GRPC UNTER ASP.NET CORE

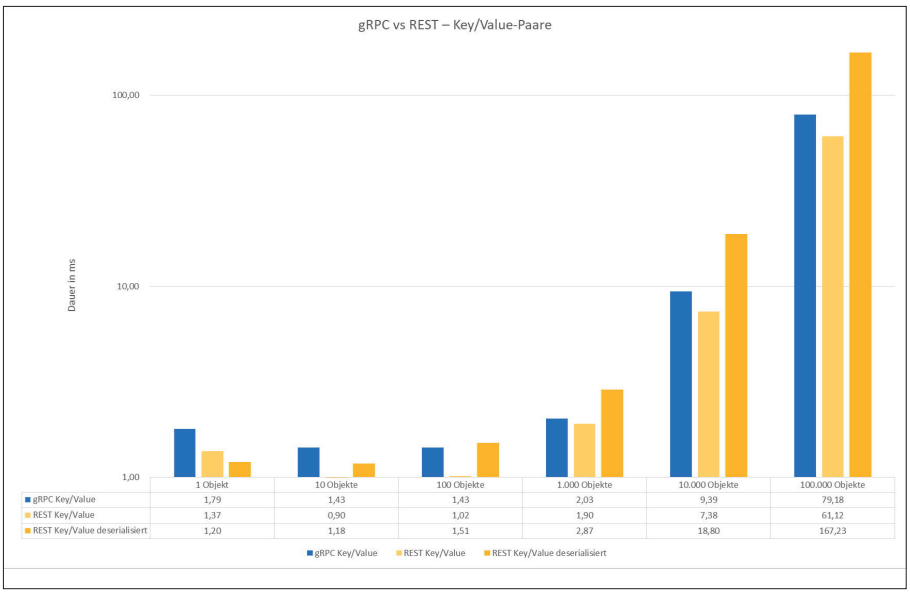
Ruf mich auf

Das von Google entwickelte Framework gRPC soll die Kommunikation zwischen Microservices verbessern und beschleunigen. Das geht auch unter .NET.

In verteilten Systemen müssen Rechner Informationen austauschen und Prozesse anstoßen. Das war schon zu Beginn der 1960er-Jahre so, als die ersten Ideen eines Computernetzwerks umgesetzt wurden, und das gilt heute mehr denn je. Das Konzept des Remote Procedure Call (RPC) und die ersten Implementierungen sind in den 70er- und 80er-Jahren entstanden [1].

Google hat für sein Rechenzentrum nicht auf gängige Technologien für die Interkommunikation setzen wollen und daher ein neues Protokoll und ein neues Framework zum Aufruf von Funktionen auf anderen Rechnern erstellt: gRPC.

Bei mehreren Milliarden Aufrufen pro Sekunde kommt es auf jede Nanosekunde an. Sonst ist die Performance nicht zufriedenstellend.



Benchmark gRPC vs. REST mit kleinen Objektgrößen (Bild 1)

Entscheidend für Googles Entwicklung von gRPC war, dass die CPU- und Bandbreitenlast bei der Service-to-Service-Kommunikation gegenüber anderen Alternativen wie REST, Message Bus oder anderen RPC-Alternativen wie JSON-over-REST, CORBA oder auch SOAP erheblich reduziert werden sollte.

Mit diesem intern erst mal Stubby genannten Framework entwickelte eine hochskalierbare, lose gekoppelte und performante Lösung, um die verschiedenen eingesetzten Technologien, Programmiersprachen und Komponenten in den verteilten Systemen zu verbinden [2].

Im August 2016 wurde die erste Version veröffentlicht und erhielt den Namen gRPC, was als rekursives Akronym für „gRPC Remote Procedure Call“ [3] steht. Böse Zungen behaupten natürlich, dass der Buchstabe *g* für den Hersteller stehen würde.

Mit jedem Minor-Release von gRPC steht nun das *g* für ein anderes Wort, beispielsweise für *green*, *gandalf* oder auch *gangnam*.

Dass diese Technologie noch längst nicht zu Ende implementiert ist, lässt sich am Release-Plan erkennen: Alle sechs Wochen wird ein neues Release veröffentlicht.

Schon direkt mit der ersten Version ist auch eine Implementierung für .NET zur Verfügung gestellt worden, sodass in eigenen Projekten mittels NuGet-Paketen gRPC verwendet werden kann.

Seit März 2017 ist gRPC ein offizielles Projekt der Cloud Native Computing Foundation. Andere prominente Projekte der Foundation sind beispielsweise Kubernetes, Helm oder Prometheus [4].

Gemeinsamkeiten von RPC-Systemen

RPC-Systeme ermöglichen den Aufruf anderer Prozesse. Diese können auf dem gleichen zugrunde liegenden System erfolgen oder auf anderen, entfernten Rechnern.

Der Aufruf von Funktionen auf anderen Systemen wird dabei von der Netzwerktechnologie entkoppelt. Die Schnittstellen für die Methodenaufrufe mit den Aufruf- und Antwort-Parametern werden in einer Art Interface Definition Language (IDL) – dem Contract – beschrieben, die unabhängig von der Programmiersprache ist und daher von verschiedenen Sprachen verwendet werden kann.

Ausgehend von diesen IDLs können mithilfe weiterer Tools Libraries für die serverseitige und clientseitige Implementierung (sogenannte Stubs) erstellt werden.

Die Art und Weise, wie diese Aufrufe nun transportiert werden, ist ein Implementierungsdetail. Einige verwenden hierbei TCP, andere setzen auf die Protokolle HTTP beziehungsweise REST auf. ▶

Listing 1: PROTO-Datei für Service

```
syntax = "proto3";

import "Proto/Messages/category.proto";

package services;

service Categories {
  rpc GetAllCategories (GetAllCategoriesRequest)
  returns (GetAllCategoriesResponse);
}

message GetAllCategoriesRequest {
}

message GetAllCategoriesResponse {
        repeated message.Category Categories = 1;
}
```

Architektur von gRPC

Die Architektur von gRPC besteht aus mehreren Schichten. Die unterste Schicht ist die Transportschicht. gRPC basiert auf HTTP/2 – entstanden aus dem Google-internen Projekt SPDY. Dieses Protokoll wurde aus vielerlei Gründen entwickelt. So unterstützt es eine vollduplex-bidirektionale Kommunikation. Das heißt, beide Kommunikationspartner können Daten senden und empfangen, und das zur gleichen Zeit. Im Gegensatz zu HTTP 1.x ist es kein auf Text basierendes Protokoll, sondern rein binär [5].

Der große Vorteil von HTTP/2 ist auch, dass in derselben aufgebauten Verbindung Anfragen und Antworten gebündelt werden können und somit auch starke Performance-Steigerungen möglich sind.

Auf der nächsthöheren Architekturschicht, dem Channel, wird das Mapping von RPC auf die unterliegende Schicht durchgeführt.

Diese Schicht stellt Methoden für einen blockierenden, asynchronen oder auch streamenden Aufruf bereit.

Die dritte Schicht wird anhand der IDL durch ein Generierungstool erstellt. Dieser sogenannte Stub – ein Platzhalter für die Zugriffe auf der Empfängerseite – stellt Methoden mit den entsprechenden Methodenparametern bereit, die dann von einem Client aus verwendet werden können.

Dieser Stub verwendet diejenigen Methoden, die im Channel implementiert sind. Nur in dieser Schicht spielt die IDL eine Rolle [6].

Protocol Buffer: das Kommunikationsformat

Als Format für die Übertragung von Daten wird Protocol Buffer (protobuf) verwendet. Dieses binäre Format ist bei der Serialisierung und Deserialisierung wesentlich performanter als beispielsweise XML oder JSON, da es die entsprechenden Nachrichten verdichtet [7].

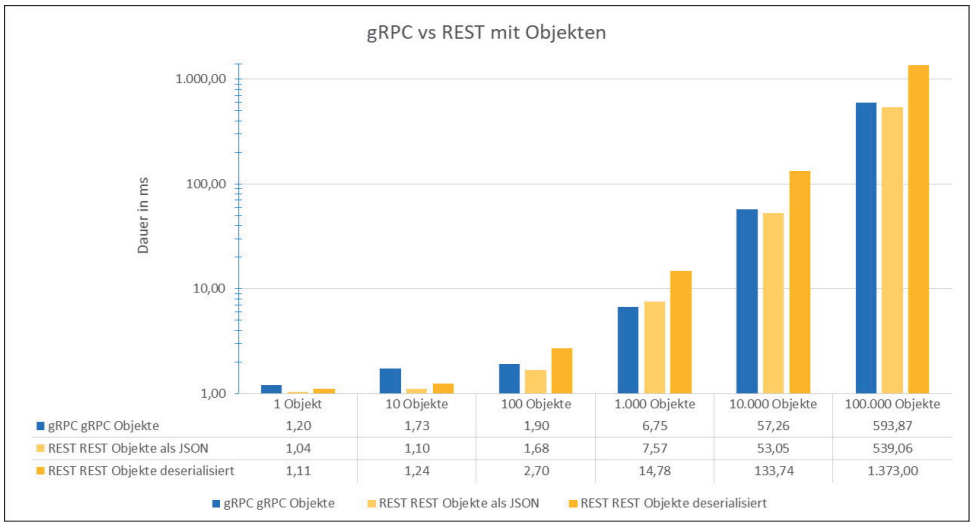
Protocol Buffer können auch unabhängig von RPC-Systemen verwendet werden. Die Definition, wie diese Nachrichten aufgebaut und verdichtet werden sollen, erfolgt in einer Interface Definition Language, den sogenannten PROTO-Dateien. Eine PROTO-Datei sieht beispielsweise so aus (*category.proto*):

```
syntax = "proto3";

package message;

option csharp_namespace = "GrpcWorld.Entities";

message Category {
        int32 Id = 1;
        string CategoryName = 2;
        string Description = 3;
}
```



Benchmark gRPC vs. REST mit großen Objektgrößen (Produkt mit Hersteller und Kategorie) (Bild 2)

Der Wert *proto3* in *syntax* beschreibt die aktuelle Version des Protocol Buffer. Auch wenn die Vorgängerversion *proto2* teilweise noch unterstützt wird, sollte diese nicht mehr verwendet werden.

Auffällig ist die Angabe der Zahlen: Die Feldbezeichnungen wie *Id* und *CategoryName* werden nicht mit serialisiert. Daher bedarf es dieser Field Numbers.

Diese müssen eindeutig sein und definieren das Feld im serialisierten Binärformat.

RPC-Services, also diejenigen Methoden, die vom Client aus

aufgerufen werden können, werden ebenfalls in den PROTO-Dateien definiert, wie in Listing 1 angegeben. In dieser PROTO-Datei wurde *category.proto* importiert, damit *Category* als *Messagetype* bekannt gemacht wird.

Aus dieser PROTO-Datei können dann für verschiedene Sprachen die entsprechenden Klassen erstellt werden.

Das gilt sowohl für die Message-Klassen (in diesem Beispiel *Category* und die Request/Response-Klassen) als auch für die Klasse für die Server-Implementierung und den entsprechenden Stub für den Client.

Koexistenz von gRPC for C# und gRPC for .NET

Eine Unterstützung für C# (GitHub: *grpc/grpc*) als Sprache neben diversen anderen existiert bereits seit dem ersten Release von gRPC [8].

Im September 2019 hat Microsoft mit dem ersten Release von gRPC for .NET (GitHub: *grpc/grpc-dotnet*) nachgelegt und eine bessere Integration der bereits verfügbaren gRPC-NuGet-Packages in eigene ASP.NET-Core-Projekte implementiert. Mit anderen Worten: gRPC for .NET ersetzt nicht die anderen Libraries, sondern verwendet diese für die Erstellung der Clients, Server und Infrastruktur. Der Compiler und die Integration in MS Build sind weiterhin im gRPC-Hauptprojekt [9] enthalten. Sie folgt sogar dem sechswöchigen Release-Plan, sodass auch zukünftig mit weiteren neuen Features gerechnet werden kann.

Erstellung eines gRPC-Projekts über die Konsole

In Visual Studio 2019 sind die Projekttypen für die Erstellung von gRPC-Services integriert. Auf der Konsole kann das Projekt wie folgt angelegt werden:

```
dotnet new grpc -n GrpcExample
```

Hiermit wird das Template *ASP.NET Core gRPC Service* verwendet und ein Projekt namens *GrpcExample* angelegt. Standardmäßig wird auch eine *greet.proto*-Datei angelegt und entsprechende Server- und -Client-Implementierungen abgelegt. Mit

```
dotnet run -p GrpcExample\GrpcExample.csproj
```

kann der erstellte gRCP-Service direkt gestartet werden.

Damit ASP.NET Core auch gRPC-Aufrufe richtig zuordnen kann, werden in *Startup.cs* mit *services.AddGrpc()* aus dem *Grpc.AspNetCore*-Paket die Services registriert.

Die korrekte Zurordnung der gRPC-Calls zu den implementierten Methoden erfolgt dann über die Registrierung der Middleware-Endpoints:

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapGrpcService<GreeterService>();
});
```

Sind bereits PROTO-Dateien vorhanden, so können diese über die CLI wie folgt hinzugefügt werden:

```
dotnet grpc add-file category.proto
```

Mit diesem Aufruf wird auch automatisch in der *.csproj*-Datei folgender Eintrag erstellt:

```
<ItemGroup>
  <Protobuf Include="category.proto" />
</ItemGroup>
```

Dieser Eintrag sorgt wiederum dafür, dass Basisklassen für die Server- und Client-Implementierung generiert werden. Das ist auch der Standardfall.

Verantwortlich für diesen Eintrag in der *.csproj*-Datei ist das .NET-Core-Tool *dotnet-grpc*. Wenn es noch nicht installiert ist, kann dies auf der Konsole nachgeholt werden: ▶

Listing 2: Ein simpler Category-Service mit generierter CategoriesBase als Basisklasse

```
public class CategoryService :
  Categories.CategoriesBase
{
  private readonly ICategoryRepository
    _categoryRepository;

  public CategoryService(
    ICategoryRepository categoryRepository)
  {
    _categoryRepository = categoryRepository;
  }

  public override async Task<GetAllCategoriesResponse>
    GetAllCategories(
      GetAllCategoriesRequest request,
      ServerCallContext context)
  {
    var all =
      await _categoryRepository.FetchAllAsync();

    var response = new GetAllCategoriesResponse();

    response.Categories.AddRange(
      all.Select(p => p.MapToMessage()));

    return response;
  }
}
```

```
dotnet tool install -g dotnet-grpc
```

Sollen bestimmte Implementierungen nicht automatisch erstellt werden, so kann das im entsprechenden Eintrag festgelegt werden:

```
<Protobuf Include="category.proto"
  GrpcServices="None" />
```

GrpcServices kann dabei die Werte *Server*, *Client*, *Both* (Standardfall) oder *None* annehmen. In Visual Studio kann dieser Eintrag auch über die Dateieigenschaft der PROTO-Datei (mit Rechtsklick auf die Datei) festgelegt werden.

Die Integration von gRPC in Visual Studio kann als gelungen bezeichnet werden. So führt jegliche Änderung in den PROTO-Dateien sofort zur (erneuten) Erstellung der Basisimplementierungen.

Falls dies nicht auf Anhieb funktionieren sollte, kann es an dem falschen Eintrag *Build Action* der Dateieigenschaften liegen: Er muss auf *Protobuf Compiler* stehen. Nach der Erstellung eines neuen gRPC-Projekts sollte auch ein Update des NuGet-Pakets *Grpc.AspNetCore* durchgeführt werden, um den aktuellen Stand zu haben.

Implementierung der Services

In Listing 1 ist eine PROTO-Datei für einen Category-Service aufgeführt. Der Protobuf Compiler erstellt im Hintergrund eine statische Klasse *Categories* im Namespace *Services*, angegeben durch die Bezeichnung in *packages*.

Nun kann die generierte Klasse *CategoriesBase*, welche in der statischen Klasse *Categories* enthalten ist, als Basisklasse für die Server-Implementierung verwendet werden, wie in Listing 2 angegeben.

Listing 3: Ein Client mit Zugriff auf den gRPC–Service

```
public async Task FetchCategoriesExampleCall()
{
  var channel = GrpcChannel.ForAddress(
    "https://localhost:5001");

  var client =
    new Categories.CategoriesClient(channel);

  var request = new GetAllCategoriesRequest();

  var replies =
    await client.GetAllCategoriesAsync(request);

  var categories = replies.Categories.ToList();

  Console.WriteLine(
    $"Received {categories.Count} categories.");
}
```

Listing 4: Implementierung von CategoryService

```
public override async Task CreateNewCategory(
  IAsyncStreamReader<CreateNewCategoryRequest>
    requestStream,
  IServerStreamWriter<CreateNewCategoryResponse>
  responseStream, ServerCallContext context)
{
  var categoryId = 1;
  while (await requestStream.MoveNext())
  {
    _logger.LogInformation($"New Category
      {requestStream.Current.CategoryName}.");
    await responseStream.WriteAsync(
      new CreateNewCategoryResponse
        {CategoryId = categoryId++});
  }

  _logger.LogInformation(
    $"Last CategoryId {categoryId}.");
}
```

Hier zeigt sich die Besonderheit von gRPC for .NET: Durch die Integration in ASP.NET-Core-Projekte sind Features wie Dependency Injection (DI), Protokollierung sowie Authentifizierung und Autorisierung mit enthalten.

Dieser Category-Service ist nun Teil der Middleware-Pipeline des ASP.NET-Core-Projekts. Durch Constructor Injection können dann weitere Instanzen des Loggers, Repositories oder Services injiziert werden.

In diesem Beispiel wurde ein *CategoryRepository* mit übergeben, das Testdaten bereithält.

Die in diesem Artikel verwendeten Beispiele sind zur weiteren Verwendung auf GitHub [10] abgelegt.

Aufruf der Services: Implementierung des Clients

Die Client-Implementierung für den Zugriff auf die Services gestaltet sich simpel. Als NuGet-Paket wird *Grpc.Net.Client* verwendet; eine Referenz auf das Projekt, welches den generierten Client enthält, wird hinzugefügt. In Listing 3 ist ein Beispielcode für den Zugriff auf den gRPC-Service angegeben.

Der verwendete Port 5001 ist hierbei der Default eines ASP.NET-gehosteten Service und sollte in einer konkreten Implementierung mit übergeben werden. Die Erzeugung einer Channel-Instanz über *GrpcChannel.ForAddress* kann aus Performance-Gründen einmalig passieren und für verschiedene Clients verwendet werden.

Streaming

Durch das zugrunde liegende HTTP/2-Protokoll unterstützt gRPC verschiedene Arten des Streamings von Daten [11]:

- Unär (kein Streaming)
- Server Streaming

- Client Streaming
- Bidirektionales Streaming

Welche Art und welche Kombination von Streaming verwendet werden soll, ergibt sich aus der PROTO-Datei.

Die jeweiligen Streamingvarianten werden in der Service-Definition der PROTO-Datei mit angegeben:

```
rpc CreateNewCategory(stream CreateNewCategoryRequest)
        returns (stream CreateNewCategoryResponse);
```

In der serverseitigen Implementierung enthält die Basisklasse die entsprechende Methodensignatur, die dann überschrieben werden kann, wie das beispielhaft in Listing 4 angegeben ist.

Wie gut zu erkennen, wird – solange der Client neue Nachrichten streamt – eine entsprechende neue Nachricht auf dem gleichen Kanal zurückgeschrieben.

Performance-Vergleich gRPC und Web API

Kann gRPC mit Web API verglichen werden? Sind das nicht zwei unterschiedliche Paradigmen und Anwendungsfälle?

Mit RPC werden Operationen zur Verfügung gestellt, die auf bestimmte Daten angewandt werden sollen. REST hingegen stellt Daten als Ressourcen zur Verfügung, die der Client weiter verwenden kann.

Allerdings verschwimmen in der Praxis diese Grenzen, sodass ein Performance-Vergleich für bestimmte Szenarien nützlich ist. Um einen ersten Eindruck von der Reaktions- und Ausführungszeit dieser beiden Technologien zu bekommen, wurde ein Benchmarktest erstellt.

Dieser Test spiegelt nicht unbedingt die realen Anforderungen im Praxisbetrieb wider, sondern soll aufzeigen, ob mit gRPC auf Anhieb starke Performance-Steigerungen zu erwarten sind.

Für die folgenden Benchmarktests wurden verschiedene Szenarien als Grundlage angenommen:

- Key/Value-Paare als kleines Objekt mit ID, Name und Description (Objekt-Kategorie)
- Objekte mit mehreren Properties unterschiedlicher einfacher Datentypen, einschließlich zweier weiterer Unterobjekte (Produkt mit Hersteller und Kategorie)

Die Anfragen vom Client erfolgten so, dass in mehreren Durchläufen eine unterschiedliche Menge von Objekten angefordert wurde.

In den Diagrammen ist dies in der horizontalen Achse aufgeführt. Die vertikale Achse zeigt die Dauer der Anfrage bis zu einer Antwort in logarithmischer Darstellung.

Die Server- und Client-Implementierungen, die als Grundlage dieser Tests dienen, sind ebenfalls im Repository auf GitHub enthalten.

Im Unterschied zu REST kann der Client bei gRPC direkt auf die Objekte zugreifen, was bedeutet, dass diese bereits deserialisiert sind.

Daher wurde der Benchmark bei REST aufgeteilt in zwei Tests: einmal ein Test mit Deserialisierung und einmal einer ohne. Das bedeutet, es wurde nur der JSON-Text aus der Antwortnachricht extrahiert.

Bei diesen Ergebnissen waren sowohl Client als auch Server auf der gleichen Maschine. Ähnliche Ergebnisse wurden bei Zugriffen auf entfernte Server-Instanzen im gleichen Netzwerk erzielt.

Den großen Geschwindigkeitsvorteil kann gRPC bei größeren Datenmengen ausspielen, bei kleineren liegen beide Varianten etwa gleichauf.

Fazit

Microsoft bietet mit ASP.NET Core 3.x eine stärkere Integration der gRPC-NuGet-Pakete und auch der Projektvorlagen an. Somit werden Entwickler besser bei der Entwicklung von gRPC-Diensten unterstützt.

Dank der automatischen Generierung von server- und clientseitigem Code bei der Einbindung von PROTO-Dateien ist die Implementierung sehr einfach. Nicht nur im Web- oder Microservices-Umfeld, sondern auch bei der Desktop-Entwicklung kann gRPC verwendet werden.

Die Performance-Tests haben ergeben, dass gRPC und Web API in ihren Übertragungsgeschwindigkeiten unter ASP.NET Core 3.x sehr nah beieinander liegen.

Allerdings zeigen sich bei größeren Datenmengen die Vorteile von gRPC, da hier der zeitintensive Schritt der Deserialisierung entfällt. ■

[1] Bruce Nelson, Andrew Birell, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, 1984, www.dotnetpro.de/SL2007gRPC1

[2] gRPC: a true internet-scale RPC framework, www.dotnetpro.de/SL2007gRPC2

[3] 'g' stands for something different, www.dotnetpro.de/SL2007gRPC3

[4] Cloud Native Computing Foundation To Host gRPC from Google, www.dotnetpro.de/SL2007gRPC4

[5] Hypertext Transfer Protocol Version 2, www.dotnetpro.de/SL2007gRPC5

[6] Joshua Humphries et al, Practical gRPC, Kapitel 2, www.dotnetpro.de/SL2007gRPC6

[7] Encoding, www.dotnetpro.de/SL2007gRPC7

[8] gRPC C#, www.dotnetpro.de/SL2007gRPC8

[9] https://github.com/grpc/grpc

[10] GrpcWorld, www.dotnetpro.de/SL2007gRPC9

[11] Call gRPC services with the .NET client, www.dotnetpro.de/SL2007gRPC10

André Munk-Wendlandt
ist Diplom-Informatiker und arbeitet als Softwareentwickler und -architekt bei der adesso SE. Seit 2001 entwickelt er leidenschaftlich Software, am liebsten im .NET-Umfeld. Er hilft Firmen bei der Appentwicklung und der Verbesserung der Softwarequalität.

dnpCode A2007gRPC