

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

**VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

****-----□□&□□-----

**BÁO CÁO BÀI TẬP LỚN**

**TÌM HIỂU VỀ RPC VÀ**

**CHƯƠNG TRÌNH MINH HỌA**

**Giáo viên hướng dẫn: Đỗ Quốc Huy**

**Sinh viên thực hiện**

**Ngô Phú Thái 20187197**

**Nguyễn Đức Thâu 20187199**

Mục lục

[RPC là gì? 2](#_Toc93330986)

[Kiến trúc của RPC 3](#_Toc93330987)

[Ưu điểm và nhược điểm 6](#_Toc93330988)

[Giới thiệu về gRPC 7](#_Toc93330989)

[Vì sao cần gRPC? 7](#_Toc93330990)

[RPC không phải là REST API 7](#_Toc93330991)

[gRPC hoạt động như thế nào 8](#_Toc93330992)

[Chương trình minh họa 9](#_Toc93330993)

[Tạo file Protocol Buffer: 9](#_Toc93330994)

[Tạo service chứa các thủ tục gRPC cho máy chủ: 10](#_Toc93330995)

[Gọi thủ tục từ ứng dụng máy khách: 11](#_Toc93330996)

# RPC là gì?

RPC – viết tắt của Remote Procedure Call, là một mô hình kỹ thuật mạng dùng cho các ứng dụng hệ phân tán, theo mô hình máy khách – máy chủ. Nó được dựa trên sự mở rộng của việc gọi hàm cục bộ thông thường cho nên RPC được coi là gọi hàm không cần nằm trong cùng một hệ thống. Đây là một loại giao thức yêu cầu-phản hồi, có thể dễ dàng được giải thích sử dụng mô hình truyền thông máy khách/máy chủ. Quá trình gọi một yêu cầu được gọi là “máy khách” và quá trình trả lời lại yêu cầu này gọi là “máy chủ”. Lưu ý, máy khách hoặc máy chủ có thể là các thiết bị khác nhau trong một hệ thống mạng hoặc có thể là các tiến trình khác nhau nằm trong cùng hệ thống. Có hai loại thủ tục xuất hiện trong chương trình gồm:

* Thủ tục cục bộ là thủ tục được cài đặt và thực thi tại máy của của chương trình.
* Thủ tục ở xa là thủ tục được định nghĩa, cài đặt và thực thi trên một máy tính khác.

# Kiến trúc của RPC

Khi thực hiện 1 thủ tục RPC:

Timeline

Description automatically generated

1. Môi trường gọi sẽ bị ngưng lại, các trường thủ tục được chuyển tiếp trong mạng tới môi trường mà chứa thủ tục sẽ được thực hiện.
2. Khi thủ tục hoàn thành và đưa ra kết quả, kết quả đó sẽ được chuyển lại về môi trường gọi nó, nơi mà sự thực thi được tiếp tục như thể vừa nhận được kết quả từ một lần gọi hàm thông thường.

RPC rất hợp cho mô hình máy khách – máy chủ khi mà dòng điều khiển của chúng hay qua lại nhau giữa người gọi và người được gọi. Theo lí thuyết, máy khách và máy chủ không thực thi cùng một thời điểm. Thay vào đó, luồng thực thi nhảy từ người gọi tới người được gọi và quay ngược lại.

Hoạt động trong RPC:

Diagram

Description automatically generated

**Phần Client** là một quá trình người dùng, nơi khởi tạo một lời gọi thủ tục từ xa. Mỗi lời gọi thủ tục ở xa trên phần Client sẽ kích hoạt một thủ tục cục bộ tương ứng nằm trong phần Stub của Client.

**Phần Client Stub** cung cấp một bộ các hàm cục bộ mà phần Client có thể gọi. Mỗi một hàm của Client Stub đại diện cho một hàm ở xa được cài đặt và thực thi trên server.

Mỗi khi một hàm nào đó của Client Stub được gọi bởi Client, Client Stub sẽ gửi thông điệp để mô tả thủ tục ở xa tương ứng mà Client muốn thực thi cùng với các tham số nếu có. Sau đó nó sẽ nhờ hệ thống RPC Runtime cục bộ gửi thông điệp này đến phần Server Stub của Server.

**Phần RPC Runtime** quản lý việc truyền thông điệp thông qua mạng giữa máy Client và máy Server. Nó đảm nhận việc truyền lại, báo nhận, chọn đường gói tin và mã hóa thông tin.

RPC Runtime trên máy Client nhận thông điệp yêu cầu từ Client Stub, gửi nó cho RPC Runtime trên máy Server bằng lệnh send(). Sau đó gọi lệnh wait() để chờ kết quả trả về từ Server. Khi nhận được thông điệp từ RPC Runtime của Client gửi sang, RPC Runtime bên phía server chuyển thông điệp lên phần Server Stub.

**Server Stub** mở thông điệp ra xem, xác định hàm ở xa mà Client muốn thực hiện cùng với các tham số của nó. Server Stub gọi một lệnh tương ứng nằm trên phần Server. Khi nhận được yêu cầu của Server Stub, Server cho thực thi lệnh được yêu cầu và gửi kết quả thực thi được cho Server Stub.

Server Stub đưa kết quả thực vào một gói tin trả lời, chuyển cho phần RPC Runtime cục bộ để nó gửi sang RPC Runtime của Client

**Runtime** cục bộ để nó gửi sang RPC Runtime của Client. RPC Runtime bên phía Client chuyển gói tin trả lời nhận được cho phần Client Stub. Client Stub mở thông điệp chứa kết quả thực thi về cho Client tại vị trí phát ra lời gọi thủ tục xa.

Trong các thành phần trên, RPC Runtime được cung cấp bởi hệ thống. Client Stub và Server Stub có thể tạo ra thủ công (phải lập trình) hay có thể tạo ra bằng các công cụ cung cấp bởi hệ thống. Cơ chế RPC được hỗ trợ bởi hầu hết các hệ điều hành mạng cũng như các ngôn ngữ lập trình.

# Ưu điểm và nhược điểm

**Ưu điểm**

1. RPC cung cấp sự trừu tượng, ví dụ như “truyền tải thông điệp” trong giao tiếp mạng được ẩn khỏi người sử dụng.
2. RPC thường lược bỏ nhiều lớp thủ tục để tăng hiệu năng. Kể cả khi một tí hiệu năng nó đem lại cũng rất quan trọng vì một chương trình thường xuyên gọi RPC.
3. RPC cho phép ứng dụng có thể được sử dụng trong môi trường hệ phân tán chứ không phải mỗi môi trường cục bộ.
4. Code với RPC cũng giúp nỗ lực viết lại, thiết kế lại không gặp nhiều khó khăn và mất thời gian.
5. Kiến trúc hướng tiến trình và hướng luồng được hỗ trợ bởi RPC.

**Nhược điểm**

1. Cách chia nhiều hàm để gọi tồn tại một số hạn chế khi thời gian trễ mỗi lần gọi RPC là khó có thể bỏ qua, càng nhiều lần gọi, tổng thời gian trễ sẽ tăng, khả năng nghẽn cổ chai do kiểu hỏi đáp liên tục sẽ tăng.
2. Đối với lời gọi cục bộ, đối tượng gọi (caller) và đối tượng bị gọi (receiver) nằm trong cùng một process. Kiểu tham số truyền được kiểm tra nghiêm ngặt khi biên dịch. Còn với RPC, việc kiểm tra như thế sẽ dẫn đến nhiều rủi ro hơn, dữ liệu có thể bị nghe lén hoặc bị thay đổi trên đường truyền. Việc bảo mật lời gọi RPC dẫn đến cần phải mã hóa, gắn kèm chữ ký kiểm tra…điều này sẽ khiến thư viện bên dưới của caller và receiver sẽ phải làm việc nhiều hơn, độ trễ lại cao hơn.

# Giới thiệu về gRPC

gRPC là framework mã nguồn mở, hiện đại và hiệu năng cao cho việc tạo và sử dụng API RPC qua giao thức HTTP/2 và có thể chạy trên bất kỳ môi trường nào.

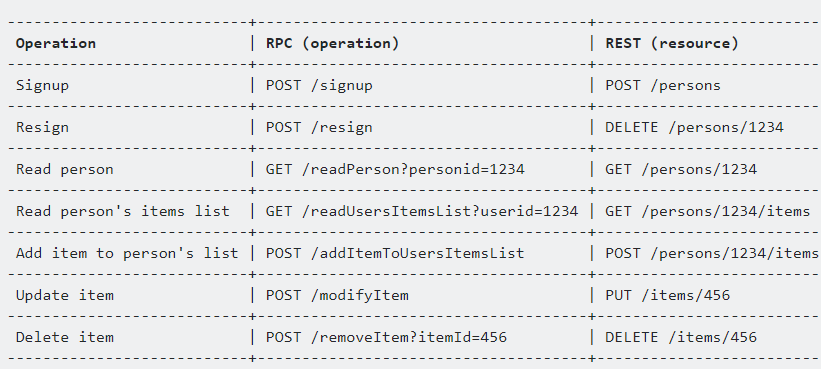
Truyền thống, có 2 cách tạo ra API: RPC và REST.

## Vì sao cần gRPC?

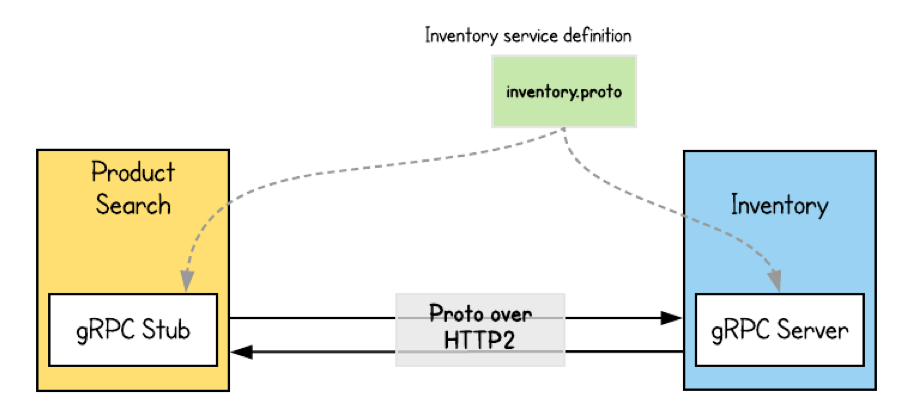
Dưới thời huy hoàng và phát triển rực rỡ từ REST API, cơ bản là giao tiếp giữa client và server đã được giải quyết khá tốt. Nhưng dưới thời đại Microservices, rõ ràng chúng ta cần một phương pháp tốt hơn để tăng tải và thông lượng giữa các services.

## RPC không phải là REST API

* REST API: Client và Server cần trao đổi state thông qua các resource được trả về. Do đó các response trả về thường là một resource.
* RPC: Client cần server thực hiện tính toán hoặc trả về một thông tin cụ thể nào đó. Bản chất giống y như ta đang gọi hàm, chỉ là hàm đó ở máy chủ khác hoặc service khác. Từ đó response trả về chỉ là kết quả của “hàm” thôi, không hơn, không kém.



## gRPC hoạt động như thế nào



Quay lại với câu chuyện tăng tải cho cả hệ thống nhiều services (hay Microservices), Google đã phát triển 2 thứ:

* Một giao thức mới để tối ưu các connection, đảm bảo dữ liệu đi trao đổi liên tục với ít băng thông nhất có thể.
* Một định dạng dữ liệu mới để 2 đầu service (hoặc client và server) có thể hiểu được các message của nhau mà ít phải encode/decode.

Đầu tiên Google phát triển một giao thức thay thế cho HTTP/1.1 với tên gọi SPDY. Sau này giao thức này được open source thậm chí chuẩn hoá, lấy làm nền móng cho giao thức HTTP/2. Khi có HTTP/2 rồi thì giao thức SPDY ngừng phát triển. gRPC chính thức hoạt động trên HTTP/2 luôn từ sau năm 2015.

HTTP/2 sẽ hoạt động rất tốt với binary thay vì là text. Vì thế Google phát minh kiểu dữ liệu binary mới với tên gọi: Protobuf (tên đầy đủ là Protocol Buffers).

# Chương trình minh họa

## Tạo file Protocol Buffer:

calculator.proto

syntax = "proto3";

option csharp\_namespace = "Server";

package calculate;

service Calculator {

rpc Add (IntFactorRequest) returns (ResultReply);

rpc Subtract (IntFactorRequest) returns (ResultReply);

rpc Multiply (IntFactorRequest) returns (ResultReply);

rpc Divide (IntFactorRequest) returns (ResultReply);

}

message IntFactorRequest {

int32 x = 1;

int32 y = 2;

}

message ResultReply {

int32 f = 1;

}

## Tạo service chứa các thủ tục gRPC cho máy chủ:

CalculatorService.cs

namespace Server

{

public class CalculatorService : Calculator.CalculatorBase

{

public override Task<ResultReply> Add(IntFactorRequest request, ServerCallContext context)

{

return Task.FromResult(new ResultReply

{

F = request.X + request.Y,

});

}

public override Task<ResultReply> Subtract(IntFactorRequest request, ServerCallContext context)

{

return Task.FromResult(new ResultReply

{

F = request.X - request.Y,

});

}

public override Task<ResultReply> Multiply(IntFactorRequest request, ServerCallContext context)

{

return Task.FromResult(new ResultReply

{

F = request.X \* request.Y,

});

}

public override Task<ResultReply> Divide(IntFactorRequest request, ServerCallContext context)

{

if (request.Y == 0) throw new ArgumentException("Divisor must not be zero.");

return Task.FromResult(new ResultReply

{

F = request.X / request.Y,

});

}

}

}

## Gọi thủ tục từ ứng dụng máy khách:

using var channel = GrpcChannel.ForAddress("https://localhost:5001");

var calculatorClient = new Calculator.CalculatorClient(channel);

var calculatorRequest = new IntFactorRequest() { X = 10, Y = 5 };

var addResponse = await calculatorClient.AddAsync(calculatorRequest);

var subtractResponse = await calculatorClient.SubtractAsync(calculatorRequest);

var multiplyResponse = await calculatorClient.MultiplyAsync(calculatorRequest);

var divideResponse = await calculatorClient.DivideAsync(calculatorRequest);

Console.WriteLine($"Calculator:\n" +

$"Addition: {calculatorRequest.X} + {calculatorRequest.Y} = {addResponse.F}\n" +

$"Subtraction: {calculatorRequest.X} - {calculatorRequest.Y} = {subtractResponse.F}\n" +

$"Multiplication: {calculatorRequest.X} \* {calculatorRequest.Y} = {multiplyResponse.F}\n" +

$"Divisor: {calculatorRequest.X} / {calculatorRequest.Y} = {divideResponse.F}\n");

Console.WriteLine("Press any key to exit...");

Text

Description automatically generated