețe) care specifică metodele ce pot fi apelate la distanță, împreună 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



# 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 metode 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.

# Bibliografie

- Protocol Buffers

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

<https://github.com/google/protobuf>

- gRPC

<http://www.grpc.io/>