UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ERICSON JOSÉ DA SILVA SOARES
RAPHAEL DE CARVALHO ALMEIDA
VITOR AUGUSTO DA SILVA VASCONCELLOS

aRPC (Antenna RPC): Um *framework* de chamada de procedimento remoto (RPC) para uso em computação de alto desempenho (HPC)

RIO DE JANEIRO
2021

ERICSON JOSÉ DA SILVA SOARES
RAPHAEL DE CARVALHO ALMEIDA
VITOR AUGUSTO DA SILVA VASCONCELLOS

aRPC (Antenna RPC): Um *framework* de chamada de procedimento remoto (RPC) para uso em computação de alto desempenho (HPC)

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Gabriel Pereira Silva

RIO DE JANEIRO
2021

Soares, Ericson José da Silva
S676a aRPC (Antenna RPC): um framework de chamada de procedimento remoto (RPC) para uso em computação de alto desempenho (HPC) / Ericson José da Silva Soares, Raphael de Carvalho Almeida, Vitor Augusto da Silva Vasconcellos. – 2021.

88 f.

Orientador: Gabriel Pereira Silva.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2021.

1. Chamada de Procedimento Remoto. 2. Sistemas Distribuídos. 3. QUIC. I. Almeida, Raphael de Carvalho. II. Vasconcellos, Vitor Augusto da Silva. III. Silva, Gabriel Pereira (Orient.). IV. Universidade Federal do Rio de Janeiro, Instituto de Matemática. V. Título.

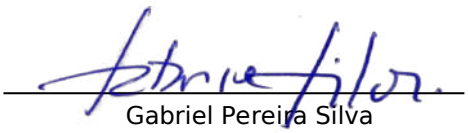
ERICSON JOSÉ DA SILVA SOARES
RAPHAEL DE CARVALHO ALMEIDA
VITOR AUGUSTO DA SILVA VASCONCELLOS

aRPC (Antenna RPC): Um *framework* de chamada de procedimento remoto (RPC) para uso em computação de alto desempenho (HPC)

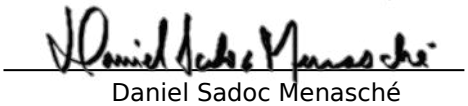
Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 21 de julho de 2021

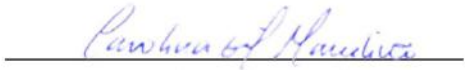
BANCA EXAMINADORA:



Gabriel Pereira Silva
D.Sc. (UFRJ)



Daniel Sadoc Menasché
Ph.D. (UFRJ)



Carolina Gil Marcelino
D.Sc. (UFRJ)

Dedicamos este trabalho para nossos amigos do grupo *Code Fairy*, que sempre foram fontes de inspiração e incentivo.

AGRADECIMENTOS

Agradecemos ao nosso orientador Gabriel pela paciência e incentivo para a realização deste trabalho. Bem como a todos os professores do Departamento de Ciência da Computação da UFRJ que contribuíram para a nossa formação técnica. Agradecemos também aos nossos familiares e amigos pela paciência e suporte durante essa jornada.

"I'm a scientist;
because I invent, transform, create, and destroy for a living,
and when I don't like something about the world,
I change it."

Rick Sanchez

RESUMO

A crescente adoção da arquitetura de microsserviços e a necessidade de comunicação entre linguagens distintas estimulou o desenvolvimento de novas soluções para a chamada de procedimento remoto (RPC). A diversidade de necessidades e propósitos resultou em uma variedade de implementações de RPC: algumas com foco na ergonomia de *software*; outras na abrangência de linguagens e funcionalidades; e, finalmente, uma parcela visando a eficiência em computação de alto desempenho (HPC). Nesse sentido, é apresentado neste trabalho um *framework* de RPC, de nome antena RPC (aRPC), com ênfase tanto no desempenho como na ergonomia de *software*, inspirado no *framework* gRPC, e que faz uso de novos serializadores e do protocolo de transporte QUIC para comunicação. Nas avaliações efetuadas, o aRPC obteve desempenho superior ao gRPC nos casos com grande quantidades de elementos nas estruturas de dados e quando os dados são mais heterogêneos e menos sintéticos. O *framework* proposto consegue ser até 7% mais rápido em relação ao gRPC, desde que as premissas descritas sejam respeitadas. Em situações com perda frequente de pacotes ou em redes de baixa qualidade, o aRPC possui desempenho muito superior ao gRPC, sendo até três vezes melhor no teste de vazão. Os resultados do aRPC abrem um campo de aplicação em sistemas de computação de alto desempenho e a resiliência apresentada faz com que seja uma opção interessante nos ambientes de IoT. Em termos gerais, o aRPC é competitivo quando comparado ao gRPC no contexto de HPC. Entretanto, o protocolo gRPC apresenta melhor desempenho para estruturas de dados mais simples e menos heterogêneas e para volumes de dados reduzidos.

Palavras-chave: arpc; antenna remote procedure call; rpc; chamada remota de procedimento; remote procedure call; quic; hpc; high performace computing; computação de alto desempenho; colfer; serialização; sistemas distribuídos; thrift.

ABSTRACT

The growing adoption of microservices architecture and the need for communication between distinct languages has stimulated the development of new solutions for remote procedure call (RPC). The diversity of needs and purposes has resulted in a variety of RPC implementations: some focusing on software ergonomics; others on the range of languages and functionalities; and, finally, a portion aiming at efficiency in high performance computing (HPC). In this sense, this work presents an RPC framework, named antenna RPC (aRPC), with emphasis on both performance and software ergonomics, inspired by the gRPC framework, and that makes use of new serializers and the QUIC transport protocol for communication. In the evaluations performed, aRPC outperformed gRPC in cases with large amounts of elements in the data structures and when the data is more heterogeneous and less synthetic. The proposed framework can be up to 7% faster than gRPC, provided the assumptions described are respected. In situations with frequent packet loss or in low quality networks, aRPC performs much better than gRPC, being up to three times better in the throughput test. The results of aRPC open a field of application in high-performance computing systems and its resilience makes it an interesting option in IoT environments. Overall, aRPC is competitive when compared to gRPC in the context of HPC. However, the gRPC protocol performs better for simpler and less heterogeneous data structures and for small data volumes.

Keywords: arpc; antenna remote procedure call; rpc; remote procedure call; quic; hpc; high performace computing; colfer; serialization; distributed systems; thrift.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

aRPC	*Antenna Remote Procedure Call*
RPC	*Remote Procedure Call* - Chamada de Procedimento Remoto
HPRPC	*High Performance Remote Procedure Call*
HPC	*High Performance Computing*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Secure Hyper Text Transfer Protocol*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
IP	*Internet Protocol*
TLS	*Transport Layer Security*
LAN	*Local Area Network*
IDL	*Interface Definition Language*
AST	*Abstract Syntax Tree*
RTT	*Round-Trip Time*
CLI	*Command Line Interface*
IETF	*Internet Engineering Task Force*
TI	Tecnologia da Informação
UTC	*Universal Time Coordinated*
UTF-8	Unicode (*or Universal Coded Character Set*) *Transformation Format* – 8-bit
I/O	*Input/Output*
IoT	*Internet of Things*

SUMÁRIO

1 INTRODUÇÃO

De uma maneira geral, sistemas computacionais são compostos por dados e por procedimentos que operam nesses dados. É comum que tais sistemas sejam executados inteiramente por um único computador. Entretanto, conforme as demandas de processamento e de novas funcionalidades vão crescendo, executar e escalonar esses sistemas em uma única máquina começa a se tornar uma tarefa inviável.(NEUMAN, 1994).

Para atender à demanda de crescimento são aplicados conceitos de sistemas distribuídos, onde partes distintas do sistema são executadas em computadores independentes, que se comunicam através de uma rede de interconexão. Um grande exemplo é a *World Wide Web*, onde existem duas entidades principais: o cliente executado num navegador, que é responsável por receber a entrada de dados do usuário, requisitar as páginas para o servidor, exibir o seu conteúdo para o usuário, etc. E o servidor, sendo executado numa máquina remota, recebendo as requisições feitas pelo usuário, processando os dados, cuidando de operações em bancos de dados e emitindo uma resposta para o cliente.

Existem diferentes metodologias para a implementação de sistemas distribuídos, tais como a troca de mensagens, produtor-consumidor e a chamada de procedimento remoto (RPC - *Remote Procedure Call*), que é a metologia abordada neste trabalho.

A construção de sistemas distribuídos utilizando o paradigma de RPC é um tema estudado há décadas (NELSON, 1981). Com o advento recente da computação em nuvem, o RPC retomou sua notoriedade. Hoje diversas empresas realizam a comunicação entre seus sistemas utilizando *frameworks* de RPC, tais como o gRPC, Thrift, Avro, entre outros.

A proposta deste trabalho é o desenvolvimento de um *framework* RPC com foco em computação de alto desempenho, o antena RPC (aRPC), e avaliar o seu desempenho comparando com alternativas já consolidadas no mercado. Este trabalho foi dividido nos seguintes capítulos: referencial teórico e projetos correlatos; descrição da proposta; método experimental e análise de resultados; e conclusões e trabalhos futuros.

Para o desenvolvimento de um novo *framework* de RPC, primeiro foi necessário o estudo do funcionamento de alguns dos *frameworks* de RPC modernos, bem como a revisão da literatura existente sobre o tema. Esses tópicos são apresentados no capítulo 2 que detalha o referencial teórico e projetos correlatos.

No capítulo 3, que descreve a proposta são apresentados detalhes sobre a arquitetura adotada, além da análise de alternativas levantadas em cada uma das camadas de implementação do *framework* e seus componentes. São apresentados também detalhes de implementação e eventuais dificuldades e soluções encontradas durante o desenvolvimento do projeto.

No capítulo 4 é evidenciado o método experimental e análise de resultados é feita

uma comparação entre os resultados observados em algumas aplicações construídas, tanto com o protocolo aRPC e com outros *frameworks* alternativos, detalhando vantagens e desvantagens descobertas em cada caso.

Por último, no capítulo 5, são apresentadas as conclusões do trabalho, abordando os insumos coletados nos capítulo anteriores, destacando os prós e contras do aRPC, bem como a indicação de possíveis trabalhos futuros e acréscimos para o *framework*.

2 REFERENCIAL TEÓRICO E PROJETOS CORRELATOS

A chamada de procedimento remoto (RPC - *Remote Procedure Call*) é um paradigma de projeto de sistemas distribuídos que permite que duas ou mais entidades se comuniquem através de um canal de comunicação por meio de uma especificação padronizada de requisições e respostas. A definição de RPC evoluiu através das décadas, se referindo inicialmente apenas a um par cliente-servidor (NELSON, 1981), até sua definição atual de um grupo de serviços interconectados trocando dados independentes de linguagem ou plataforma (SLEE; AGARWAL; KWIATKOWSKI, 2007).

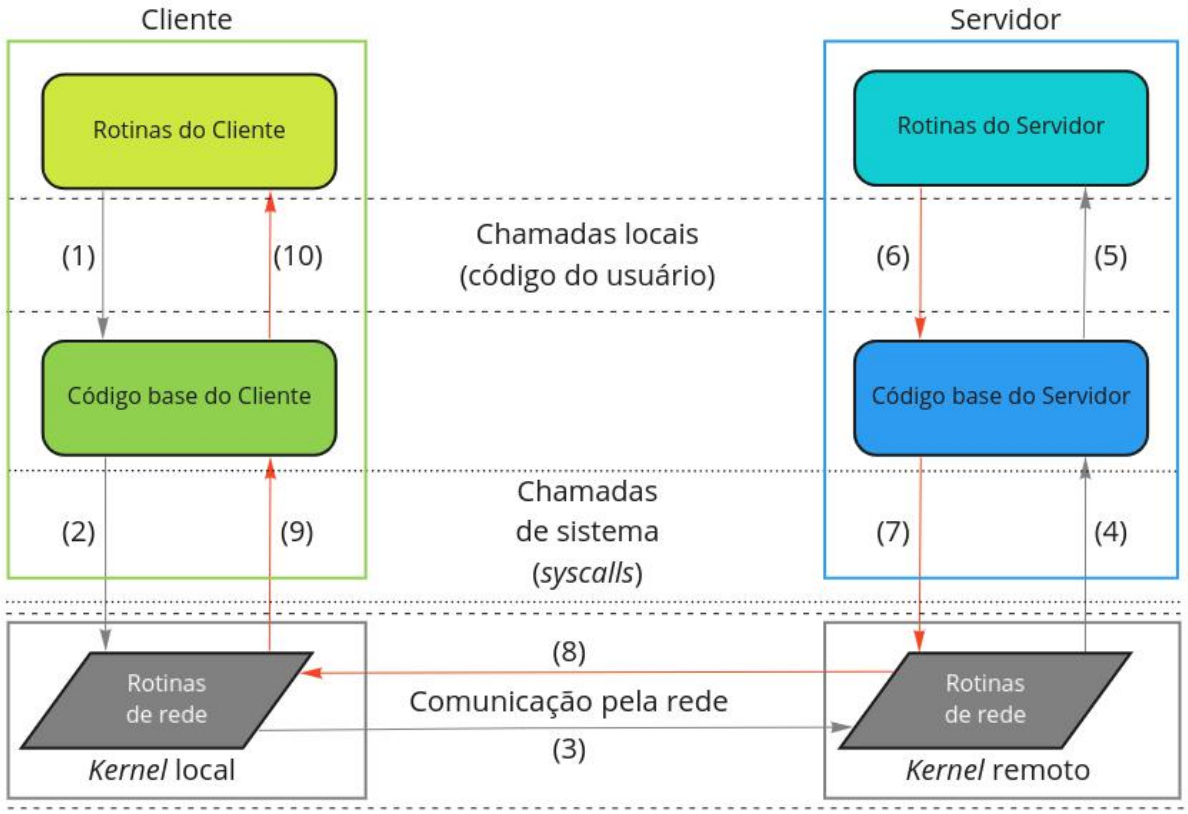
Sobre o aspecto da progressão, o RPC originou-se como um sistema síncrono de requisições e respostas, onde o cliente ficava esperando a resposta retornar, sem executar nenhuma outra tarefa nesse meio tempo. Além disso, existiam problemas de resiliência e segurança, pois não haviam mecanismos para evitar perdas de pacotes e os dados trafegavam sem nenhuma tipo de criptografia envolvida. Contudo, com seu desenvolvimento, surgiram meios para a realização de requisições assíncronas, permitindo que o cliente continue executando outras tarefas enquanto o servidor processa a resposta, além de melhorias da segurança através da inclusão da criptografia nas camadas de transporte de rede, como o TLS (*Transport Layer Security*). A resiliência na entrega de dados também evoluiu, com uso do TCP ou de implementações de meios de garantia de entrega na própria camada de aplicação da rede.

2.1 O QUE É UM RPC?

Uma visão mais simplificada, de alto nível, pode definir o RPC como dois *endpoints* de comunicação conectados pela rede, onde um lado emite requisições, enquanto o outro lado gera respostas a partir das requisições. É um paradigma de requisição e resposta onde os dois *endpoints* possuem espaços de endereçamento de memória distintos. O *endpoint* que emite as requisições muitas vezes é chamado de *caller*, enquanto o outro que gera as respostas é chamado de *callee*. O RPC se diferencia de outros modelos de requisição e resposta como a *world wide web*, por exemplo, pois é mais voltado para a integração de sistemas.

Essa visão simplificada pode ser conferida na Figura 1. Nessa representação, tem-se um cliente, que é o *caller*, e o servidor, que é o *callee*, separados por uma rede física. Existe uma separação lógica para a aplicação, onde o código implementado pelo desenvolvedor é executado apenas nas camadas superiores, compondo as rotinas do cliente e do servidor. Essas rotinas, por sua vez, chamam procedimentos implementados pelo *framework*, representados na Figura 1 pelo conjunto dos blocos **código base do cliente** e **código base do servidor**. Vale ressaltar que no bloco **código base do cliente** estão expostas

Figura 1 – Diagrama simplificado de cliente e servidor



funções que representam as rotinas a serem executados no servidor, de modo a tornar a utilização do RPC transparente para o desenvolvedor, apenas reproduzindo o comportamento de funções executadas localmente. O **código base do servidor** é responsável por registrar os procedimentos remotos escritos pelo desenvolvedor, de modo que o cliente consiga enviar suas requisições para cada um deles de maneira independente. Esse código base também é responsável por interagir com o sistema operacional, de modo a utilizar as funcionalidades do *kernel* para preparar os dados e enviá-los pela rede. As setas cinzas representam o percurso dos dados do cliente, enquanto as laranjas representam a resposta do servidor. A ordem de ocorrência de cada etapa da comunicação está numerada na figura para facilitar o entendimento do leitor.

Os *endpoints* num RPC podem ser nomeados de diversas formas: cliente e servidor; nós numa rede *peer-to-peer*; *hosts* de sistema computacional em *grid*; ou até mesmo como microsserviços. Nesse modelo não há restrição para que sejam somente duas entidades, sendo que múltiplos *endpoints* se comunicando são suportados(BERGSTROM; PANDEY, 2007).

Conforme descrito em (BIRRELL; NELSON, 1984), originalmente a ideia de RPC foi desenvolvida com um mecanismo síncrono de requisições e respostas, amarrado a uma linguagem de programação específica e utilizando um protocolo de rede próprio, para

que a computação a ser realizada pudesse ser exportada do cliente para o servidor. O objetivo desse sistema era proporcionar a chamada de funções numa máquina remota, com um espaço de endereçamento diferente, a partir das mesmas semânticas encontradas para executar funções localmente. O cliente e o servidor executavam uma única *thread* e funcionavam de maneira síncrona, o cliente enviava uma requisição e ficava bloqueado esperando a resposta do servidor. Tal RPC foi desenvolvido por uma equipe de pesquisa de *internetwork* da empresa Xerox, na década de 1980 e era executado numa LAN pequena e fechada.

Figura 2 – RPC síncrono

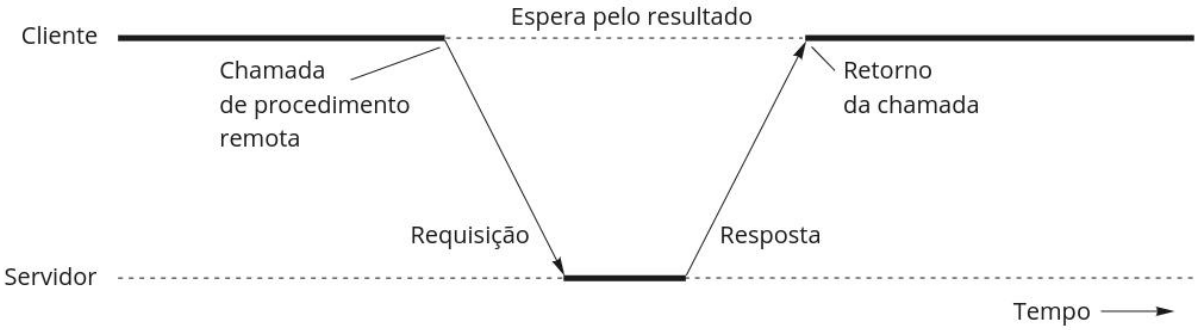
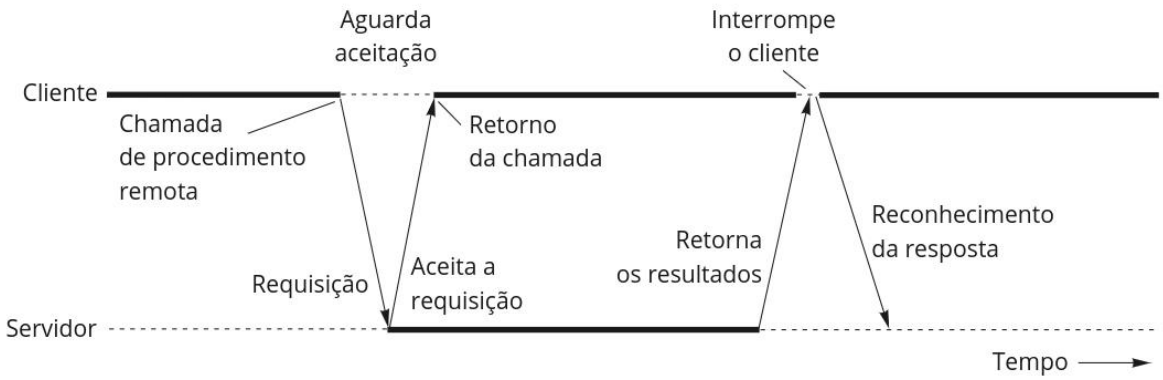


Figura 3 – RPC assíncrono



A comparação entre modos síncrono e assíncrono de RPC é melhor observada nas Figuras 2 e 3, respectivamente. Enquanto o cliente fica parado esperando no modo síncrono, no modo assíncrono uma resposta de aceitação da requisição é emitida instantaneamente, e no futuro quando a requisição for de fato processada, o servidor enviará a resposta para o cliente. Nesse intervalo o cliente fica livre para executar outras tarefas em paralelo e, ao receber a resposta, ele retorna para o fluxo anterior.

Outro aspecto importante dos *frameworks* de RPC modernos foi a introdução de uma camada composta por um compilador e uma IDL (*Interface Definition Language*), contendo arquivos que descrevem textualmente o formato e os tipos dos dados que serão trafe-

gados pela rede, bem como os procedimentos que operam esses dados (SLEE; AGARWAL; KWIATKOWSKI, 2007). Além de servir como uma documentação viva das entradas e saídas do sistema, ainda é possível empregar ferramentas de geração de código, que transformarão os tipos declarados nos arquivos de IDL nos códigos de base do cliente e do servidor automaticamente, de maneira a permitir que o desenvolvimento de aplicações RPC seja mais ágil e simples.

2.2 GRPC

O protocolo gRPC foi criado pela Google e liberado como um sistema de código aberto em 2015, sendo desenvolvido com o objetivo de permitir a comunicação entre diversos microsserviços utilizados internamente pela empresa. Atualmente serve também como porta de acesso aos diversos serviços externos para os clientes da sua plataforma de computação em nuvem. A comunicação é feita utilizando o protocolo HTTP/2, tendo como serializador e IDL a biblioteca Protobuffers, criada pela mesma empresa.

2.2.1 Arquitetura

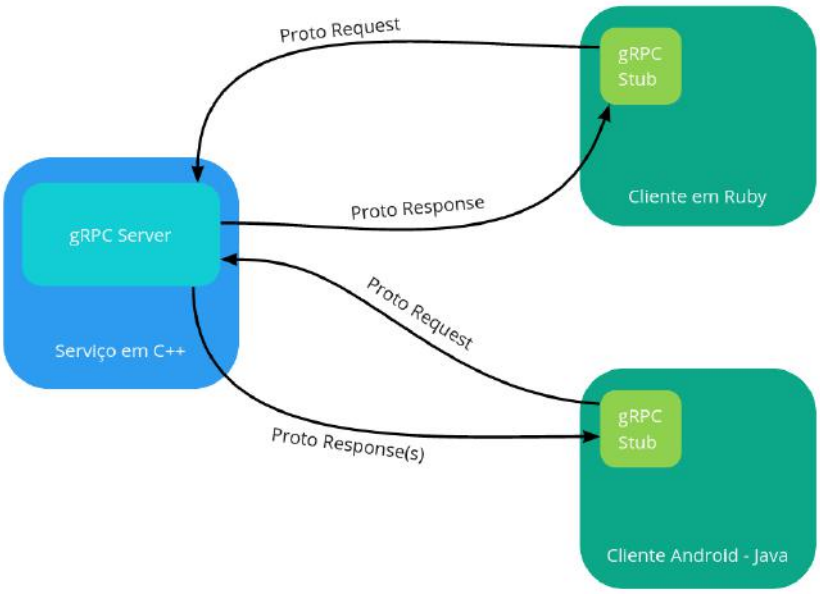
O gRPC (gRPC, 2015) é baseado na ideia da definição de serviços, cada um deles contendo um ou mais procedimentos de RPC, com seus parâmetros de entrada e de saída definidos na IDL. Toda essa especificação de tipos de entrada e de saída, serviços e procedimentos de RPC são descritas na sintaxe do Protobuffers em arquivos com a extensão **.proto**. Após a definição desses arquivos, o desenvolvedor utiliza a CLI (*Command Line Interface*) do compilador do Protobuffers (protoc), que irá gerar os arquivos de base para clientes e servidores nas linguagens que o desenvolvedor escolher, dentre uma vasta gama disponível, tais como C++, C#, Go, Java, Kotlin, Python, Ruby, etc. Desse modo é possível que clientes e servidores em diferentes linguagens interajam sem dificuldades, conforme mostra a Figura 4.

2.2.1.1 Transporte

Dado o contexto do Google de utilizar o gRPC como porta de entrada para seus serviços de nuvem pela internet, o gRPC é implementado com o protocolo mais comumente utilizado para comunicação na *web*: o HTTP, especificamente na sua segunda versão.

O protocolo HTTP/2 (BELSHE; PEON; THOMSON, 2015) introduz algumas melhorias em relação à sua versão anterior, tais como: o transporte de dados em formato binário ao invés de texto simples, o que proporciona melhores otimizações de codificação e compactação da informação trafegada; multiplexação entre múltiplos *streams* de dados numa mesma conexão, de tal modo que uma única conexão é utilizada para enviar e receber múltiplas requisições do gRPC; e compressão de cabeçalhos para diminuir a quantidade de dados transportados na rede.

Figura 4 – Arquitetura do gRPC



O protocolo de camada de transporte utilizado por baixo do HTTP/2 é o TCP, o que permite a oferta de algumas garantias não definidas no HTTP/2. Tais como: a garantia de entrega e ordenação de pacotes, controle de congestionamento e fluxo, entre outras. Cabe ainda comentar que há planos futuros para o gRPC utilizar o HTTP/3 (BISHOP, 2021), que faz o uso do QUIC para a camada de transporte.

2.2.1.2 Serialização e geração de código base

Para transportar os dados, o gRPC faz uso do Protobuffers, também utilizado como linguagem de IDL. Seu compilador é responsável por gerar os arquivos de código que os desenvolvedores utilizam como base para implementar as lógicas de negócio presente nas funções que serão executadas de maneira remota. Tais códigos base são exemplificados na Figura 4 e são as caixas nomeadas como **gRPC Server** e **gRPC Stub**, que abstraem as lógicas do servidor e dos clientes, respectivamente. Essas abstrações se responsabilizam por permitir que os desenvolvedores se preocupem apenas com a implementação das funções, sem se preocupar com as outras etapas necessárias para a realização do RPC, como por exemplo, lidar com a rede e serialização os dados.

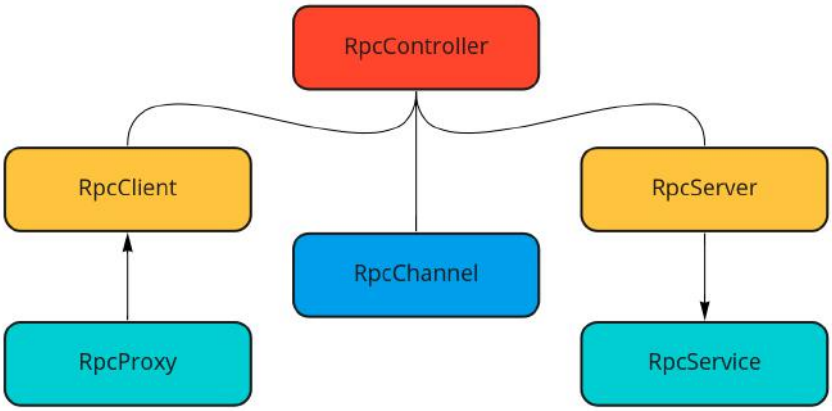
2.3 HPRPC

O HPRPC foi um protocolo RPC concebido com três propósitos em mente: arquitetura simples e leve, alta performance e suporte multiplataforma. O protocolo foi concebido para uso no contexto de HPC, e foi desenhado com arquitetura baseada no gRPC (BAGCI; KARA, 2016).

2.3.1 Arquitetura

O protocolo possui uma arquitetura com quatro componentes principais: um controlador, um componente para abstração de chamadas ao protocolo de transporte e outros dois componentes responsáveis por tratar o envio de dados por parte do cliente e recebimento de dados por parte do servidor.

Figura 5 – Arquitetura do HPRPC

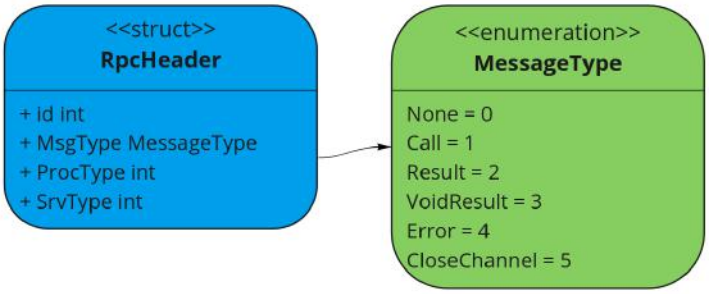


Na Figura 5 é possível observar o **RpcController** responsável por enviar mensagens dos clientes para os servidores usando os *channels*. Na mesma imagem também é possível observar o **RpcChannel** responsável pela camada de comunicação, que implementa os métodos de envio para camada de transporte por TCP. Além desses dois blocos, existem também os blocos **RpcClient** e **RpcServer**, que são camadas responsáveis pela abstração da chamada, recebimento e processamento do procedimento remoto. É interessante notar que no **RpcClient** existe o **RpcProxy**, o *proxy* consiste em classes geradas que fazem a abstração do transporte para que a chamada do RPC seja transparente ao programador. O último dos blocos é o **RpcService**, que consiste em uma classe customizada gerada que sobrescreve métodos do bloco **RpcServer**, para fornecer uma maneira de invocar o método correto do serviço.

2.3.2 Cabeçalhos

O cabeçalho do HPRPC possui quatro atributos. O primeiro atributo é o **id**, que armazena o identificador da mensagem, esse campo é importante para associar uma resposta a uma requisição. Outro campo disponível no cabeçalho é o **MsgType**, que indica se a mensagem em questão é uma chamada de RPC, uma resposta, uma resposta vazia, um erro, uma notificação de canal fechado ou uma mensagem sem tipo. Além desses, existem outros dois campos **ProcType** e **SrvType**, que indicam quais os serviços e procedimentos remotos a serem executados. O diagrama da classe **RpcHeader** pode ser visto na Figura 6.

Figura 6 – Cabeçalho do HPRPC



2.3.3 Limitações

O HPRPC foi desenvolvido com simplicidade como um dos pilares, por essa razão algumas limitações foram impostas ao protocolo e estão listadas abaixo:

- Chamadas síncronas - O HPRPC não suporta nativamente execução de procedimentos remotos de forma assíncrona.
- Cliente e servidor únicos - Somente um nó cliente e um nó servidor são suportados pelo protocolo.
- Plataformas homogêneas - O HPRPC foi implementado em diversas linguagens, entretanto, aplicações se comunicando a partir de linguagens distintas não podem ter *endianness* (ordenação de bytes) divergentes, além disso, outra limitação é que a codificação de *strings* deve ser idêntica, em essência os dados das linguagens não podem divergir de forma a gerar inconsistência entre o que é enviado e recebido.

2.3.4 Serialização de dados

Os autores do HPRPC implementaram um serializador próprio, chamado de KodoSis. Ele é a principal melhoria do HPRPC sobre o gRPC no contexto de HPC, possuindo um mecanismo de serialização simplificado que dispensa a necessidade de identificadores por campo.

A estratégia de suprimir identificadores nos campos serializados incorre na geração de uma sequência de dados serializados de tamanho menor, que implica em uma quantidade menor de dados transmitidos e por fim em um tempo menor de execução, quando comparado ao gRPC.

É importante ressaltar que o ganho de desempenho associado às simplificações do HPRPC restringem a implementação de diversos mecanismos, que o gRPC possui, para facilitar desenvolvimento e manutenção das aplicações RPC. Entre essas, é possível citar o suporte a argumentos opcionais e o versionamento de protocolo.

O KodoSis usa uma IDL própria a fim de normalizar o formato de dados e automatizar a geração de código para as linguagens suportadas. Atualmente o HPRPC e o KodoSis suportam C# com .NET e C++.

Deve-se notar que o KodoSis foi otimizado para serializar listas de tipos básicos de modo a gerar o menor dado serializado possível, dessa forma, os testes propostos pelo artigo estudado foram construídos com foco em listas de **int32** e **double**, para os quais o tempo de execução e o desempenho são melhores que o gRPC.

2.4 THRIFT

O Thrift (SLEE; AGARWAL; KWIATKOWSKI, 2007) foi desenvolvido no Facebook como um ambiente escalável para desenvolvimento de serviços entre diversas linguagens de programação. Seu objetivo principal é permitir que serviços heterogêneos possam se comunicar de maneira eficiente e confiável.

Isso é solucionado através de uma linguagem neutra, uma IDL, na qual desenvolvedores podem definir interfaces de serviços e seus respectivos tipos de dados, de modo que todo código necessário para estabelecer chamadas remotas de procedimentos entre clientes e servidores possa ser gerado automaticamente a partir dessas definições.

2.4.1 Tipos

A especificação define sete tipos básicos de dados: booleano; byte com bit de sinal; inteiros de 16, 32 e 64 bits acompanhados de sinal (**int16**, **int32**, **int64**); número de ponto flutuante com resolução de 64 bits (**float64**); textos agnósticos de codificação, que podem representar sequências binárias ou de caracteres (**binary** ou **string**).

Também são definidos quatro tipos complexos de dados: estruturas de dados, como um tipo comum representativo de um objeto, também chamado de *struct*; *containers*, especificamente listas, conjuntos e mapas de dados; exceções, como estruturas de dados especificas para tratamento de erro; e serviços, que servem como um definições para os procedimentos que podem ser executados remotamente entre clientes e servidores.

2.4.2 Transporte

O Thrift normalmente é utilizado sobre os protocolos TCP/IP, mas isso não é um requisito. Sendo possível definir qualquer tipo de transporte a ser usado. A especificação define um conjunto básico de métodos necessários para que o sistema possa ler e escrever dados por qualquer meio de comunicação a ser implementado.

2.4.3 Protocolo

Este *framework* define uma estrutura básica de mensagens que encapsulam os dados trocados entre cliente e servidor. Entretanto, ele não especifica um único formato para serialização de dados. Em contrapartida, sua especificação define um conjunto genérico de métodos a serem disponibilizados por qualquer tipo de codificador em um formato específico implementado para interagir com o Thrift. Os métodos definidos focam na escrita e leitura dos diferentes tipos de dados, assim como na definição da estrutura básica das mensagens.

2.4.4 Compilador e IDL

O compilador do Thrift é implementado em C++ e utiliza os programas *lex* e *yacc* para fazer a análise léxica e processar o código de sua IDL. A geração do código utilizado para estabelecer a comunicação RPC é feita em duas etapas. Inicialmente é executada a resolução de arquivos externos importados pela IDL, depois as definições de tipos são resolvidas e analisadas por verificadores de erros. Por fim, é gerado o código a partir da árvore sintática da IDL, que cria as interfaces a serem implementadas pelos procedimentos definidos, assim como a lógica necessária para interagir com meio de transporte escolhido, instanciar os servidores e clientes para comunicação e codificar os dados no formato implementado.

É possível analisar um exemplo de uma IDL do Thrift no Código 1.

Código 1 – Código exemplificando o uso da IDL do Thrift

```
namespace java calculator
namespace py calculator

enum Operation {
  ADD = 1,
  SUBTRACT = 2,
  MULTIPLY = 3,
  DIVIDE = 4
}

struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}

exception InvalidOperation {
  1: i32 what,
  2: string why
}

service Calculator {

  void ping(),

  i32 add(1:i32 num1, 2:i32 num2),

  i32 calculate(1:i32 logid, 2:Work w) throws (1:InvalidOperation ouch),

  /**
   * Este metodo tem o modificador oneway,
   * que implica que o cliente apenas faz a requisicao mas
   * nao espera a resposta.
   * Metodos com oneway devem retornar void.
   */
  oneway void zip()
}
```

Adaptado da documentação do Thrift (https://github.com/apache/thrift/blob/master/tutorial/tutorial.thrift)

O estudo dos *frameworks* de RPC apresentados neste capítulo, juntamente com o re-

ferencial teórico, demonstram a diversidade e robustez dos *frameworks* de RPC utilizados nos dias atuais. Neste sentido, no capítulo a seguir são apresentadas as decisões arquiteturais tomadas para o aRPC, bem como as justificativas para as mesmas. São também apresentados paralelos com soluções alternativas e cada um dos componentes do aRPC são minunciosamente detalhados.

3 DESCRIÇÃO DA PROPOSTA DE PROJETO

Conforme descrito nos projetos correlatos, já foi desenvolvida uma gama de *frameworks* RPC voltados para a comunicação entre processos e linguagens diferentes, cada qual com características únicas e diferentes objetivos. Foram propostas desde iniciativas para resolver as necessidades específicas de um conjunto de *softwares* de um empresa, até conceitos mais genéricos que visam melhorar a comunicação de *softwares* que usam o modelo cliente—servidor. Entretanto, mesmo com essa variedade de propostas, identificamos a falta de uma solução simples direcionada para a comunicação remota na área de computação de alto desempenho (SOUMAGNE; CARNS; ROSS, 2020). Em especial, em aplicações que incentivam desassociação de funcionalidades em microsserviços, que dependem mais da comunicação para compor sistemas mais distribuídos.

A partir dessas premissas, concebeu-se um *framework* de RPC para atender às demandas de um ambiente de computação de alto desempenho, com foco no desempenho e na simplicidade de uso, e também com o intuito de reduzir a sobrecarga cognitiva de uso pelos desenvolvedores, uma vez que sistemas voltados para computação de alto desempenho já impõem alta complexidade (LYNN, 2018).

Na Seção 2.1 foi elucidada a diferença entre RPC síncrono e assíncrono, bem como as vantagens desse último. Para proporcionar assincronia no aRPC foram utilizadas as funcionalidades de *goroutines* da linguagem Go. Onde a própria linguagem mantém um escalonador das *threads* leves, chamadas de *goroutines*, garantindo que a execução dos procedimentos bloqueantes não travem a aplicação.

3.1 COMPONENTES

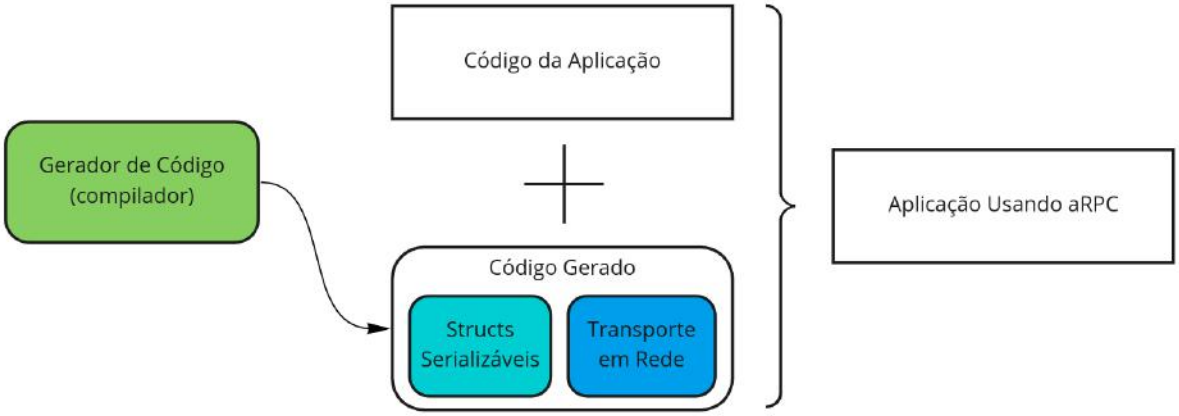
O aRPC é composto por três grupos principais de componentes: o protocolo de transporte, o serializador e o gerador de código (compilador), o diagrama de blocos pode ser visto na Figura 7.

O protocolo de transporte é responsável pela comunicação entre os nós clientes e os servidores, efetuando o envio dos dados referentes aos argumentos dos clientes para os servidores e das respostas dos servidores de volta aos clientes.

O serializador é responsável por traduzir estruturas de dados complexas em sequências de bytes e transformar sequências de bytes de volta em estruturas de dados complexas (NEWMARCH, 2017). No contexto do RPC, o serializador também viabiliza a comunicação entre processos e linguagens distintas.

A geração de código é uma etapa opcional, realizada por um compilador. Alguns protocolos de RPC exigem a definição de uma especificação das estruturas de dados e dos procedimentos disponibilizados pelos serviços oferecidos (FORD; HIBLER; LEPREAU,

Figura 7 – Componentes do aRPC



1995). No caso do aRPC, o compilador é responsável por efetuar uma etapa de leitura do arquivo de especificação (também chamado de IDL - *Interface Definition Language*) e em seguida gerar, na linguagem escolhida, o código que disponibiliza ao cliente as abstrações para efetuar chamadas de procedimentos, e para o servidor as interfaces necessárias para a implementação desses procedimentos. Desse modo, a serialização e transmissão de dados se torna transparente ao desenvolvedor.

3.2 PROTOCOLO DE TRANSPORTE

O aRPC tem como objetivo aprimorar a comunicação entre aplicações através da rede. Assim, se fez necessário a escolha de um protocolo de camada de transporte para melhor atender às características e necessidades de aplicações com foco em alto desempenho. Foram avaliados os prós e contras dos protocolos TCP, UDP e QUIC para utilização pelo *framework* proposto.

3.2.1 TCP

O Transmition Control Protocol (TCP) é um dos protocolos originais da implementação da rede de computadores. Normalmente referido como TCP/IP (CERF; KAHN, 1974). Desenvolvido em 1974 por Vint Cerf e Bob Kahn, o protocolo assegura a transmissão confiável de dados entre nós de uma rede de computadores. Para isso, implementa diversos mecanismos, como retransmissão de pacotes e controle de fluxo e congestão, que possibilitam um entrega de dados garantida, ordenada e com verificação de erros.

Enquanto essas características são de interesse para o funcionamento correto do aRPC, a complexidade de alguns desses métodos, do modo que foram implementados no TCP, podem trazer desvantagens para alguns casos de uso almejados.

Um problema em especial está no funcionamento do mecanismo de garantia de entrega ordenada, implementada pelo TCP, para casos onde há conjuntos de dados independentes sendo transmitidos. Isso ocorre porque o TCP não tem como averiguar o conteúdo dos dados, o que obriga que todo pacote, mesmo que não relacionado com os demais, caso chegue ao destino fora de sua ordem, tenha que esperar por todos os pacotes anteriores (SCHARF; KIESEL, 2006). Essa situação leva ao aumento da latência na comunicação, especialmente nos casos onde problemas na camada de enlace possam ocasionar elevadas ocorrências de retransmissão de pacotes. Além disso, com o rápido crescimento da largura de banda da rede de comunicação, algumas deficiências no desempenho do TCP se tornaram aparentes. Mesmo com ajuste minuciosos dos parâmetros de conexões, o protocolo subutiliza a largura de banda da rede disponível.

Estes problemas de latência e eficiência do TCP podem ser contornados com o uso de multiplexação de conexão a nível de aplicação acompanhado de ajustes no algoritmo de controle de congestão (BLANTON; PAXSON; ALLMAN, 2009). Entretanto, algumas características intrínsecas de funcionamento do protocolo dificultam um desempenho ótimo em diversos casos de interesse para um *frameworks* RPC. O alto custo do mecanismo de *handshake* em três etapas, especialmente entre nós já conhecidos, impossibilita o uso de multiplexação para conexões com vida útil curta, pois a latência introduzida resulta em desempenho médio pior do que com o uso de uma única conexão. Além disso, estratégias de *slow start* do mecanismo de congestão impossibilitam o uso de novas conexões para a transmissão de uma pequena quantidade de dados, pois dificilmente ela sairia do período de *slow start*, o que acarretaria em uso sub ótimo da largura de banda da conexão. A falta de multiplexação no nível de protocolo, impossibilita a reutilização dos sistemas de controle de congestão para múltiplas conexões entre os mesmos pares. Isso significa que a criação de novas conexões para cada transmissão de dado única sempre executa etapas que poderiam ser reaproveitadas. Logo, soluções de multiplexação de TCP que visem um bom desempenho, dependem do uso de estratégias de *pool* de conexões, para alcançar um equilíbrio entre a latência decorrente do agrupamento de dados dissociados e a latência decorrente da manutenção de novas conexões TCP.

A criptografia dos dados em trânsito na rede é essencial atualmente, além de ser uma prática básica em diversos regulamentos de segurança em Tecnologia da Informação (TI). O TCP não apresenta nenhum recurso de segurança nativo. Isso pode ser contornado com o uso de protocolos sobre o TCP, como o TLS (RESCORLA; DIERKS, 2008). Entretanto, a adição de TLS sobre TCP ocasiona um aumento na latência, devido à adição dos mecanismos de funcionamento necessários para esses protocolos extras.

Por fim, o TCP não foi escolhido como protocolo de transporte para o aRPC. Mesmo oferecendo diversas funcionalidades interessantes, as desvantagens demonstradas pelo protocolo vão contra vários pontos essenciais para o bom desempenho do *framework*, em especial na questão de latência (LAKSHMAN; MADHOW, 1997). Trabalhar para contornar

essas questões parece um acréscimo de complexidade e de escopo que escapam do propósito almejado, além de ir contra a premissa de manter a simplicidade de implementação.

3.2.2 UDP

O protocolo User Datagram Protocol (UDP) é um dos principais membros do conjunto de protocolos para a rede de computadores. Desenvolvido em 1980 por David P. Reed e padronizado na RFC768 (POSTEL, 1980), foi concebido com objetivo de ser um protocolo simples com um mínimo de mecanismos necessários para seu funcionamento sobre a camada IP. Tem como características principais um funcionamento independente de conexão, sem estado e sem garantias de entrega ou lógica para retransmissão, manutenção de ordem ou proteção contra duplicação. Sendo assim, o protocolo utiliza como unidade básica de transporte o datagrama, que são estruturas básicas de encapsulamento normalmente compostas por um cabeçalho e um bloco de dados. No cabeçalho é incluído a porta de origem e destino, além do seu tamanho e um código de verificação. Comparando-se com o pacote do TCP, que incluiu um cabeçalho consideravelmente maior e mais complexo, com informação referente a ordem, confirmação, informação para controle do fluxo de dado e controle de congestão.

Um protocolo simples traz diversas vantagens para certos aplicações e casos de uso. Em aplicações extremamente sensíveis à latência, a falta de um etapa de *handshake* e de atrasos por retransmissão são grandes vantagens do UDP. Especialmente para dados que são resilientes à perda de certa quantidade de informação, nesse caso é possível citar transmissões de mídia em tempo real (VoIP, *streaming* de video), na qual os *codecs* de mídia já incluem mecanismos para correção de erro ou então a perda de dados não prejudica de maneira muito relevante o conteúdo transmitido.

No caso específico para uso como protocolo de transporte de um *framework* de chamada de procedimento remoto, a simplicidade do protocolo é muito interessante. A minimização da latência é importante já que, para ser um substituto para a execução de funções locais, requer um mínimo de atraso no estabelecimento de comunicação entre os pares. Entretanto, a falta de mecanismos básicos de garantia de entrega e ordenação de datagramas são problemáticos, particularmente a perda de dados relativos aos argumentos e respostas de procedimentos não é um comportamento que possa ser considerável válido no contexto do *framework*.

É importante ressaltar que o UDP pode apresentar algumas restrições de vazão dependendo do ambiente de rede. Isso é feito como medida de proteção e prevenção contra alguns tipos de abusos e ataques específicos ao UDP. Dentre eles, os ataques de amplificação (ROSSOW, 2014) se destacam como fonte de apreensão entre provedores de serviços, devido à facilidade com que podem ser explorados por agentes mal intencionados e à grande extensão do impacto que podem causar. Por isso, muitos provedores de

serviços implementam limites nas taxas de transferência do UDP para impedir que suas infraestruturas seja usadas para promover esses tipos de ataques.

Sendo assim, o UDP não foi escolhido como protocolo de transporte para o aRPC. Já que seu funcionamento, por mais que seja simples e eficiente, não oferece funcionalidades básicas esperadas para que o *framework* possa cumprir o funcionamento desejado. Além disso, a implementação dos mecanismos necessários para fornecer esses recursos sobre o protocolo, na camada de aplicação, foge da premissa inicial do aRPC de manter a simplicidade de implementação.

3.2.3 QUIC

QUIC é um protocolo da camada de transporte, inicialmente desenvolvido por Jim Roskind no Google, implementado para aplicações da empresa a partir de 2012, como o *Youtube* por exemplo (LANGLEY et al., 2017). Em 2015 foi submetido à IETF (*Internet Engineering Task Force*) para ser padronizado de maneira aberta e independente.

Este protocolo se apresenta como um protocolo da camada de transporte, porém possui algumas características encontradas em protocolos da camada de aplicação, como a multiplexação de *streams*, que pode ser encontrada no HTTP/2 (SAXCé; OPRESCU; CHEN, 2015), além de utilizar o protocolo UDP, que também é um protocolo de camada de transporte.

Além da multiplexação de *streams*, o QUIC ainda apresenta outras características interessantes para um RPC, tais como a conexão sendo realizada em 1-RTT, ao invés de 3-RTT presente no TCP; transmissão de dados em 0-RTT para conexões subsequentes para um servidor já conhecido; garantia de entrega de pacotes, possuindo ainda garantia de ordenação de dados apenas dentro uma mesma *stream*. Dessa maneira evita o problema de *Head-of-Line Blocking*, que ocorre quando um pacote de uma *stream* é perdido e todas as *streams* ficam esperando esse pacote ser reenviado, mesmo que as *streams* sejam independentes. Tais diferenças entre o QUIC e o TCP podem são descritas em (COOK et al., 2017). Cabe ainda ressaltar que o protocolo impõe que toda a comunicação de dados seja sempre efetuada utilizando TLS, desse modo, os dados trafegados são criptografados e é possível verificar a autenticidade de servidores e clientes remotos. Pela imposição do TLS, o protocolo já se organiza para que a troca de chaves criptográficas ocorra no *handshake* inicial, sem a necessidade de efetuar mais de um RTT para preparar a comunicação criptografada, como é o caso do TCP.

Devido ao uso do UDP como base, os problemas de restrições presentes em alguns provedores de serviço podem afetar a performance do QUIC, conforme visto na subseção anterior sobre o UDP. Isso é um fato relevante, entretanto, o protocolo inclui um mecanismo para impedir falsificação de IPs de origem (ROSSOW, 2014), no qual clientes são atribuídos um *token* autenticado pelo servidor que estão se comunicando para assegurar a propriedade do seu IP de origem. Isso impede que ataques de amplificação possam ocor-

rer, já que eles dependem da falsificação do IP de origem do datagrama. Sendo assim, por mais que atualmente alguns provedores possam ter regras que impactam o QUIC, espera-se que num futuro próximo essas restrições sejam removidas, dado a popularização do uso do protocolo.

O protocolo QUIC foi o escolhido como mecanismo de transporte do aRPC por conta das características apresentadas, essa escolha tem como objetivo diminuir o *overhead* de transporte perante outros *frameworks* que utilizam o TCP como base. Outras características que merecem destaque são a multiplexação de *streams* que é usada pelo aRPC para separar chamadas de procedimentos distintas, possibilitando um código mais simples e abrindo mão da necessidade de armazenar IDs das requisições e respostas nos cabeçalhos. Além disso, do ponto de vista do balanceamento de carga onde um cliente pode se conectar a vários servidores distintos ao longo do tempo, as características de realizar a conexão em 1-RTT e envio de dados em 0-RTT para conexões subsequentes também se tornam características muito desejáveis para um *framework* de RPC (FATEMIAN, 2020).

3.3 SERIALIZADOR

Quando se trafega dados pela rede, é necessário que as duas pontas, cliente e servidor, estejam se comunicando no mesmo formato. Para o protocolo HTTP até a versão 1.1 por exemplo, esse formato é em texto simples, que é legível por humanos. A partir da versão 2 do HTTP e em outros protocolos, a comunicação é feita utilizando dados em formato binário. Um dos benefícios da utilização de um formato pré-definido de comunicação entre as partes é a possibilidade que cliente e servidor sejam implementados em linguagens e plataformas distintas (SLEE; AGARWAL; KWIATKOWSKI, 2007).

Tanto cliente quanto servidor lidam diretamente com dados estruturados dentro de suas respectivas aplicações, para traduzir esses dados num formato binário a ser trafegado pela rede, se faz necessária uma etapa de serialização, ou seja, tradução das estruturas de dados para formato binário usado no transporte.

Para a implementação deste trabalho, vários serializadores foram considerados, onde foram avaliados seus prós e contras, tais como o Protobuffers, Cap'n Proto e Colfer, que são descritos a seguir.

3.3.1 Protobuffers

Desenvolvido pelo Google como um formato para comunicação entre aplicações internas da empresa, tem o propósito de ser agnóstico à linguagem e à plataforma.

Ao utilizar Protobuffers, o desenvolvedor especifica as estruturas de dados que ele deseja serializar em um IDL com a extensão **.proto**, esse arquivo define as estruturas com capacidade de serialização. O programador então passa esses arquivos para o compilador do Protobuffers, que irá gerar o código que define estruturas com capacidade de serializa-

ção na linguagem que o desenvolvedor escolher. Protobuffers suporta uma vasta gama de linguagens, como C, C++, Java, Python, Go, Ruby, Objective-C, C#, JavaScript, Rust, entre outras.

Este serializador também oferece uma vasta gama de funcionalidades, tais como diversos tipos primitivos, mapas, enumerações, listas, tipos aninhados, campos opcionais, campos com valor padrão, entre outras (GOOGLE, 2008). Um ponto negativo que se pode citar é a imposição do Protobuffers que cada campo nas *structs* tenham um número identificador, tal número é incorporado na serialização, aumentando a quantidade de bytes na mensagem, que por consequência tende a aumentar o tempo de transmissão dos dados (BAGCI; KARA, 2016), entretanto, este fator é mitigado pelas estratégias de otimização na serialização.

Um ponto relevante a se destacar, é o alto acoplamento que este serializador tem com o *framework* gRPC, também do Google, diversas funcionalidades do Protobuffers existem pra atender ao funcionamento do gRPC, assim como diversas *funcionalidades* do gRPC se adéquam ao modo como o Protobuffers funciona. Tais motivos fizeram com que ele não fosse o serializador escolhido para o aRPC, dado que diversas funcionalidades que existem para atender às demandas do gRPC não se fazem necessárias para o propósito deste trabalho, introduzindo apenas um custo computacional desnecessário.

Código 2 – Código exemplificando o uso da IDL do Protobuffers

```
syntax = "proto3";

option go_package = "github.com/almeida-raphael/arpc_examples";

message NumberList {
  repeated int32 entries = 1;
}

message Result {
  int32 value = 1;
}

service DoubleType {
  rpc Average(NumberList) returns (Result) {}
}
```

No Código 2 pode ser conferido um exemplo de arquivo **.proto**. Nele são definidos os tipos de mensagem **NumberList** e **Result**, utilizados como argumentos de entrada e de saída, respectivamente, para o procedimento **Average** dentro do serviço **DoubleType**. Ressaltando-se também que pode haver partes adicionais específicas parada cada lingua-

gem que os desenvolvedores estejam usando, um exemplo pode ser visto na linha com a variável **go_package** que define o pacote a ser utilizado pelo arquivo de código base gerado para a linguagem Go. Tais detalhes exclusivos de cada linguagem podem ir gerando uma sobrecarga maior no arquivo de IDL ao longo do tempo, que potencialmente pode conter diversas linhas com elementos específicos de diversas linguagens, introduzindo uma complexidade desnecessária.

O Protobuffers ainda permite que os códigos gerados para serialização dos dados tenham otimizações específicas escolhidas pelo desenvolvedor na hora de executar o compilador para gerar os arquivos, tais como otimizações para comprimir os dados e trafegar uma menor quantidade de dados pela rede ou então para serializar os dados de maneira mais simplificada, de modo com que o custo computacional de processamento na serialização e desserialização seja menor. Entretanto, neste trabalho essas otimizações não foram avaliadas, o compilador do Protobuffers foi utilizado no modo padrão, onde o código gerado tende a ser eficiente de uma maneira mais genérica, sem nenhuma otimização voltada para casos específicos.

3.3.2 Cap'n Proto

Desenvolvido por Kenton Varda, um dos principais autores do Protobuffers após sua saída do Google, consolidado a partir de anos de experiência do criador ao trabalhar com sua criação anterior. O Cap'n Proto procura acertar em pontos cujo o autor julgou como problemáticos no Protobuffers. Um exemplo pode ser conferido no Código 3

Código 3 – Código exemplificando o uso da IDL do Cap'n Proto

```
using Go = import "/go.capnp";
@0x85d3acc3f0f0e0f8;
$Go.package("books");
$Go.import("pkg/books");

struct Book {
    # Titulo do livro.
    title @0 :Text;

    # Numero de paginas no livro.
    pageCount @1 :Int32;
}
```

Adaptado da documentação da biblioteca em Go do Cap'n Proto (https://github.com/capnproto/go-capnproto2/wiki/Getting-Started)

Um dos conceitos chave do Cap'n Proto é o de zero cópias de memória nas etapas de serialização e desserialização dos dados. Os mesmos bytes operados na aplicação são

os bytes que serão trafegados pela rede. Evitando assim o processamento dos dados na preparação da mensagem para envio.

O Cap'n Proto ainda apresenta diversas funcionalidades interessantes, tais como: a leitura incremental dos dados, que permite que uma mensagem possa começar a ser lida e desserializada antes ter sido recebida por completo; acesso aleatório, onde é possível ler um dos dados da mensagem sem a necessidade de desserializar os demais dados; o código gerado é pequeno, enquanto o Protobuffers gera código específico de serialização e desserialização para cada tipo de mensagem diferente, o Cap'n Proto utiliza chamadas otimizadas de sua própria biblioteca para lidar com essa parte, sendo que a própria biblioteca que é executada junto do programa também é pequena e otimizada.

Apesar das funcionalidades interessantes, assim como o Protobuffers, o Cap'n Proto também é desenvolvido ao redor de um *framework* de RPC próprio, com isso diversas funcionalidades existem apenas com o propósito de atender como o autor vislumbrou que o RPC deve funcionar, não atendendo tão bem às demandas deste trabalho. Além disso, para proporcionar um conjunto rico de funcionalidades e ao mesmo tempo proporcionar uma boa performance, o Cap'n Proto tomou certas escolhas de implementação que prejudicam a ergonomia na escrita de código, forçando os desenvolvedores a adequarem o resto da aplicação aos padrões do Cap'n Proto ou então ficar com o código heterogêneo no que se refere a manipulação de dados, conforme pode ser visto no Código 4.

Código 4 – Código exemplificando o meio de popular os objetos do Cap'n Proto

```
//  Monta uma nova mensagem vazia, alocando uma struct do Cap'n Proto.
msg, seg, err := capnp.NewMessage(capnp.SingleSegment(nil))
if err != nil {
    panic(err)
}

// Cria uma nova struct de Book.
book, err := books.NewRootBook(seg)
if err != nil {
    panic(err)
}
book.SetTitle("Ready Player One")
book.SetPageCount(389)

// Escreve a mensagem para a saida padrao.
err = capnp.NewEncoder(os.Stdout).Encode(msg)
if err != nil {
    panic(err)
}
```

Adaptado da documentação da biblioteca em Go do Cap'n Proto (https://github.com/capnproto/go-capnproto2/wiki/Getting-Started)

3.3.3 Colfer

Desenvolvido por Pascal de Kloe, Colfer é um formato de serialização binária, otimizado em tamanho e velocidade.

O uso do Colfer se dá através de um compilador que processa aquivos **.colf**, escritos em uma IDL que discrimina as estruturas de dados a serem utilizadas, e gera código fonte em uma das linguagens de programação suportadas. O código gerado é responsável por serializar as estruturas de dados previamente descritas em binário e vice-versa.

A IDL utilizada para descrição dos dados é derivada da sintaxe da linguagem Go, especificamente da sua definição de *structs*, com a extensão de alguns tipos específicos definidos pelo Colfer, conforme pode ser visto no Código 5.

Código 5 – Código exemplificando o uso da IDL do Colfer

```
// Course e o campo de golf onde o jogo acontece
type course struct {
    ID    uint64
    name  text
    holes []hole
    image binary
    tags  []text
}

type hole struct {
    // Lat e a latitude do buraco.
    lat float64
    // Lon e a longitude do buraco.
    lon float64
    // Par e o indice de dificulade.
    par uint8
    // Water indica a presenca de agua.
    water bool
    // Sand indica a presenca de areia.
    sand bool
}
```

Adaptado da documentação do Colfer (https://github.com/pascaldekloe/colfer)

A vantagem do uso da sintaxe do Go é a fácil reutilização do ferramental já disponível para análise e manipulação de código textual, como o módulo AST, nativo do Go, que permite facilmente construir e interagir com a árvore sintática abstrata a partir do código fonte fornecido.

O Colfer define quatorze tipos de dados distintos. Entre eles, um tipo básico repre-

sentando um booleano; quatro tipos básicos de inteiros não incluindo sinal, com resolução de 8, 16, 32 e 64 bits; dois tipos de inteiros incluindo sinal, com resolução de 32 e 64 bits; dois tipos representativos de números com ponto flutuante, com resolução de 32 e 64 bits; um tipo que representa o registro de data/hora; um tipo que representa uma sequência variável de caracteres textuais; um tipo que representa uma sequência variável de dados binários; um tipo que representa um lista variável de qualquer um dos tipos disponíveis.

O formato de dados constitui-se por uma estrutura contendo zero ou mais definições de campos de valores, seguido por um byte de término 0x7f. Somente são serializados campos de valores que representem dados não nulos, ou valores diferentes de zero. Cada campo é representado em memória no formato *big-endian*, composto por um cabeçalho de 8 bits seguido por um campo de dado codificado em um formato de tamanho fixo ou de tamanho variável em base 128 (UBEB128) (WANG et al., 2017). Os 7 bits menos significativos do cabeçalho identificam o tipo do dado, de acordo com um índice internamente atribuído e o bit mais significativo é uma *flag* reservada para usos específicos.

O formato do campo de dados de cada tipo é definido abaixo:

- Booleanos somente são codificados quando verdadeiros, e não tem campo de dados, sendo definidos exclusivamente pelo cabeçalho.
- Inteiros de 8 bits sem sinal são codificados em formato fixo como 1 byte.
- Inteiros de 16 bits sem sinal, menores que 0xFF, são codificados em formato fixo como 1 byte e com o valor do bit de *flag* definido. Valores maiores são codificados em dois bytes.
- Inteiros de 32 bits sem sinal, menores que 0x200000 são codificados como inteiros de comprimento variável. Valores maiores são codificados em formato fixo como 4 bytes e com o valor do bit de *flag* definido.
- Inteiros de 64 bits sem sinal, menores que 0x2000000000000 são codificados como inteiros de comprimento variável. Valores maiores são codificados em formato fixo como 8 bytes e com o valor do bit de *flag* definido.
- Inteiros de 32 e 64 bits com sinal são codificados como inteiros de comprimento variável, onde o valor do bit da *flag* representa o sinal. No caso de inteiros de 64 bits maiores que 0x8000000000000000, o ultimo byte da codificação UBEB128 não é codificado pois ele sempre será 0x1.
- Números de ponto flutuante são codificados conforme a IEEE754 (IEEE..., 2019).
- Registros de data/hora podem ser codificados de dois modos:
  - Inteiro de 32 bits sem sinal representando o número de segundos decorridos a partir de 00:00:00 UTC, quinta-feira, 1 de janeiro de 1970

  - Inteiro de 64 bits sem sinal representando o número de segundos decorridos a partir de 00:00:00 UTC, quinta-feira, 1 de janeiro de 1970, e com o valor do bit de *flag* definido

  Ambos os formatos são seguidos de um inteiro de 32 bits (onde somente os primeiros 30 bits são utilizados) sem sinal representando a fração de nanosegundos.

- Listas são codificadas por um inteiro de tamanho variável representando o número de entradas(**N**), seguido por **N** campos de valores representando os dados da lista.
- Sequências de bytes são definidas por por um inteiro de tamanho variável representando o número de entradas(**N**), seguido por **N** bytes descrevendo o dado original.
- Textos seguem a mesma codificação de sequências de bytes, com os caracteres codificados em UTF-8 (UNICODE..., 2010).

O código gerado pelo Colfer, a partir do arquivo de definições, é livre de dependências e somente utiliza as funcionalidades nativas de cada linguagem suportada para codificar e decodificar as estruturas de dados. Orientado para simplicidade e com um mínimo de *overhead* para realizar as tarefas necessárias para serialização, o código gerado não considera como o dado será transportado, sendo de responsabilidade do desenvolvedor interagir com os métodos disponibilizados.

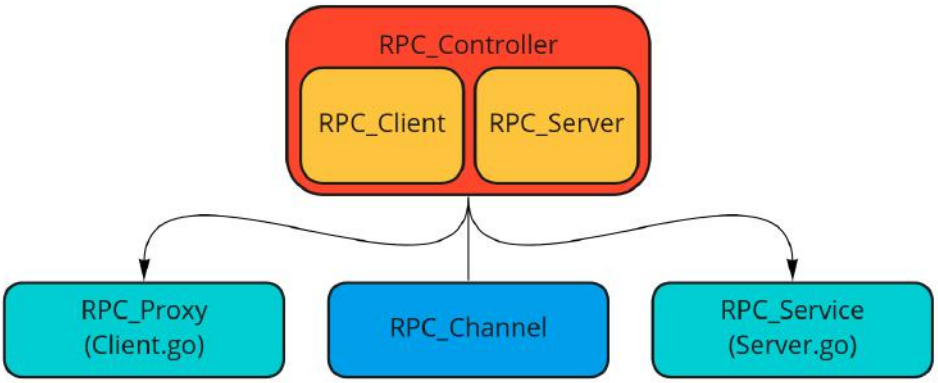
O Colfer se destaca entre os serializadores analisados por diversas vantagens: simplicidade e extensibilidade que permitem fácil integração de suas funcionalidades; não necessita de dependências externas, com todas as funcionalidades sendo oferecidas pelo compilador; geração de código nativo na linguagem alvo, com foco no alto desempenho, sem sacrifício de segurança. Esse conjunto de características fez com que o Colfer fosse o serializador escolhido para uso no aRPC.

3.4 ARQUITETURA DO ARPC

A arquitetura do aRPC é inspirada pela arquitetura do gRPC e do HPRPC (BAGCI; KARA, 2016), como pode-se observar na figura 8.

Primeiramente, o **RPC_Client** e **RPC_Server** que definem as funções de envio e recebimento foram incorporadas no **RPC_Controler**. O **RPC_Proxy** se refere ao código gerado para clientes e **RPC_Service** para servidores, dessa forma são análogos aos arquivos **Client.go** e **Server.go** gerados pelo compilador do aRPC para cada serviço. Além disso, o **RPC_Channel** foi implementado usando interfaces baseadas na implementação do QUIC no Go, facilitando dessa forma a construção do controlador a partir das facilidades do QUIC, como a multiplexação de *streams*. As alterações na arquitetura explicitadas acima podem ser observadas ao comparar as Figuras 5 e 8. A implementação do **RPC_Channel** no aRPC parte de interfaces que possibilitam o uso de

Figura 8 – Arquitetura do aRPC



outro protocolo como base, como HTTP/2 ou TCP. Essas implementações foram deixadas como recomendação de trabalhos futuros, para uma melhor comparação dos protocolos de transporte isoladamente.

O seguintes critérios foram considerados fundamentais na definição do *framework*: rapidez na serialização, na desserialização e no transporte; capacidade de produzir binários pequenos para envio; não comprometimento da ergonomia de desenvolvimento normal da linguagem e; finalmente, facilidade de utilização pelos desenvolvedores. Atualmente a única linguagem suportada é Go, entretanto, como o *framework* é simples, sua portabilidade e funcionamento em conjunto com outras linguagens é de fácil implementação.

Quanto ao envio de mensagens, optou-se por usar 1 byte inicial que define o tamanho do cabeçalho, em seguida dentro do mesmo existe um campo que define o tamanho dos dados enviados, com essas duas informações ambas as pontas (cliente e servidor) são capazes de montar a requisição e de serializar e desserializar cabeçalho e dados. Essa arquitetura de mensagem foi definida pensando facilitar a desserialização e também em usar a característica do Colfer de serializar somente os dados necessários, reduzindo a sobrecarga de envio do cabeçalho e do corpo de cada mensagem.

É interessante notar que, do modo como o controlador foi implementado, é possível rodar o cliente e o servidor de serviços aRPC simultaneamente. Com uma única instância de **aRPC_Controller** é possível consumir múltiplos serviços e ao mesmo tempo oferecer serviços a múltiplos clientes.

Anteriormente, foi dito que o aRPC possui um gerador de código, este compilador é responsável por gerar os códigos base que são utilizados para integrar os serializadores com o controlador e fazer serialização e transportes serem transparentes do ponto de vista do desenvolvedor. Um exemplo de código gerado para o cliente pode ser conferido no Código 6, enquanto o lado do servidor pode ser visto no Código 7.

Código 6 – Código gerado pelo compilador do aRPC para o cliente de um serviço simples

```
package greetings

// Codigo gerado pelo aRPC; NAO EDITAR!

import (
    "context"
    "github.com/almeida-raphael/arpc/controller"
)

type Greetings struct {
    controller *controller.RPC
}

func NewGreetings(controller *controller.RPC) Greetings {
    return Greetings{
        controller: controller,
    }
}

func (greetings *Greetings)SayHi(SayHiRequest *SayHiRequest, ctx ...
   context.Context)(*SayHiResponse, error){
    if ctx == nil || len(ctx) == 0 {
        ctx = []context.Context{context.Background()}
    }

    response := SayHiResponse{}

    err := greetings.controller.SendRPCCall(
        ctx[0],
        4148486943,
        0,
        SayHiRequest,
        &response,
    )

    if err != nil {
        return nil, err
    }

    return &response, nil
}
```

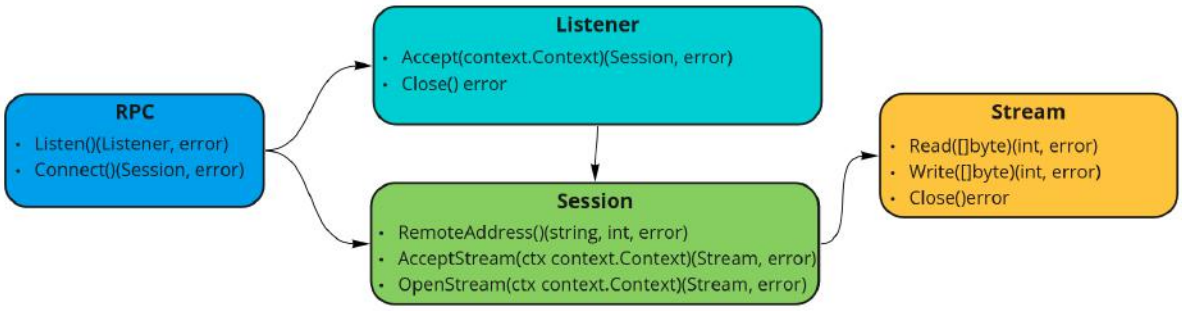
Código 7 – Código gerado pelo compilador do aRPC para o servidor de um serviço simples

```
package greetings

// Codigo gerado pelo aRPC; NAO EDITAR!

import "github.com/almeida-raphael/arpc/controller"

type GreetingsServer interface {
    SayHi(SayHiRequest *SayHiRequest)(*SayHiResponse, error)
}

func bindSayHi(server GreetingsServer)(
    func(msg []byte)([]byte, error),
) {
    return func(msg []byte)([]byte, error){
        SayHiRequest := SayHiRequest{}
        err := SayHiRequest.UnmarshalBinary(msg)
        if err != nil {
            return nil, err
        }

        response, err := server.SayHi(&SayHiRequest)
        if err != nil {
            return nil, err
        }

        responseBytes, err := response.MarshalBinary()
        if err != nil {
            return nil, err
        }

        return responseBytes, nil
    }
}

func RegisterGreetingsServer(controller controller.RPC, srv
   GreetingsServer){
    controller.RegisterService(
        4148486943,
        map[uint16]func(message []byte)([]byte, error){
            0: bindSayHi(srv),
        },
    )
}
```

3.4.1 Channel

O módulo **aRPC_Channel** é responsável por abstrair para o controlador qual protocolo de transporte será usado na comunicação aRPC. Neste trabalho foi implementado o **QUIC_Channel**, fornecendo uso do aRPC através de QUIC. No entanto, a implementação de *channels* para outros protocolos de transporte é de fácil construção, necessitando apenas que sejam implementadas as interfaces a seguir:

Figura 9 – Relação das interfaces de um *channel* no aRPC



3.4.1.1 Interface channel.RPC

O **channel.RPC** é a interface de definição raiz do canal de comunicação e deve fornecer duas funções, **Listen** e **Connect**. As funções e suas respectivas definições estão listadas abaixo:

- **Listen()(Listener, error)** - Função responsável por iniciar o recebimento de conexões, retornando em caso de sucesso um **Listener**, que é uma outra interface que deve ser implementada e cuja definição será vista na subseção seguinte.
- **Connect()(Session, error)** - Função que executa para um cliente a conexão com um servidor que tenha chamado o método **Listen** previamente. Retornando assim uma **Session** que é uma interface que deve ser implementada e cuja definição será vista a seguir.

3.4.1.2 Interface channel.Listener

A interface **channel.Listener** provê ao **aRPC_Controller** as funções referentes ao servidor e um meio de receber novas conexões de clientes. Dessa forma, essa interface define os métodos abaixo:

- **Accept(context.Context)(Session, error)** - Função que espera novas conexões e em seguida retorna uma sessão para conexão com o cliente. Como dito anterior-

mente, **Session** é uma das interfaces que devem ser implementadas e cuja definição será vista mais à frente.

- **Close() error** - Essa função fecha o *listener* para recebimento de novas conexões, e consequentemente, o servidor associado a ele.

3.4.1.3 Interface channel.Session

A interface **channel.Session** é responsável por fornecer ao **aRPC_Controller** acesso às propriedades da conexão e também fazer a abertura ou recebimento de *streams* de dados. No aRPC cada chamada de procedimento remoto abre uma nova *stream* que é fechada ao fim da execução do procedimento. Uma observação importante é que fechar a *stream* não implica em fechar a conexão, já que a *stream* é um mecanismo lógico. A interface **channel.Session** define assim, os métodos abaixo:

- **RemoteAddress()(string, int, error)** - Essa função retorna para o nó em questão (cliente ou servidor) o endereço e porta usados para conexão com a máquina remota.
- **AcceptStream(ctx context.Context)(Stream, error)** - Função bloqueante, responsável por liberar o fluxo quando um nó remoto abre uma nova **Stream** e retornar essa **Stream** que foi aberta. Esse objeto retornado também é uma interface que deve ser implementada e cuja definição será vista adiante.
- **OpenStream(ctx context.Context)(Stream, error)** - Inicia uma nova **Stream** dentro da **Session** atual. A implementação da interface **Stream** é vista a seguir.

3.4.1.4 Interface channel.Stream

A interface **channel.Stream** é a última das interfaces que deve ser implementada para disponibilizar um novo **Channel**, nela estão definidas funções de envio e leitura de bytes e também um método para fechar a **Stream**. A interface **channel.Stream** especifica os seguintes métodos:

- **Read([]byte)(int, error)** - Lê a partir da **Stream** dados enviados pelo nó remoto.
- **Write([]byte)(int, error)** - Escreve os dados contidos em um **[]byte** na **Stream** para leitura pelo nó remoto.
- **Close()error** - Essa função fecha a **Stream** em questão e no caso do aRPC finaliza uma execução de um procedimento de RPC.

3.4.1.5 Implementação do QUIC_Channel

No estado atual, o *framework* disponibiliza um **Channel** que usa QUIC como protocolo de transporte nativo, e essa implementação pode ser vista tanto no github do aRPC (https://github.com/almeida-raphael/arpc) quanto nos Códigos 8, 9, 10 e 11. A implementação atual disponibiliza *streams* multiplexadas, *handshake* de baixo custo computacional, entre outros benefícios associados ao uso do QUIC como protocolo de transporte. Como as interfaces foram desenhadas baseadas em conceitos desse protocolo de transporte, isso facilitou muito sua implementação como **Channel** inicial disponibilizado no *framework*. Dessa forma, pode-se observar que o código é bastante semântico e curto. O QUIC foi escolhido como **Channel** nativo por apresentar características que indicavam ser uma ótima escolha inicial para o *framework*, que por definição fornece criptografia e verificação de autenticidade de origem com TLS, além de ser um protocolo de simples utilização.

Código 8 – Implementação da interface channel.RPC para QUIC

```
// QUICRPC implementacao do canal de RPC para QUIC
type QUICRPC struct {
    address   string
    port      int
    tlsConfig *tls.Config
    config    *quic.Config
}

// Listen escuta as conexoes de entrada no QUICRPC
func (q *QUICRPC) Listen() (Listener, error) {
    listener, err := quic.ListenAddr(
        fmt.Sprintf("%s:%d", q.address, q.port),
        q.tlsConfig,
        q.config,
    )
    if err != nil {
        return nil, err
    }
    return &QUICListener{listener: listener}, nil
}

// Connect conecta num servidor com QUICRPC
func (q *QUICRPC) Connect() (Session, error) {
    session, err := quic.DialAddr(
        fmt.Sprintf("%s:%d", q.address, q.port),
        q.tlsConfig,
        q.config,
    )
    if err != nil {
        return nil, err
    }
    return &QUICSession{session: session}, nil
}
```

Código 9 – Implementação da interface channel.Listener para QUIC

```
// QUICListener e a implementacao do Listener para QUIC
type QUICListener struct {
    listener quic.Listener
}

// Close fecha o QUICListener
func (l *QUICListener) Close() error {
    return l.listener.Close()
}

// Accept recebe uma conexao para um dado QUICListener
func (l *QUICListener) Accept(ctx context.Context) (Session, error) {
    session, err := l.listener.Accept(ctx)
    if err != nil {
        return nil, err
    }

    return &QUICSession{session: session}, nil
}
```

Código 10 – Implementação da interface channel.Session para QUIC

```
// QUICSession e a implementacao de Session para o QUIC
type QUICSession struct {
    session quic.Session
}

// RemoteAddress retorna o endereco e a porta de uma QUICSession
func (qs *QUICSession) RemoteAddress() (string, int, error) {
    addrParts := strings.Split(qs.session.RemoteAddr().String(), ":")
    if len(addrParts) != 2 {
        return "", 0, fmt.Errorf("invalid remote address")
    }

    portInt64, err := strconv.ParseInt(addrParts[1], 10, 32)
    if err != nil {
        return "", 0, fmt.Errorf("invalid port number")
    }

    return addrParts[0], int(portInt64), nil
}

// AcceptStream recebe uma stream de dados de um client
func (qs *QUICSession) AcceptStream(ctx context.Context) (Stream, error)
    {
    stream, err := qs.session.AcceptStream(ctx)
    if err != nil {
        return nil, err
    }

    return &QUICStream{stream: stream}, nil
}

// OpenStream cria uma nova stream de dados com o servidor
func (qs *QUICSession) OpenStream(ctx context.Context) (Stream, error) {
    stream, err := qs.session.OpenStreamSync(ctx)
    if err != nil {
        return nil, err
    }

    return &QUICStream{stream: stream}, nil
}
```

Código 11 – Implementação da interface channel.Stream para QUIC

```
// QUICStream is a channel stream implementation for QUIC
type QUICStream struct {
    stream quic.Stream
}

// Close fecha a QUICStream
func (s *QUICStream) Close() error {
    return s.stream.Close()
}

// Write escreve os dados em uma QUICStream
func (s *QUICStream) Write(buf []byte) (int, error) {
    return s.stream.Write(buf)
}

// Read le os dados de uma QUICStream
func (s *QUICStream) Read(buf []byte) (int, error) {
    return s.stream.Read(buf)
}
```

3.4.2 Controler

O **aRPC_Controller** é o bloco central que disponibiliza métodos de envio e recebimento de requisições de RPC. Ele é responsável por controlar tamanhos de mensagens, gerar e ler cabeçalhos, serializar e desserializar dados, registrar serviços e procedimentos oferecidos e dar ao *framework* formas de enviar e receber respostas RPC, sem se preocupar com a etapa de transporte.

No aRPC, a camada do **Controler** não é diretamente usada pelo desenvolvedor para registrar e chamar serviços e procedimentos, essa camada é invocada pelo código gerado pelo compilador, o que torna a execução do aRPC e o fornecimento de serviços e procedimentos transparentes ao programador. Dessa forma, o transporte e eventual execução remota não impactam significativamente a estrutura normal da linguagem. É um dos fundamentos do aRPC manter o desenvolvedor dentro das características da linguagem que a aplicação está sendo desenvolvida e oferecer mais ergonomia que algumas das soluções disponíveis no mercado.

O **Controller** fornece em sua API pública o seguinte conjunto de funções:

- **StartServer** - Função responsável por iniciar o servidor e disponibilizá-lo para recebimento de requisições RPC.

- **StartClient** - Função que inicia a conexão do cliente com os respectivos servidores e o prepara para invocar de forma remota os procedimentos e serviços oferecidos pelos servidores alvo.
- **RegisterService** - Função usada pelo gerador de código para registrar serviços e procedimentos implementados por um servidor aRPC.
- **SendRPCCall** - Função que é chamada pelo gerador de código para efetuar envio de requisições RPC a partir do cliente, esta função cuida da serialização, envio, recebimento de resposta e desserialização dos dados.
- **SendData** - Função usada pelo gerador de código para efetuar o envio de resposta de um RPC executado já com dados serializados para o cliente, essa função se relaciona intimamente com o registro e invocação de serviços e procedimentos.

3.4.3 Código Gerado

O gerador de código é responsável por tornar transparente ao desenvolvedor o transporte e execução remota do RPC. O aRPC tem um foco grande em ergonomia, sendo assim, não é exigido que o programador construa o código base para serialização, envio de dados e interação com o **Controller**. Todo esse código é gerado por uma ferramenta de CLI e seu funcionamento é descrito em detalhes à frente.

O gerador de código produz uma nova função para cada serviço a ser invocado pelo cliente, empacotando a função **controller.SendRPCCall** em chamadas específicas definidas no arquivo de IDL, cuidando da sua declaração e tipagem, tornando dessa forma a definição do RPC invisível ao cliente. Um exemplo de IDL do aRPC pode ser conferido no Código 12.

Código 12 – Exemplo de IDL usada pelo compilador do aRPC para definição de um serviço simples

```
package greetings

// Person e uma struct com os dados de uma pessoa
type Person struct {
    title uint64
    name  text
}

// SayHiRequest e a struct passada como entrada do procedimento SayHi
type SayHiRequest struct {
    person Person
}

// SayHiResponse e a struct passada como saida do procedimento SayHi
type SayHiResponse struct {
    response text
}

// SayHi declaracao do procedimento SayHi
type SayHi func(request *SayHiRequest) (*SayHiResponse, error)
```

Existem no servidor dois blocos de código a serem compostos pelo gerador: o primeiro consiste nas funções oferecidas pelo serviço declarado na IDL e sua interface; o segundo refere-se à função de registro de serviço disponibilizada no código gerado para o servidor. Para construir o primeiro grupo, são construídos códigos que desserializam os argumentos de execução e chamam a função implementada pelo nó. Em seguida, os dados são serializados e enviados usando a função **controller.SendData** para cada uma das funções definidas na IDL. Por fim, para o último grupo, é gerada uma função de registro que recebe uma *struct* que implementa a interface disponibilizada para o servidor. Por último, essa função registra todos os procedimentos e o serviço no **Controler**. Os serviços são registrados com a função hash **FNV-1a** para facilitar o transporte de seu id como **int32** e os procedimentos de um serviço registrados com **uint16** sequencial de seus procedimentos na ordem de geração.

Exemplos de código gerados para clientes e servidores podem ser conferidos nos Códigos 6 e 7 definidos anteriormente.

3.4.4 Mensagens

As mensagens de requisição de aRPC e de resposta são construídas da seguinte forma: o primeiro byte é responsável pelo tamanho do cabeçalho a ser lido. Em seguida, um cabeçalho de tamanho variável é enviado. Por fim, o corpo da mensagem, que tanto pode ser os argumentos para requisição do aRPC, quanto pode ser a resposta, dependendo do cabeçalho.

Figura 10 – Diagramação das mensagens do aRPC

Vale ressaltar que os dados do corpo são serializados para bytes através do sistema de serialização do aRPC. O cabeçalho carrega informações referentes ao serviço, procedimento e tipo de mensagem, como pode ser visto a seguir.

3.4.4.1 Cabeçalho

O cabeçalho do aRPC é composto por quatro campos, **MessageType** que é um **uint8**, **ServiceID** que é do tipo **uint32**, **ProcedureID** que é **uint16** e por fim o **PayloadSize** que é um **uint64**.

Figura 11 – Diagramação do cabeçalho do aRPC

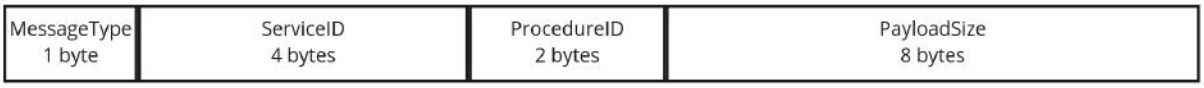


Figura 12 – Diagramação do cabeçalho serializado pelo Colfer

É possível observar, através da comparação da Figura 11 com a Figura 12, que no processo de serialização pelo Colfer, devido às estruturas necessárias à serialização, pode-se diminuir ou aumentar o tamanho do cabeçalho em relação à definição dos tipos de campos. Por isso, apesar da soma dos tamanhos dos campos no cabeçalho ser 15 bytes, o tamanho serializado varia de 1 a 20 bytes, dependendo dos valores enviados, conforme visto na Subseção 3.3.3.

O campo **MessageType** define o tipo de requisição, que pode ser **Call**, **Result** ou **Error**, com seus valores sendo 0, 1 e 2, respectivamente. **Call** é o tipo utilizado em

uma requisição aRPC, nesse caso o conteúdo do corpo são os argumentos. **Result** é o tipo utilizado na resposta de uma requisição aRPC e nesse caso o conteúdo do corpo é a resposta. Por fim se o tipo for **Error**, o conteúdo do corpo é o erro lançado pelo procedimento executado no servidor.

O **ServiceID** descreve qual é o serviço de destino da requisição e, em conjunto com o **ProcedureID**, informa ao **Controller** qual é o procedimento que deve ser executado.

O **PayloadSize** informa ao **Controller** qual é o tamanho do dado serializado a ser recebido, sendo usado para controlar se a mensagem já foi recebida completamente, ou se ainda existem bytes faltando.

Uma característica interessante do cabeçalho é que ele tem tamanho variável apesar de seus campos e tipos serem fixos. O Colfer garante que somente dados preenchidos e diferentes dos valores padrão serão serializados, além disso, estratégias distintas de compressão são utilizadas ao se serializar, para diminuir o tamanho no binário final, dependendo do tipo e tamanho do dado. Tais fatores forçam o **Controller** a receber o tamanho do cabeçalho como primeira informação para saber o momento de desserializá-lo.

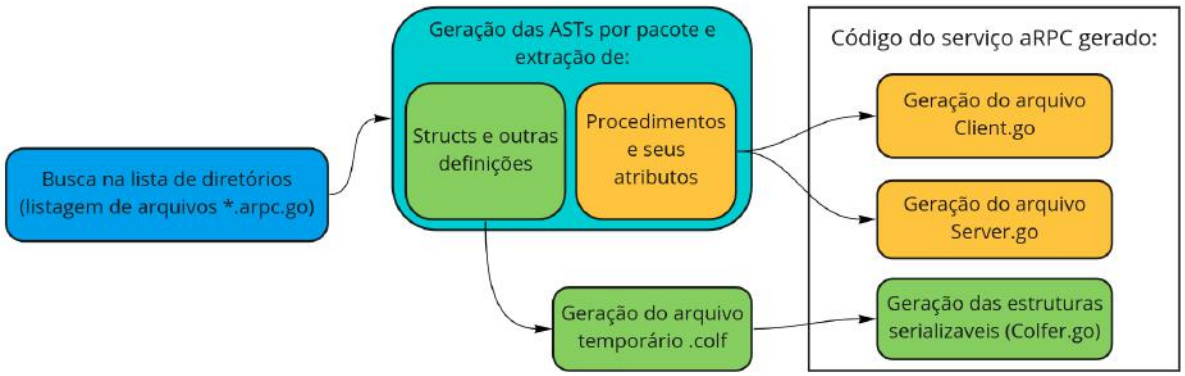
As mensagens no aRPC são enviadas em pares em **Streams** individuais, ou seja, para cada procedimento invocado, uma nova **Stream** é criada e tanto requisição quanto resposta trafegam através desse mesmo objeto. O uso dessa abstração como mecanismo de associação de requisição com resposta dispensa a necessidade do uso de campos de ID no cabeçalho e facilita a implementação do protocolo, gerando um cabeçalho mais enxuto quando comparado ao HPRPC (BAGCI; KARA, 2016).

3.5 GERADOR DE CÓDIGO

O aRPC implementa um gerador de código que processa uma IDL, com gramática igual à linguagem Go, para gerar as implementações dos serviços.

O compilador é dividido em cinco etapas, como pode ser visto na Figura 13.

Figura 13 – Fluxo de compilação feita pelo CLI do aRPC



Na primeira etapa, o compilador percorre uma lista de diretórios, a partir de um caminho fornecido pelo desenvolvedor, buscando pacotes de serviços e lendo suas respectivas definições nos arquivos de tipo **.arcp.go*. Em seguida, uma árvore sintática, onde são identificados os procedimentos remotos e seus atributos, é gerada a partir das assinaturas dos procedimentos. Depois, as definições dos tipos de dado são extraídas, sendo gerado um arquivo *.colf*, sobre o qual é invocado o gerador de código do Colfer, de modo a derivar as *structs* e suas correspondentes funções de serialização. Na quarta etapa, é gerado o arquivo **Client.go** que integra o processo de serialização dos argumentos e respostas com o controlador do aRPC para uso pela aplicação cliente. Por fim, é construído o arquivo **Server.go** para uso na aplicação servidor, que disponibiliza uma função para registrar as implementações dos procedimentos do serviço no controlador, além de integrar a serialização de seus argumentos e respostas.

Existem algumas exigências no protocolo do aRPC que devem ser seguidas pelos arquivos de definição de interface dos serviços. Primeiro, todo procedimento deve retornar um parâmetro de erro, mesmo que a implementação da função não gere erros. Isso ocorre porque no transporte em rede e na execução remota podem ocorrer erros durante a sua execução, os quais devem ser tratados e informados ao cliente. Além disso, os argumentos e resposta devem implementar a interface *Serializable* do aRPC, o que impede o uso de tipos básicos, que precisam ser encapsulados por *structs* para serem transmitidos. Por fim, somente é permitido um argumento e uma resposta por procedimento.

Além de atender aos princípios de simplicidade propostos para o *framework*, essas restrições existem para facilitar a implementação do gerador de código e controlador e, por mais que acarretem em menor flexibilidade e funcionalidades quando comparado com outros protocolos RPC de uso geral, é possível que em um trabalho futuro elas possam ser removidas a fim de melhorar a ergonomia do aRPC. Contudo, não existe nenhum empecilho severo sobre o programador ocasionado por tais limitações, e a maioria delas podem ser contornadas devido à flexibilidade do uso das *structs* de entrada e saída, que permitem maior liberdade na quantidade e tipos de argumentos e resposta.

O gerador de código foi disponibilizado como uma aplicação de *command line interface* (CLI). Dessa forma, é possível instalá-lo no sistema usando somente uma linha de comando no terminal e seu uso é fácil para novos desenvolvedores. A documentação de uso e instalação está disponível no repositório do código fonte da ferramenta no Github (https://github.com/almeida-raphael/arpc_code_generator).

3.5.1 Varredura dos Diretórios e Pacotes

Na etapa de varredura dos diretórios e pacotes, o gerador de código do aRPC recebe o caminho fornecido pelo desenvolvedor e efetua uma varredura recursiva dos diretórios procurando por arquivos de IDL (***.arpc.go**). Após obtenção da lista de arquivos, é feito um agrupamento do conteúdo dos arquivos de IDL em seus respectivos pacotes. Então,

itera-se em cada pacote e, utilizando sua IDL, é construida uma *Abstract Syntax Tree* (AST).

3.5.2 Leitura da árvore Sintática e Geração de *Structs*

A AST é utilizada para identificar os procedimentos a serem disponibilizados pelo serviço, além de seus nomes, tipos de argumentos e de retorno. Para isso, percorrem-se os nós da AST e para cada nó é verificado o seu tipo, para encontrar uma sequência de símbolos que caracterizem um procedimento (ZAYTSEV; BAGGE, 2014). Para cada procedimento são determinados seus atributos, especialmente seu nome, o nome dos argumentos, o tipo dos argumentos e o tipo de retorno. Depois de devidamente identificados e processados, os nós referentes aos procedimentos são removidos da AST e os remanescentes são traduzidos novamente em um arquivo temporário **.colf**.

Os arquivos **.colf** são arquivos de IDL usados pelo Colfer, o serializador do aRPC, para gerar as *structs* relevantes aos serviços e procedimentos definidos nos arquivos ***.arpc.go**. Uma vez construídos os arquivos temporários ***.colf**, o gerador de código do aRPC invoca a ferramenta de CLI do Colfer, passando como argumento o local onde devem ser gerados os pacotes, tamanhos máximos de estruturas e listas, e o arquivo ***.colf** de entrada. Esse procedimento é repetido para cada pacote identificado e processado nos arquivos ***.arpc.go**.

3.5.3 Geração de código aRPC para clientes

A geração do código dos procedimentos do serviço para o arquivo **Client.go** precisa somente dos dados referentes aos procedimentos disponíveis, o nome do pacote de origem e *templates* para geração de código.

Estes dados são incorporados em um *template* de chamada remota de procedimento, de modo a gerar segmentos de código referentes às funções que podem ser invocadas pelo cliente. Para cada segmento são substituídas no *template* as informações coletadas anteriormente como: nome do pacote, nome do serviço, nome do procedimento, nome dos argumentos, tipo dos argumentos e tipo de retorno. O fluxograma do processo pode ser visto na Figura 15.

O código gerado para cada procedimento é responsável por invocar a serialização dos argumentos, e chamar a função do controlador encarregada pelo envio dos dados serializados e do recebimento da resposta. Por fim, o código recebe o dado desserializado da resposta e o retorna para função que lhe invocou.

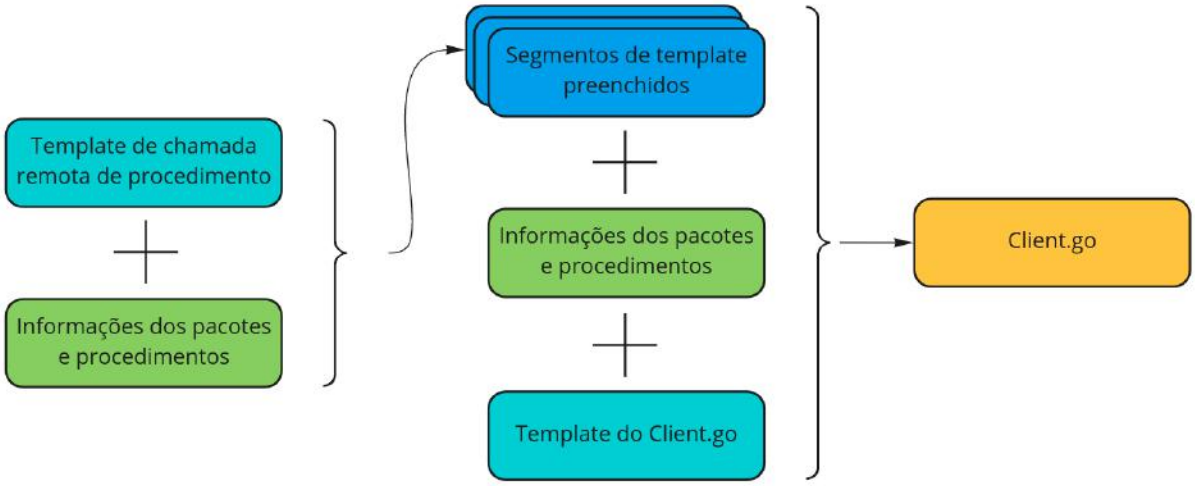
Para gerar o arquivo **Client.go** são incorporados os dados coletados do arquivo de IDL (***.arpc.go**) e os segmentos de código gerados para os procedimentos no *template* do serviço.

Figura 14 – Exclusão do nó de função da AST do arquivo de IDL (**.arpc.go**)

3.5.4 Geração de código aRPC para servidores

A geração do código dos procedimentos do serviço para o arquivo **Server.go** segue de forma análoga ao **Client.go**. Entretanto, existem algumas características distintas. No servidor, deve-se implementar uma interface que define todos os procedimentos oferecidos pelo serviço em uma *struct*, em seguida essa *struct* deve ser registrada no controlador. Dessa forma, além dos códigos que fazem o elo entre a implementação e serialização, o gerador constrói a interface do serviço a ser implementada pelo desenvolvedor, e uma função para efetuar o registro das *structs* que estendem essa interface no controlador do servidor.

Figura 15 – Processo de geração do arquivo Client.go no aRPC



Apesar da interface de registro e da função que faz o elo entre o controlador e as implementações serem diferentes do arquivo do cliente, as informações necessárias para o preenchimento de seus *templates* são as mesmas, sendo a única diferença o seus códigos base.

Para gerar o arquivo **Server.go** são incorporados os dados coletados do arquivo de IDL (***.arpc.go**) e os segmentos de código gerados para os procedimentos no *template* do serviço.

3.6 USO DO ARPC

A proposta do aRPC é ser um protocolo rápido para HPC (*high performance computing*) que impacte o mínimo possível na ergonomia da escrita da linguagem. Por essa razão o único momento onde o programador deve se preocupar com definições do aRPC é na inicialização da aplicação, onde ele fará a lógica de carregar os certificados, configuração do endereço de conexão, e configurações de **Channel**. Após definidos esses pontos, o cliente deve obter uma referência para o serviço que se deseja executar e por fim chamar **controller.StartClient()**. A partir desse momento, todas as chamadas das funções de RPC e programação da aplicação seguem o fluxo usual de desenvolvimento. Para o servidor, ao invés de gerar uma referência para o serviço, o programador deve implementar a interface para o mesmo e por fim, registrá-lo no controlador usando a função de registro do serviço gerada e então, executar **controller.StartServer()**. No Código 13 é possível observar um exemplo dessa inicialização para um serviço simples.

Código 13 – Inicialização para o cliente em um serviço simples no aRPC

```
package main

import (
    "github.com/almeida-raphael/arpc_examples/examples/greetings"
    "log"
)

func main(){
    rootCAPath := os.Getenv("CA_FILE")
    serverAddress := os.Getenv("SERVER_ADDRESS")

    rootCA, err := LoadCA(rootCAPath)
    if err != nil {
        log.Fatal(err)
    }

    tlsConfig := tls.Config{
        RootCAs:    rootCA,
        NextProtos: []string{"quic-arcp"},
    }

    aRPCController := controller.NewRPCController(
        channel.NewQUICChannel(
            serverAddress,
            7653,
            &tlsConfig,
            &quic.Config{
                MaxIncomingStreams: 100000,
            },
        ),
    )

    greetingsService := greetings.NewGreetings(&aRPCController)
    err := aRPCController.StartClient()
    if err != nil {
        log.Fatal(err)
    }

    hiResponse, err := greetingsService.SayHi(&requestData)

    // O tratamento de erros e resposta deve continuar abaixo

}
```

Vale ressaltar que todas as funções e procedimentos que utilizam rede, disponíveis no sistema do aRPC, fornecem um argumento opcional do tipo **ctx.Context** para gerenciamento do tempo limite e cancelamento de rotinas em execução. Assim, o programador pode escolher usar esses argumentos para ter mais controle da aplicação, ou omiti-los para um uso mais transparente do RPC.

4 MÉTODO EXPERIMENTAL E ANÁLISE DE RESULTADOS

Para avaliar o desempenho foram executados vários casos de teste, todos eles implementados utilizando o aRPC e para efeitos de comparação, também implementados com o gRPC, tais testes são descritos na Seção 4.1. A partir desses testes foram coletadas uma série de métricas de interesse, explicitadas na Seção 4.2. Os testes foram executados numa LAN fechada, entre dois computadores físicos, sendo um o cliente e outro o servidor, as características de *hardware* e alguns detalhes de *software* relativos à execução dos testes são levantados na Seção 4.3. Por fim, é feita uma análise dos dados obtidos em diversos cenários de execução, tal como a comparação de desempenho entre trafegar dados sem perda de pacotes ou com perda de pacotes e dentro e fora de estrutura de nuvem. Os resultados podem ser conferidos na Seção 4.4.

4.1 CASOS DE TESTE PROPOSTOS

Parte dos testes construídos foram baseados no trabalho de (BAGCI; KARA, 2016) e outra parte foi desenvolvida como validação adicional. Neste sentido, os experimentos implementados baseados no artigo do HPRPC foram:

- **Média**: Na referência essa operação, chamada de **Average**, recebe uma *struct* contendo uma lista de **int32** e retorna a média dos números recebidos em **double**. Esse teste no artigo do HPRPC tem o propósito de avaliar o comportamento do *framework* em requisições com características discrepantes de tamanho, onde a entrada é grande e a saída pequena.

- **Gerador de Números Aleatórios**: Na referência essa operação é chamada de **GetRandNums** e recebe um único número **int32** como entrada, que descreve uma quantidade. Em seguida é gerada uma lista de números pseudo aleatórios (**int32[]**) com tamanho equivalente à quantidade recebida. Esta operação foi usada para avaliação do HPRPC no cenário de requisições com características discrepantes de tamanho, onde a entrada é pequena e a saída grande.

- **Ecoar**: Na referência essa operação é chamada de **SendRcvLargeData** e recebe uma lista de **int32** e retorna a mesma lista de **int32**. Este teste funciona como um *echo server* e é utilizado para testes onde a entrada e a saída tem tamanhos equivalentes, avaliando o HPRPC em um cenário onde as requisições e respostas são igualmente grandes.

No trabalho sobre o HPRPC, os autores explicitam que foram feitas otimizações no serializador a fim de gerar ganhos no caso da transmissão de *arrays* de tipos básicos, por

esta razão, a maior parte dos testes propostos usam *arrays* de **int32**, que justifica os bons resultados do *framework* frente ao gRPC. Entretanto, tais testes são incompletos no sentido de comprovar a eficácia do HPRPC como um *framework* RPC para uso no contexto mais geral de HPC. Por essa razão, a avaliação proposta pelo autores foi complementada neste trabalho com testes de velocidade e tamanho de serialização de requisições e respostas para todos os tipos básicos suportados pelo Colfer e o Protobuffers.

Além disso, também foi efetuado um novo teste que recebe como argumento um *array* de *structs*, essas structs possuem todos os tipos suportados pelo Protobuffers e Colfer e o experimento foi nomeado de **Todos os Tipos**. A justificativa para este novo teste é trazer para a avaliação do *framework* de RPC informações que auxiliem na medição do desempenho em um caso de uso menos sintético. Outro ponto de melhoria foi a extensão das medições para dados maiores, em um contexto de HPC faz sentido com que os dados transferidos ultrapassem a casa dos kilobytes, entretanto, os testes do HPRPC se limitam a quantidades de dados inferiores a dez mil elementos em **int32**, ou seja, igual ou menor que 40KB.

4.2 MÉTRICAS DE INTERESSE

Em todos os testes foram coletados os tamanhos da requisição e da resposta, e seus tempos serialização e desserialização, além das durações totais e de execução dos procedimentos. Usando esses dados é possível calcular o tempo de protocolo e transferência avaliando a seguinte equação:

$$T_{TXPR} = T_{TOT} - T_{SERQ} - T_{DSRQ} - T_{SERE} - T_{DSRE} - T_{EX} \tag{4.1}$$

Onde:

$$\begin{aligned} T_{TXPR} &= \textit{tempo de transmissão do protocolo} \\ T_{TOT} &= \textit{tempo total} \\ T_{SERQ} &= \textit{tempo de serialização da requisição} \\ T_{DSRQ} &= \textit{tempo de desserialização da requisição} \\ T_{SERE} &= \textit{tempo de serialização da resposta} \\ T_{DSRE} &= \textit{tempo de desserialização da resposta} \\ T_{EX} &= \textit{tempo de execução} \end{aligned}$$

Também é possível calcular o tamanho total referente aos dados serializados comparativamente entre o aRPC e o gRPC através da razão entre os tamanhos em bytes de

ambos. Além disso, foi avaliada a vazão de dados a partir da razão entre o tempo completo de requisição desde seu inicio até sua resposta, excluindo a duração da execução do procedimento. Somado a essas avaliações, foi analisada a quantidade de bytes trafegados na rede. O tempo de execução absoluto também foi calculado usando como métrica razão dos tempos entre o aRPC e o gRPC. Todos os dados foram coletados para uma quantidade crescente de elementos nas *structs* de requisição e resposta. Como o aRPC foi projetado para ser usado no contexto de HPC, é importante que ele escale bem conforme a quantidade de dados aumente.

Foi efetuada uma avaliação de cada teste com 1, 2, 4, 8 e 16 *threads*, entretanto, observou-se que a interface de rede Gigabit fica saturada a partir de duas *threads*, momento em que o desempenho dos protocolos avaliados não apresenta mais divergências relevantes. Portanto os resultados apresentados contam com execuções utilizando apenas uma *thread*.

Com relação às métricas de captura de tempo, foi utilizada a função **time.Now()** da biblioteca padrão da linguagem Go (TIME. . . , 2021). De acordo com a documentação da biblioteca, esta função proporciona precisão de nanosegundos na medição do tempo. No caso específico deste trabalho, onde o sistema operacional utilizado foi o Linux, pode-se conferir que o Go faz um *link* com a função **clock_gettime** do sistema, conforme visto em (GOLANG/GO, 2021), a documentação para esta chamada de sistema no Linux pode ser conferida em (CLOCK_GETTIME(2). . . , 2021). Tendo em vista tais características, a função escolhida para medição de tempo apresenta acurácia necessária para os propósitos deste trabalho. Apesar da precisão de tempo para a função de captura ser em nanosegundos, as métricas foram avaliadas apenas em escalas de micro e milissegundos, de modo a evitar possíveis ruídos.

4.3 INFRAESTRUTURA DE EXECUÇÃO

As decisões relativas à infraestrutura de execução foram divididas em duas partes, *hardware* e *software*, cujas características são descritas na Seção 4.3.1 e na Seção 4.3.2.

4.3.1 Hardware

Para avaliar o *framework* foram executados testes em três ambientes, que são descritos adiante e têm como propósito dar às avaliações maior abrangência e detectar possíveis problemas associados aos contextos de teste. Os ambientes utilizados foram:

a) Máquinas pessoais:
   - Processador do cliente: Intel Core i7-8700k @ 3.70GHz
   - Memória do cliente: 32GB DDR4
   - Processador do servidor: Intel Core i7-3770K @ 3.50GHz
   - Memória do servidor: 16GB DDR3

  - Rede 1 Gigabit

b) Nuvem Microsoft Azure:
  - Máquina do tipo Standard F4s v2
  - Processador: Intel Xeon Platinum 8272CL @ 2.60GHz
  - Memória RAM: 8GB DDR4
  - Rede 10 Gigabits

c) Nuvem Google Cloud Platform:
  - Processador Intel Xeon E5-2699 v3 @ 2.30GHz
  - Memória RAM: 6GB DDR4
  - Rede 10 Gigabits

4.3.2 Software

Do ponto de vista de software, muitas melhorias e automações foram implementadas a fim de possibilitar testes maiores e mais complexos sem necessidade de intervenção humana. Os seguintes artefatos foram projetados: uma arquitetura de compilação; escalonadores de execução; exportação de métricas; e um *script* de processamento para geração dos gráficos e extração das métricas. Primeiro as máquinas são preparadas com os arquivos do projeto, em seguida são compilados os executáveis de cada teste (cliente e servidor), então são configurados os escalonadores com o número de amostras por experimento, intervalo de *threads* a serem avaliados, e quais testes devem ser executados, e no caso do escalonador do cliente, também é configurado o endereço do servidor.

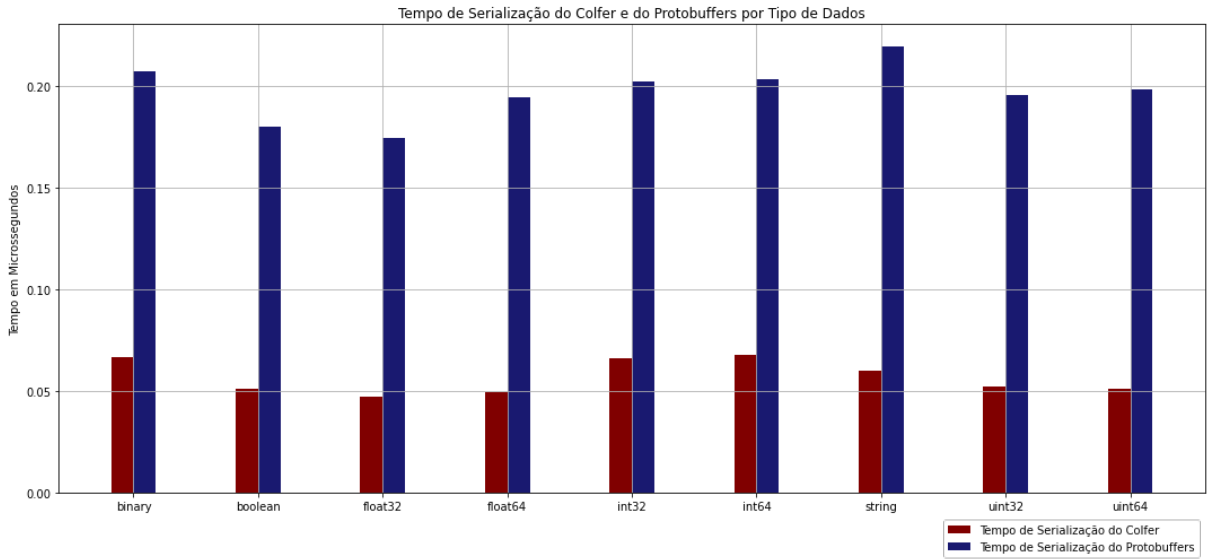
Após configuração do escalonador do cliente e do servidor, os testes são executados em ordem de número de *threads*, e amostras, e cada experimento é executado exatamente na quantidade de amostras especificada. O escalonador inicia o servidor correto e em seguida o cliente se conecta, ao fim do teste, ambos os *endpoints* salvam os dados capturados em arquivos **.json**. No **.json** do servidor são armazenados os tempos de execução dos procedimentos, enquanto no **.json** do cliente são escritos os demais dados. Ao fim dos testes, os arquivos são colocados em um google drive, que é acessado por um *script* do Google Colab que faz processamento dos dados e extração das métricas relevantes e geração dos gráficos disponíveis neste trabalho.

Para que a serialização e as medições sejam comparáveis, foi necessário executar um grande número de amostras (por volta de duas mil por teste) onde em cada uma delas foi gerada de forma automática um valor pseudo aleatório para cada campo. Essa estratégia foi empregada, pois dependendo do tamanho do dado, tanto o Protobuffers quanto o Colfer, usam estratégias de serialização distintas e nesse sentido, não seria possível comparar os resultados de forma consistente entre os testes.

Outro ponto que vários autores não se atentam é para o fato que o gRPC não possui como parte obrigatória o TLS, e por essa razão para que a execução seja segura e as comparações façam sentido, foi necessário utilizar TLS sobre gRPC, que segue o mesmo paradigma que TLS sobre HTTP/2. No caso do protocolo QUIC, usado pelo aRPC, o uso de TLS é nativo e integrado, sendo assim, a comunicação é sempre encriptada e autenticada. Com isso, todos os testes utilizaram TLS tanto no aRPC quanto no gRPC, mesmo que no caso dos nossos testes não tenham sido trafegados dados sensíveis. Desta forma, a comparação entre os dois *frameworks* de RPC é mais adequada.

Devido à quantidade de testes executados e ao grande número de configurações necessárias, essa tarefa se mostrou desafiadora. Na configuração atual, a quantidade de dados que deve ser trocada para garantir validade estatística, somado à velocidade das interfaces de rede fez com que os testes tivessem longas durações. A ordem de grandeza da quantidade de execuções é na casa de 10^6 chamadas de RPC, cada uma executando I/O (*input*/*output*) e variando-se a quantidade de elementos de 128 a 65536, que por sua vez variam de 4 a 2085 bytes. Estima-se, considerando que todos os testes rodam com 2000 amostras, que um experimento pode chegar a transmitir entre 0,98 MiB e 273,29 GiB de dados.

Figura 16 – Tempo de serialização



4.4 RESULTADOS OBTIDOS

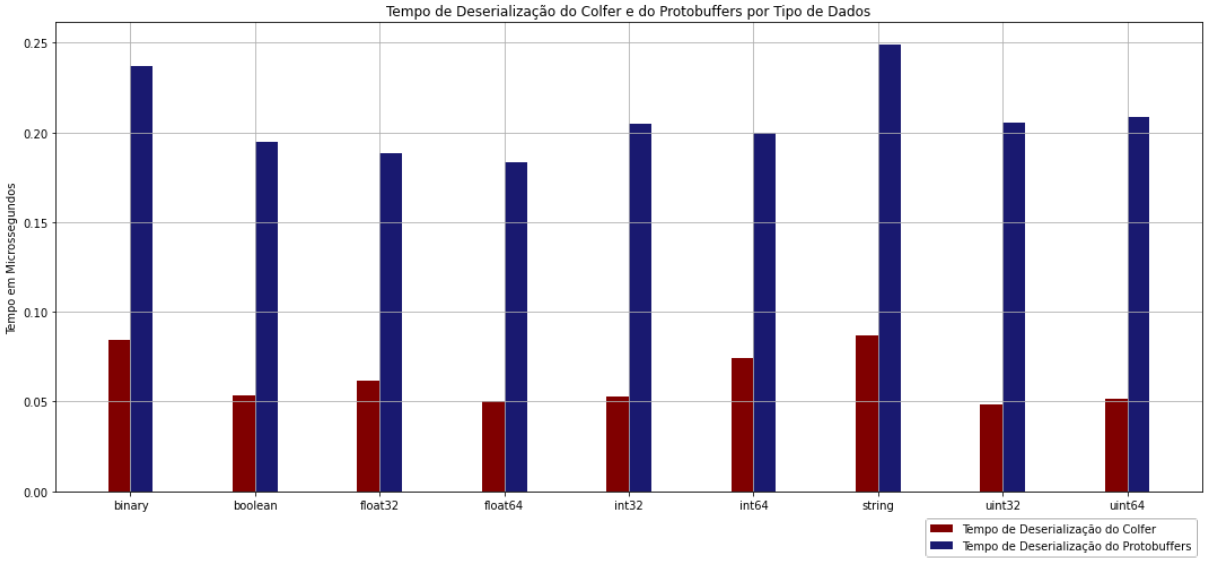
De acordo com as métricas de interesse levantadas anteriormente, é possível dividir a análise dos resultados em três etapas. Uma etapa referente à serialização dos dados, outra etapa relacionada com o transporte dos dados pela rede e finalmente, uma etapa que engloba as características de desempenho do *framework* de maneira mais completa.

4.4.1 Serialização

Para avaliar as métricas de serialização e desserialização, foram separados dois tipos de teste distintos, a realização das operações para os tipos primitivos isoladamente e os experimentos mencionados anteriormente.

4.4.1.1 Dados primitivos

Figura 17 – Tempo de desserialização



Foram realizadas medições do tempo de serialização, tempo de desserialização e tamanho dos dados serializados em ambos os serializadores, Colfer e Protobuffers. Todos os tipos primitivos compartilhados por ambos os serializadores foram testados. Estes são: **int32**, **int64**, **uint32**, **uint64**, **float32**, **float64**, **bool**, **binary** e **string**. Para a coleta desses dados, foram efetuados um milhão de testes por tipo de dado, em cada execução foi testada a serialização e deserialização de um valor para cada tipo. Os valores foram gerados aleatoriamente para cada execução e em seguida foi calculada a média dos tempos e tamanhos finais. Essa abordagem foi adotada porque os serializadores, tanto Protobuffers quanto Colfer, usam estratégias de serialização distintas para cada tipo e tamanho de dado para melhor otimizar o binário serializado. Dessa forma, com a abordagem aleatória e a média de uma grande quantidade de testes, é possível capturar a tendência para o caso de uso geral de cada tipo de dado em cada um dos serializadores. Essas três métricas podem ser conferidas nas Figuras 16, 17 e 18, respectivamente.

Conforme visto nas Figuras 16 e 17, é possível perceber que o Colfer consegue serializar os tipos primitivos de maneira mais rápida do que o Protobuffers, o ganho de *speedup* pode ser conferido na Tabela 1.

Figura 18 – Tamanho dos dados serializados

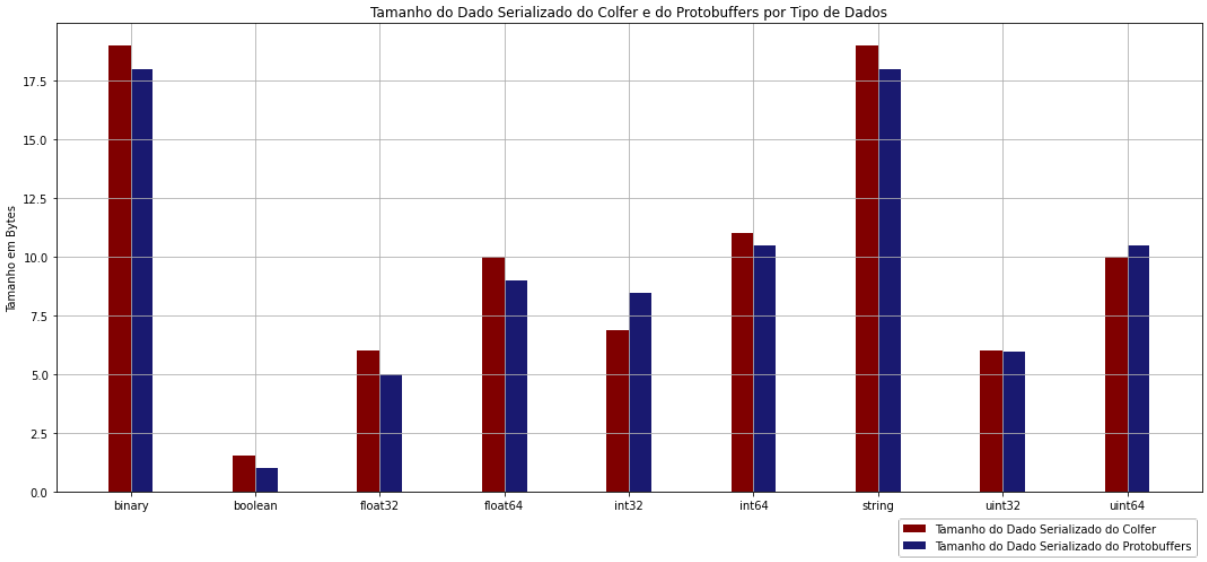


Tabela 1 – Speedups de serialização e desserialização obtidos para os tipos primitivos

Tipos	Speedup serialização	Speedup desserialização
int32	3,0723	3,886
int64	3,009	2,696
uint32	3,762	4,227
uint64	3,902	4,046
float32	3,730	3,050
float64	3,936	3,659
binary	3,108	2,799
bool	3,538	3,659
string	3,676	2,861

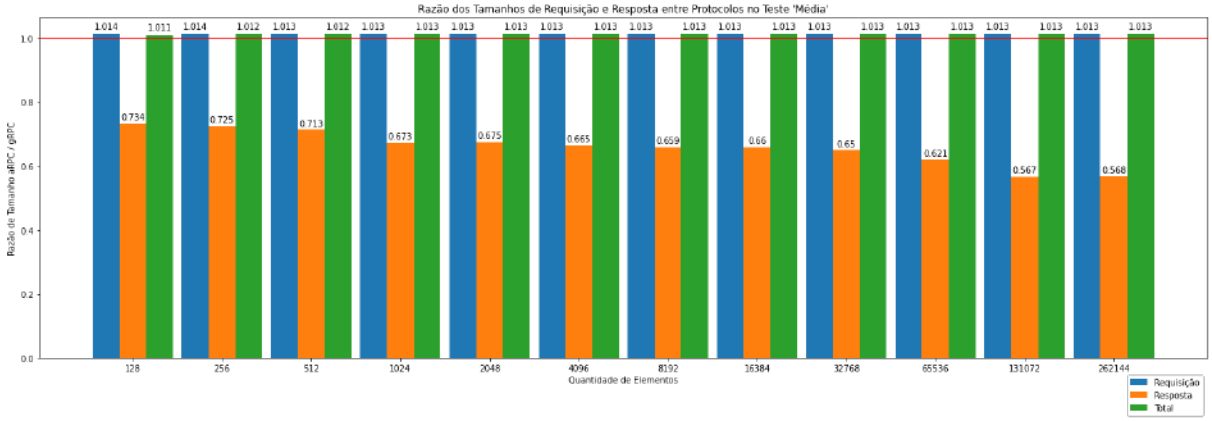
De acordo com a Figura 18, que apresenta o tamanho final dos dados serializados, é possível notar que o Protobuffers serializa os dados dos tipos **binary**, **boolean**, **float32**, **float64**, **int64** e **string** com uma quantidade menor de bytes, enquanto que os resultados são equivalentes para o caso **uint32** e que os dados de tipo **int32** e **uint64** são serializados pelo Colfer com tamanhos finais menores.

4.4.1.2 Casos de teste

Também foram avaliados o desempenho de serialização, com relação à razão entre o tamanho dos dados gerados pela serialização do aRPC e do gRPC nos casos de teste propostos: **Média**, **Gerador de Números Aleatórios**, **Ecoar** e **Todos os Tipos**. Cabe ressaltar que o teste **Todos os Tipos** recebe uma resposta vazia.

No teste **Média**, referente à Figura 19, é possível observar que os tamanhos em bytes do aRPC transmitidos em cada procedimento (requisição + resposta) são de 1,1% a

Figura 19 – Tamanho dos dados para o teste **Média**



1,3% maiores que o gRPC. Ou seja, o Colfer, apesar de muito mais rápido na serialização e desserialização, gera para este caso de testes dados ligeiramente maiores que o Protobuffers.

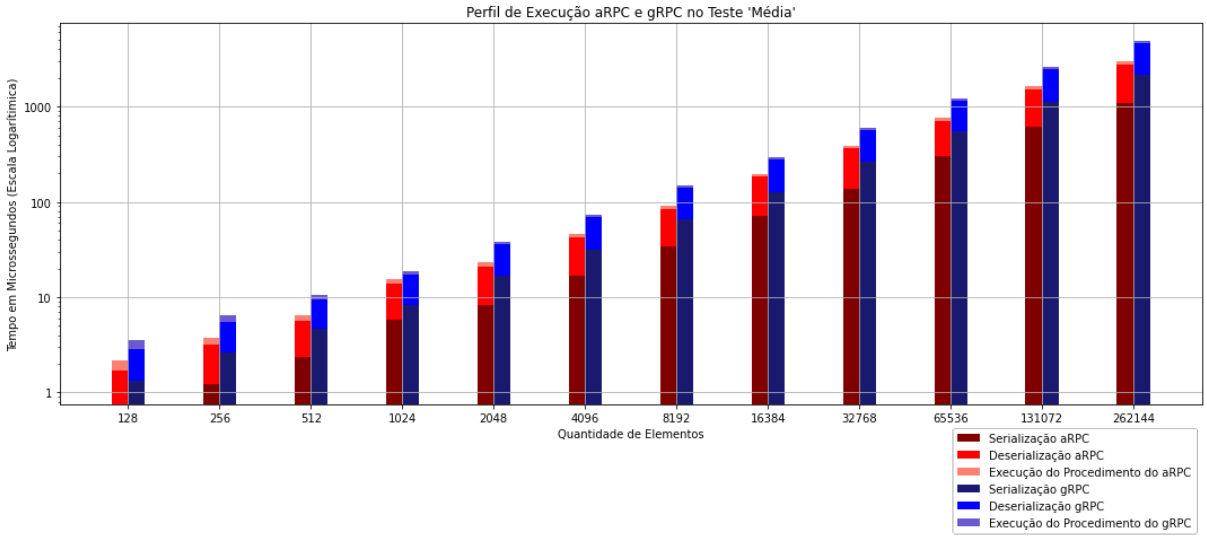
Outra característica interessante que pode ser percebida na Figura 19 é que o tamanho da resposta do aRPC em relação ao gRPC tende a diminuir com o aumento do número de elementos presentes na requisição. Isso ocorre porque conforme a quantidade de elementos aumenta, fica mais provável que a média esteja próxima do centro do intervalo, que é o número zero. Nessa situação observa-se na Figura 18 que o Colfer é mais eficiente que o Protobuffers na serialização de **int32**. Após uma análise, foi possível concluir que isso ocorre devido a uma ineficiência de serialização de valores negativos no Protobuffers.

Todos os valores negativos são serializados com 11 bytes no gRPC, independente do valor absoluto do número, enquanto que no aRPC os valores negativos são serializados usando cada vez menos bytes conforme o valor dos números negativos se aproxima de zero. Como a divergência dos tamanhos nos números negativos entre os dois protocolos é mais acentuada próxima ao centro do intervalo, o número de elementos mais eficientemente codificados pelo aRPC aumenta em relação ao gRPC nesse campo proporcionalmente ao tamanho da resposta, gerando o comportamento observado. Espera-se que esse comportamento se estabilize eventualmente conforme o número de elementos aumenta nos testes.

É interessante observar que, apesar do teste **Média** no aRPC ter gerado dados levemente maiores que o gRPC, o ganho referente aos tempos de serialização e desserialização são significativos, como pode ser visto na Figura 20. O menor ganho, nesse grupo de dados, é referente à quantidade de 512 elementos que teve *speedup* de 1,48, já o maior é referente ao caso com 128 elementos, que teve *speedup* de 2,87. Dessa forma, os ganhos em tempo de processamento na serialização ajudam a compensar a sobrecarga do envio de mais bytes pela interface de rede.

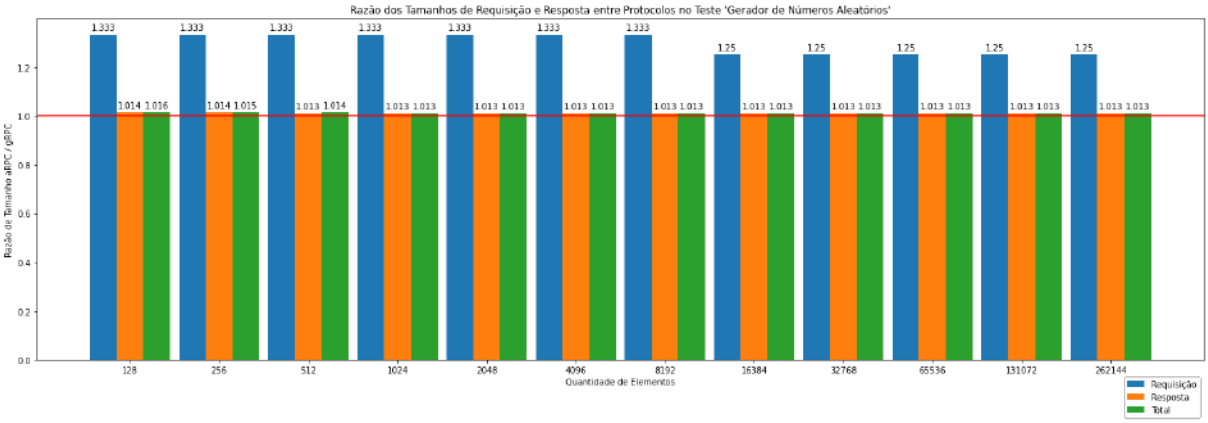
Quanto ao caso de teste **Gerador de Números Aleatórios**, na Figura 21, é inte-

Figura 20 – Perfil de execução para o teste **Média**



ressante notar que o tempo total é dominado pela resposta, em contrapartida ao caso de teste anterior (**Média**) no qual a relevância maior foi do tempo de requisição. Quanto ao tamanho das requisições serializadas serem maiores no Colfer do que no Protobuffers, isso se dá pelo fato dos números inteiros positivos serem codificados de maneira mais otimizada no Protobuffers. Cabe ressaltar que os dados presentes na resposta também são todos números positivos, caso contrário o resultado seria mais favorável ao Colfer, tendo em vista que o mesmo otimiza melhor o tamanho quando os testes apresentam números negativos.

Figura 21 – Tamanho dos dados para o teste **Gerador de Números Aleatórios**



Na Figura 22 é possível observar que no teste **Gerador de Números Aleatórios** o Colfer é mais eficiente para todas as quantidades de dados testadas, tendo *speedups* nos tempos de execução variando de 1,19 para 4096 elementos a 2,25 para 128 elementos. Como justificado anteriormente, esse ganho ajuda a compensar a leve diferença de tama-

nho nos dados transmitidos. Além disso, é relevante ressaltar que o tempo de execução deste teste é maior que todos os outros e a quantidade de recursos consumidos é significativamente maior para a geração dos números aleatórios. Este fato é relevante para os resultados comparativos do tempo de execução do aRPC contra o gRPC neste teste e será retomado à frente.

Figura 22 – Perfil de execução para o teste **Gerador de Números Aleatórios**

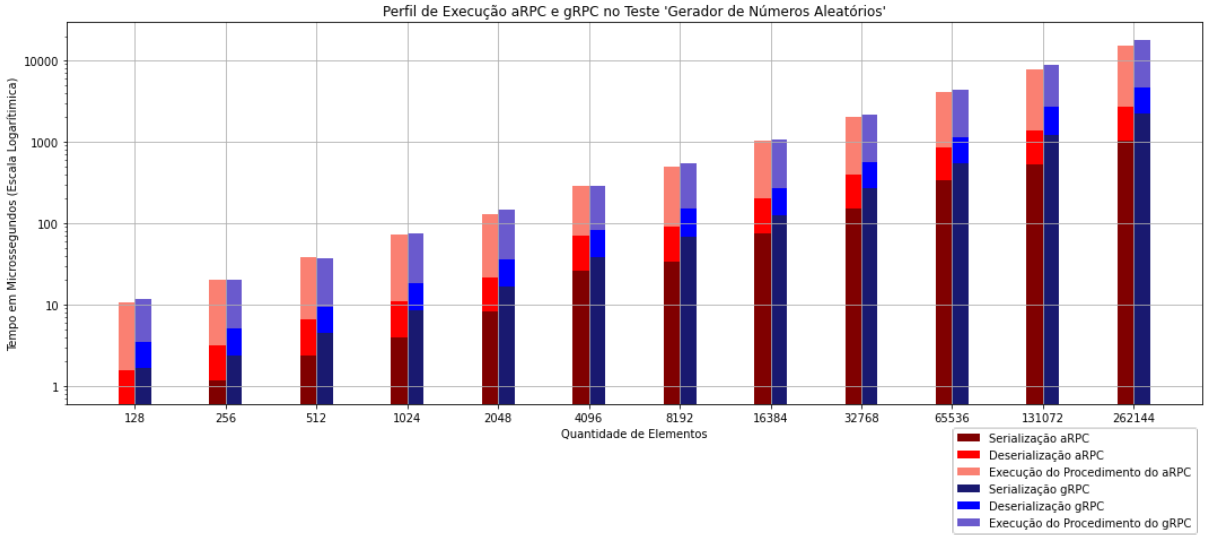
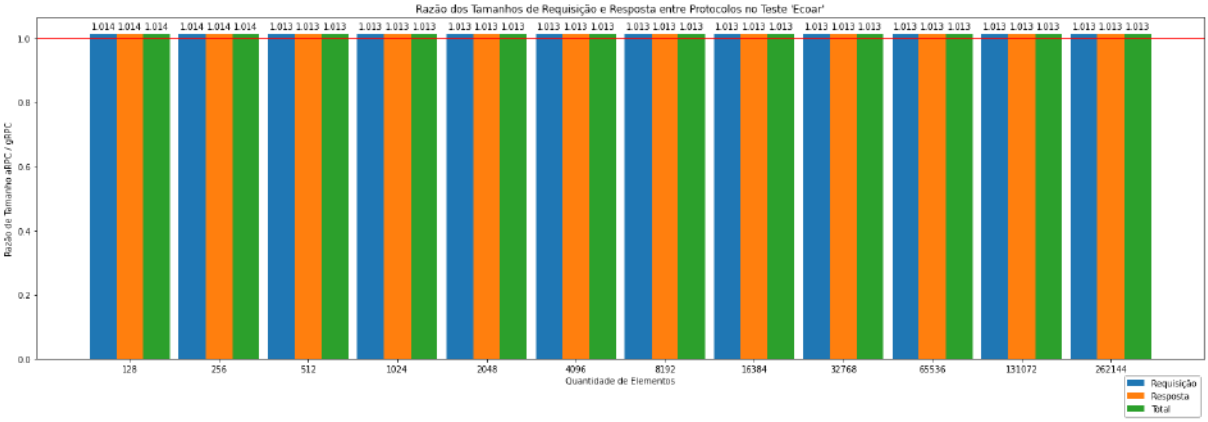


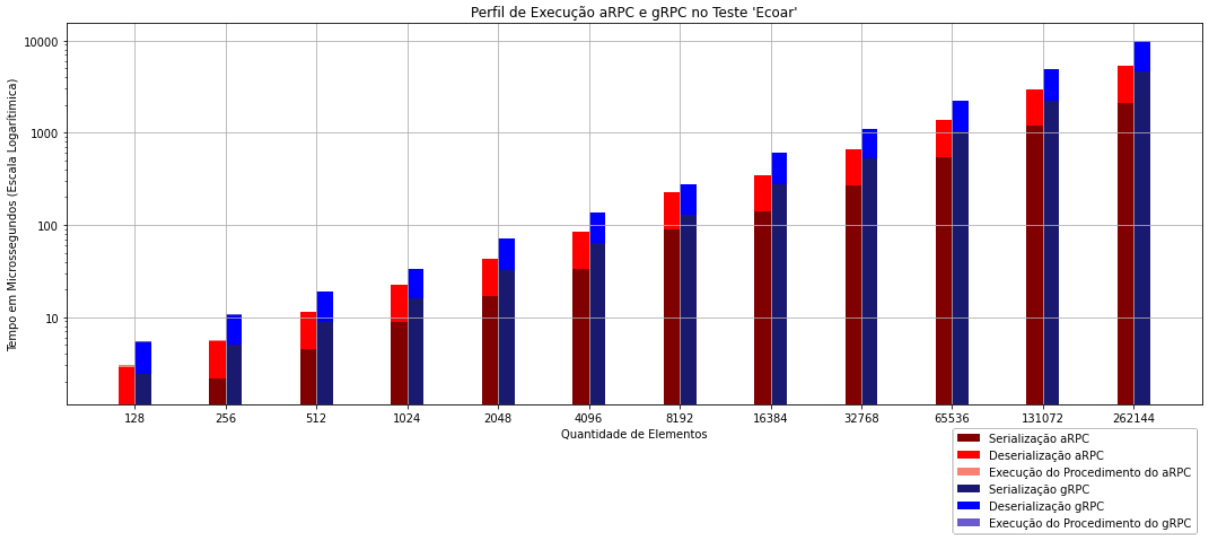
Figura 23 – Tamanho dos dados para o teste **Ecoar**



Nas Figuras 23 e 24 podem ser vistas as análises de tamanhos e de tempos para o teste **Ecoar**. Neste teste o Colfer gerou dados 1,3% a 1,4% maiores que o Protobuffers, entretanto, com *speedups* variando de 1,47 para 1024 elementos a 1,94 para 256 elementos.

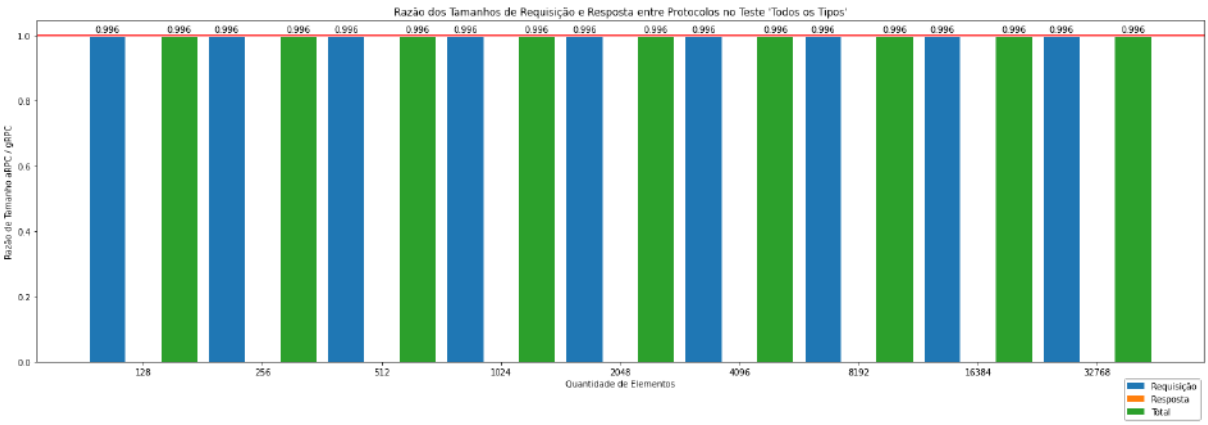
Na Figura 25 é possível observar o primeiro caso onde o serializador do Colfer gerou dados menores que o Protobuffers para requisição, resposta e o total, para todas as quantidades de elementos testadas. Esse caso de teste foi desenvolvido a fim de averiguar os serializadores e protocolos em estruturas de dados heterogêneas, com o propósito de

Figura 24 – Perfil de execução para o teste **Ecoar**



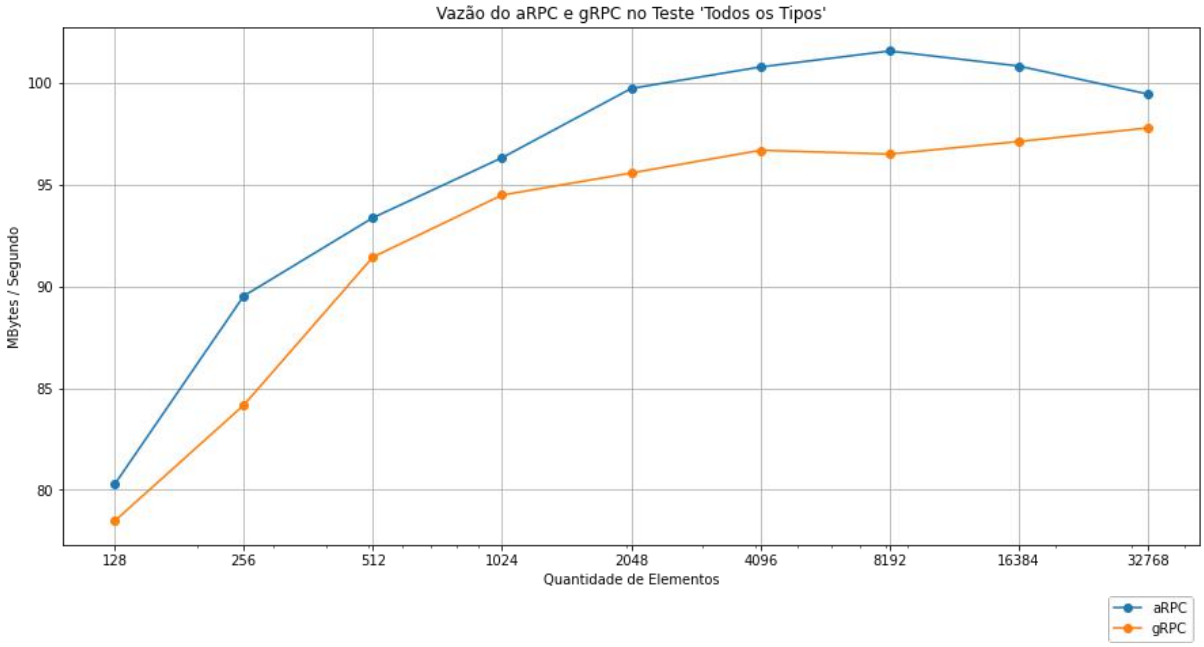
torná-lo mais representativo dos casos de uso reais. Nesse sentido, quando uma estrutura desse tipo é utilizada, o Colfer não somente mantém os *speedups* altos em relação ao Protobuffers, nos tempos de serialização e desserialização, mas ainda gera um dado serializado em média 0,4% menor, o que resulta em ganhos expressivos no tempo de execução e vazão do aRPC, como pode ser visto na Figura 26. Nesses testes é possível ver, através de análises da Figura 27, que os *speedups* variam de 1,48 para 512 elementos a 2,87 para 128 elementos.

Figura 25 – Tamanho dos dados para o teste **Todos os Tipos**



Através dessas análises, pode-se observar que o Colfer como serializador é consistente melhor em relação ao *speedup* de serialização e desserialização perante o Protobuffers, com valores variando de 1,47 a 2,87. Outro fator relevante é que com o aumento exponencial da quantidade de dados, o crescimento do tempo de execução também é exponencial. Indicando o aumento do tempo de serialização e desserialização é linear de acordo com o

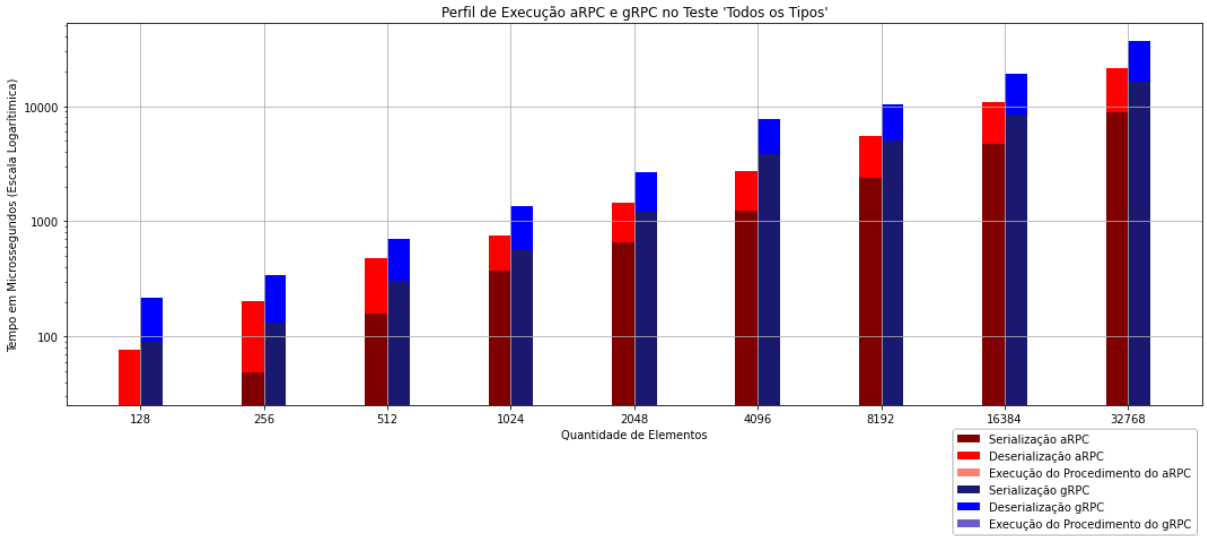
Figura 26 – Vazão do aRPC e do gRPC para o caso de teste **Todos os Tipos**



tamanho da entrada.

Nos testes onde o Colfer teve um tamanho de dados superior ao Protobuffers, esta sobrecarga nos tamanhos variou de 1,1% a 1,6%, enquanto que no caso de teste **Todos os Tipos** a redução de tamanho foi de 0,4% em relação ao Protobuffers, ressaltando que este teste é mais próximo dos casos de uso reais. Logo, o *speedup* de serialização e desserialização e eventuais ganhos no protocolo de transporte devem compensar esta sobrecarga para que o aRPC tenha desempenho melhor que o gRPC.

Figura 27 – Perfil de execução para o teste **Todos os Tipos**



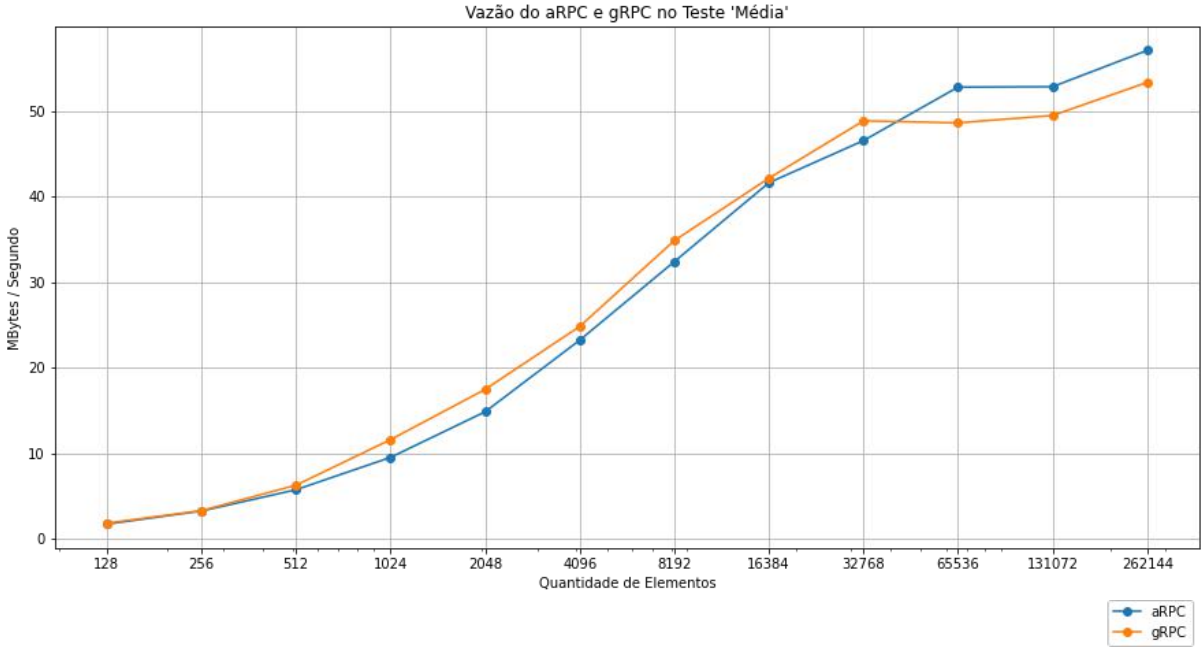
4.4.2 Transporte

Nesta seção foram feitos os testes relacionados ao transporte dos dados em cenários com redes diferentes. Os ambientes foram: conjunto de máquinas pessoais numa LAN fechada, com rede 1 Gigabit e nuvens Microsoft Azure e Google Cloud Platform, ambas com rede de 10 Gigabits

4.4.2.1 Máquinas pessoais

Na Figura 26 para o caso de teste **Todos os Tipos** é possível conferir que, devido aos dados mais heterogêneos, o aRPC se distancia do gRPC desde o início, até saturar a interface de rede quando os dois tendem a se aproximar. Enquanto nos demais testes, Figuras 28, 29 e 30, para os casos de teste **Média**, **Gerador de Números Aleatórios** e **Ecoar**, por se tratarem apenas de dados do tipo **int32** positivos, o gRPC começa com vantagem, porém, conforme o número de elementos aumenta, o aRPC tende a superá-lo.

Figura 28 – Vazão do aRPC e do gRPC para o caso de teste **Média**



Um teste foi executado simulando perda de pacotes na rede somente no ambiente de máquinas pessoais, devido ao maior controle sobre os parâmetros da rede. Para simular a perda de pacotes, foi executado o comando **sudo tc qdisc add dev enp0s31f6 root netem loss 5%**, utilizando o módulo do kernel **sch_netem**, conforme (CARDWELL, 2021). Esta perda de 5% dos pacotes pode ser verificada utilizando o programa **ping**, conforme pode ser conferido nas Figuras 31 e 32.

Na Figura 33 é possível conferir os resultados encontrados para o aRPC e gRPC, de maneira comparativa, entre os cenários com perda de pacotes e sem perda de pacotes.

Figura 29 – Vazão do aRPC e do gRPC para o caso de teste **Gerador de Números Aleatórios**

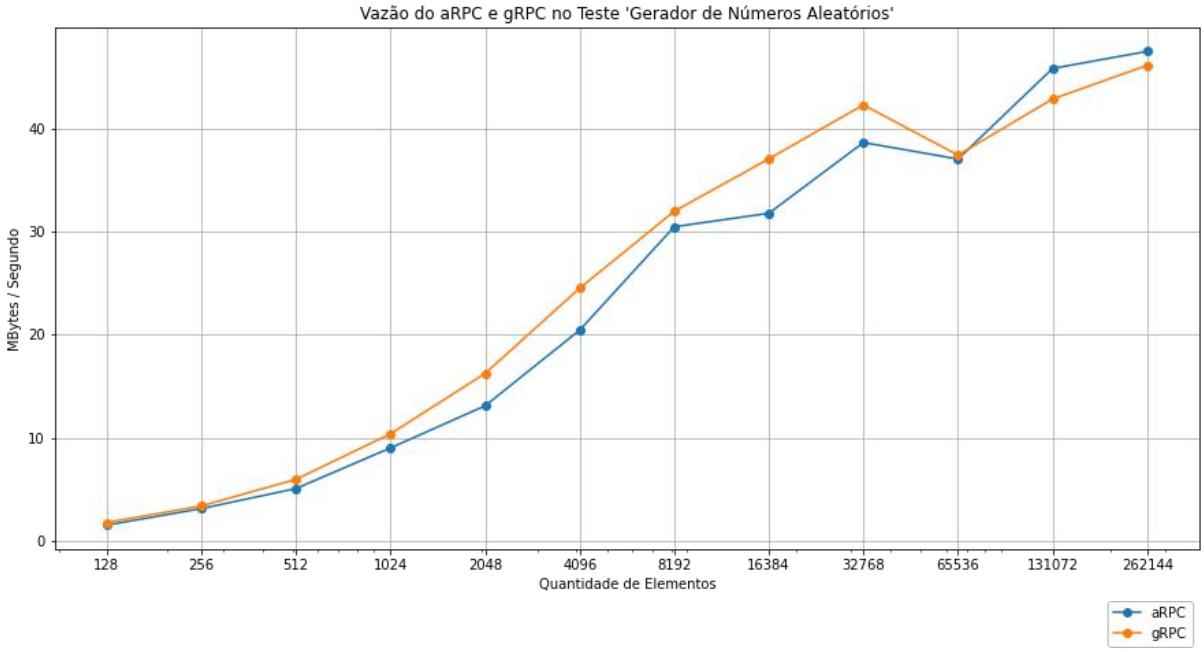
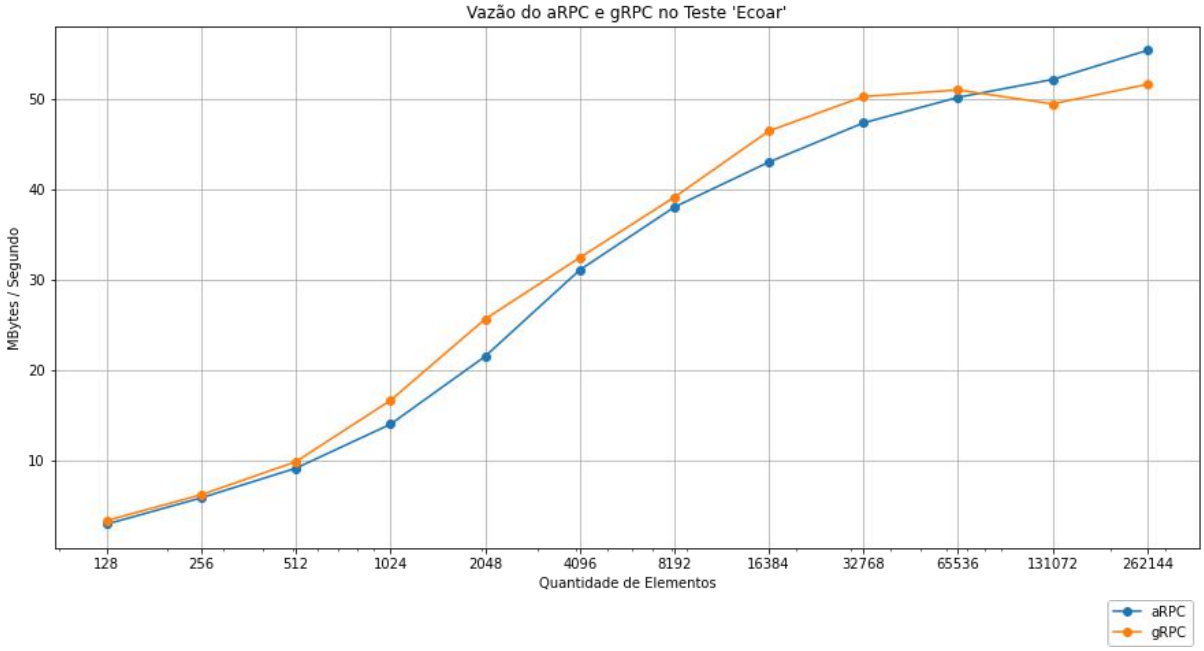


Figura 30 – Vazão do aRPC e do gRPC para o caso de teste **Ecoar**



Pode ser observado que num cenário com maior perda de pacotes, as características do QUIC se destacam mais, tal como a prevenção ao problema de *Head-of-Line Blocking*, tendo em vista que, mesmo para uma quantidade pequena de dados, a vazão do aRPC imediatamente se distancia da vazão do gRPC. Foi utilizado o caso de teste **Ecoar** pois ele utiliza o mesmo dado na requisição e na resposta.

Figura 31 – Resultados do comando **ping** do cliente para o servidor

```
sudo ping -c 1000 -i 0.1 hills-uno.vasconcellos.casa
PING hills-uno.vasconcellos.casa(hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e)) 56 data bytes
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=2 ttl=64 time=0.255 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=3 ttl=64 time=0.246 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=4 ttl=64 time=0.220 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=5 ttl=64 time=0.216 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=6 ttl=64 time=0.207 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=7 ttl=64 time=0.155 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=8 ttl=64 time=0.222 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=9 ttl=64 time=0.242 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=10 ttl=64 time=0.199 ms
[...]
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=990 ttl=64 time=0.180 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=991 ttl=64 time=0.148 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=992 ttl=64 time=0.189 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=993 ttl=64 time=0.229 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=994 ttl=64 time=0.159 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=995 ttl=64 time=0.184 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=996 ttl=64 time=0.142 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=997 ttl=64 time=0.197 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=998 ttl=64 time=0.192 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=999 ttl=64 time=0.207 ms
64 bytes from hills-uno.vasconcellos.casa (fd12:687d:8945:abcd:62a4:4cff:fe61:ac8e): icmp_seq=1000 ttl=64 time=0.192 ms

--- hills-uno.vasconcellos.casa ping statistics ---
1000 packets transmitted, 951 received, 4.9% packet loss, time 263015ms
rtt min/avg/max/mdev = 0.121/0.196/0.513/0.031 ms
```

Figura 32 – Resultados do comando **ping** do servidor para o cliente

```
PING 192.168.1.230 (192.168.1.230) 56(84) bytes of data.
64 bytes from 192.168.1.230: icmp_seq=1 ttl=64 time=0.173 ms
64 bytes from 192.168.1.230: icmp_seq=2 ttl=64 time=0.215 ms
64 bytes from 192.168.1.230: icmp_seq=3 ttl=64 time=0.194 ms
64 bytes from 192.168.1.230: icmp_seq=4 ttl=64 time=0.192 ms
64 bytes from 192.168.1.230: icmp_seq=5 ttl=64 time=0.185 ms
64 bytes from 192.168.1.230: icmp_seq=6 ttl=64 time=0.214 ms
64 bytes from 192.168.1.230: icmp_seq=7 ttl=64 time=0.159 ms
64 bytes from 192.168.1.230: icmp_seq=10 ttl=64 time=0.182 ms
[...]
64 bytes from 192.168.1.230: icmp_seq=990 ttl=64 time=0.164 ms
64 bytes from 192.168.1.230: icmp_seq=991 ttl=64 time=0.199 ms
64 bytes from 192.168.1.230: icmp_seq=993 ttl=64 time=0.167 ms
64 bytes from 192.168.1.230: icmp_seq=994 ttl=64 time=0.189 ms
64 bytes from 192.168.1.230: icmp_seq=995 ttl=64 time=0.242 ms
64 bytes from 192.168.1.230: icmp_seq=996 ttl=64 time=0.159 ms
64 bytes from 192.168.1.230: icmp_seq=997 ttl=64 time=0.149 ms
64 bytes from 192.168.1.230: icmp_seq=999 ttl=64 time=0.218 ms
64 bytes from 192.168.1.230: icmp_seq=1000 ttl=64 time=0.124 ms

--- 192.168.1.230 ping statistics ---
1000 packets transmitted, 944 received, 5.6% packet loss, time 106603ms
rtt min/avg/max/mdev = 0.095/0.186/0.264/0.027 ms
```

4.4.2.2 Microsoft Azure

Neste ambiente, apesar de possuir rede de 10 Gigabits, identificou-se que a transmissão de dados utilizando o QUIC tinha um desempenho comparativamente inferior ao TCP em relação àquele apresentado no ambiente das máquinas pessoais. Por isso, uma investigação foi realizada, utilizando o programa **iperf3**, para verificar o desempenho de transmissão entre as máquinas. O programa foi executado com tempo de teste de 300 segundos e tamanho do pacote UDP padrão, de 1460 bytes. Nos resultados foi averiguado que, enquanto a taxa de transmissão para o protocolos TCP estavam dentro dos valores esperados. A taxa de transmissão do UDP estavam sendo limitada, possivelmente devido às mitigações de segurança descritas na Seção 3.2.2. Os resultados do **iperf3** podem ser conferidos nas Figuras 34 e 35.

Figura 33 – Comparação entre aRPC e gRPC em cenários com e sem perda de pacotes

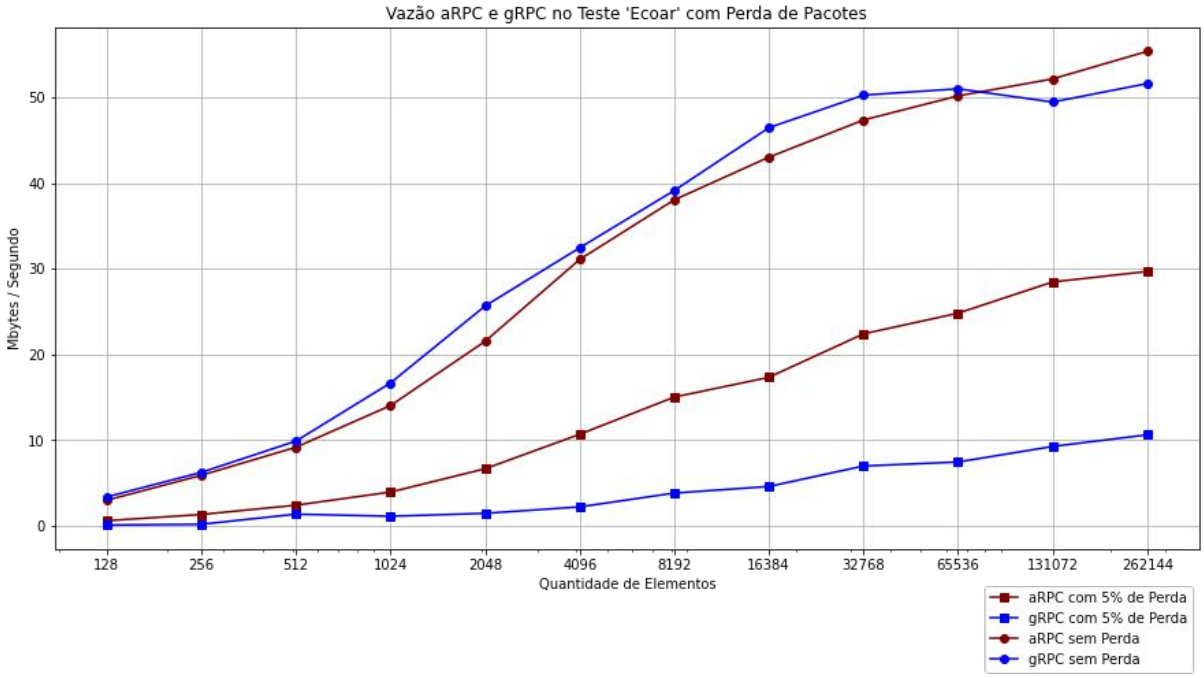


Figura 34 – Resultados do **iperf3** executado no cliente na Microsoft Azure para protocolo UDP

```
tcc@tcc-client:~/tcc$ sudo iperf3 --udp -b 0 -t 300 -c ignite
Connecting to host ignite, port 5201
[  5] local 10.0.0.4 port 34744 connected to 10.0.0.5 port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   504 MBytes  4.23 Gbits/sec  375810
[  5]   1.00-2.00   sec   504 MBytes  4.23 Gbits/sec  376120
[  5]   2.00-3.00   sec   504 MBytes  4.23 Gbits/sec  376020
[  5]   3.00-4.00   sec   504 MBytes  4.23 Gbits/sec  376240

[...]
[  5] 295.00-296.00 sec   505 MBytes  4.24 Gbits/sec  376900
[  5] 296.00-297.00 sec   506 MBytes  4.24 Gbits/sec  377140
[  5] 297.00-298.00 sec   509 MBytes  4.27 Gbits/sec  379510
[  5] 298.00-299.00 sec   526 MBytes  4.41 Gbits/sec  392410
[  5] 299.00-300.00 sec   519 MBytes  4.36 Gbits/sec  387190
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-300.00 sec   148 GBytes  4.24 Gbits/sec  0.000 ms  0/113118230 (0%)  sender
[  5]   0.00-300.00 sec   146 GBytes  4.18 Gbits/sec  0.001 ms  1695421/113118212 (1.5%)  receiver

iperf Done.
```

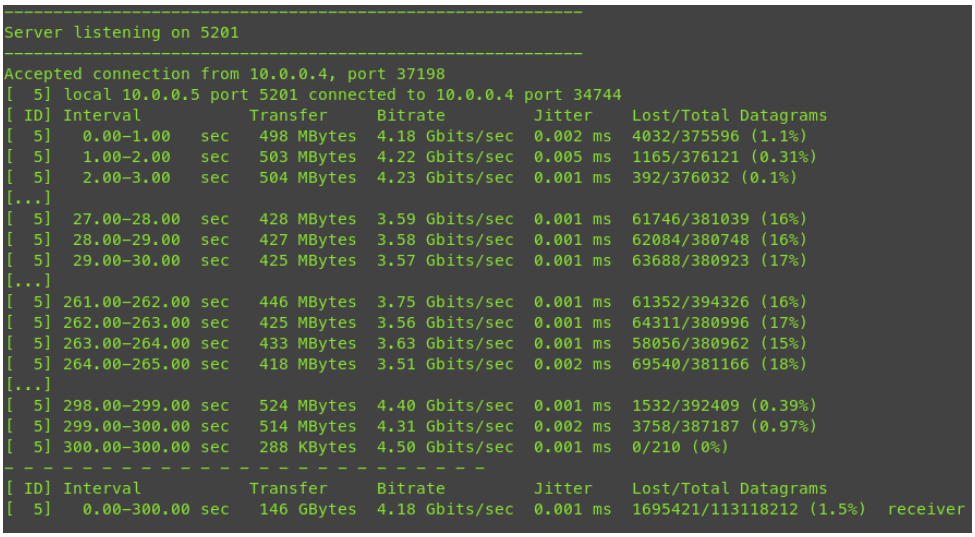
4.4.2.3 Google Cloud Platform

Assim como ocorreu no ambiente Microsoft Azure, foram observadas limitações na taxa de transmissão do protocolo UDP, que afetaram os resultados obtidos pelo aRPC. Portanto a mesma estratégia de análise com o **iperf3** foi empregada, que pode ser conferida nas Figuras 36 e 37, justificando assim o desempenho encontrado.

4.4.2.4 Comparação entre ambientes

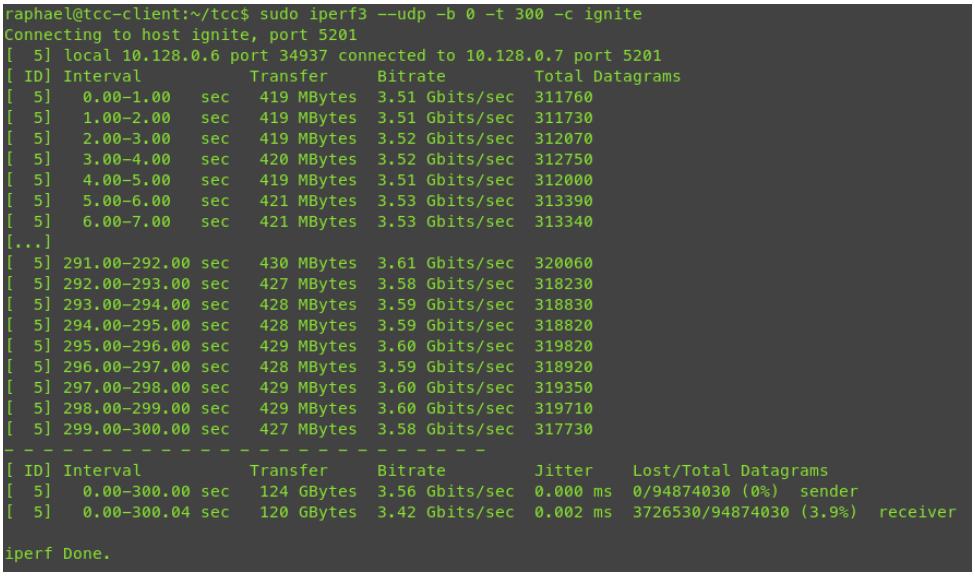
Devido aos estrangulamentos nas transmissões utilizando UDP nos ambientes de nuvem, foi feita uma comparação da evolução de escalabilidade entre os três ambientes,

Figura 35 – Resultados do **iperf3** executado no servidor na Microsoft Azure para protocolo UDP



```
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 10.0.0.4, port 37198
[  5] local 10.0.0.5 port 5201 connected to 10.0.0.4 port 34744
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec   498 MBytes  4.18 Gbits/sec  0.002 ms  4032/375596 (1.1%)
[  5]   1.00-2.00   sec   503 MBytes  4.22 Gbits/sec  0.005 ms  1165/376121 (0.31%)
[  5]   2.00-3.00   sec   504 MBytes  4.23 Gbits/sec  0.001 ms  392/376032 (0.1%)
[...]
[  5]  27.00-28.00  sec   428 MBytes  3.59 Gbits/sec  0.001 ms  61746/381039 (16%)
[  5]  28.00-29.00  sec   427 MBytes  3.58 Gbits/sec  0.001 ms  62084/380748 (16%)
[  5]  29.00-30.00  sec   425 MBytes  3.57 Gbits/sec  0.001 ms  63688/380923 (17%)
[...]
[  5] 261.00-262.00 sec   446 MBytes  3.75 Gbits/sec  0.001 ms  61352/394326 (16%)
[  5] 262.00-263.00 sec   425 MBytes  3.56 Gbits/sec  0.001 ms  64311/380996 (17%)
[  5] 263.00-264.00 sec   433 MBytes  3.63 Gbits/sec  0.001 ms  58056/380962 (15%)
[  5] 264.00-265.00 sec   418 MBytes  3.51 Gbits/sec  0.002 ms  69540/381166 (18%)
[...]
[  5] 298.00-299.00 sec   524 MBytes  4.40 Gbits/sec  0.001 ms  1532/392409 (0.39%)
[  5] 299.00-300.00 sec   514 MBytes  4.31 Gbits/sec  0.002 ms  3758/387187 (0.97%)
[  5] 300.00-300.00 sec   288 KBytes  4.50 Gbits/sec  0.001 ms  0/210 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-300.00 sec   146 GBytes  4.18 Gbits/sec  0.001 ms  1695421/113118212 (1.5%)  receiver
```

Figura 36 – Resultados do **iperf3** executado no cliente na Google Cloud Platform para protocolo UDP



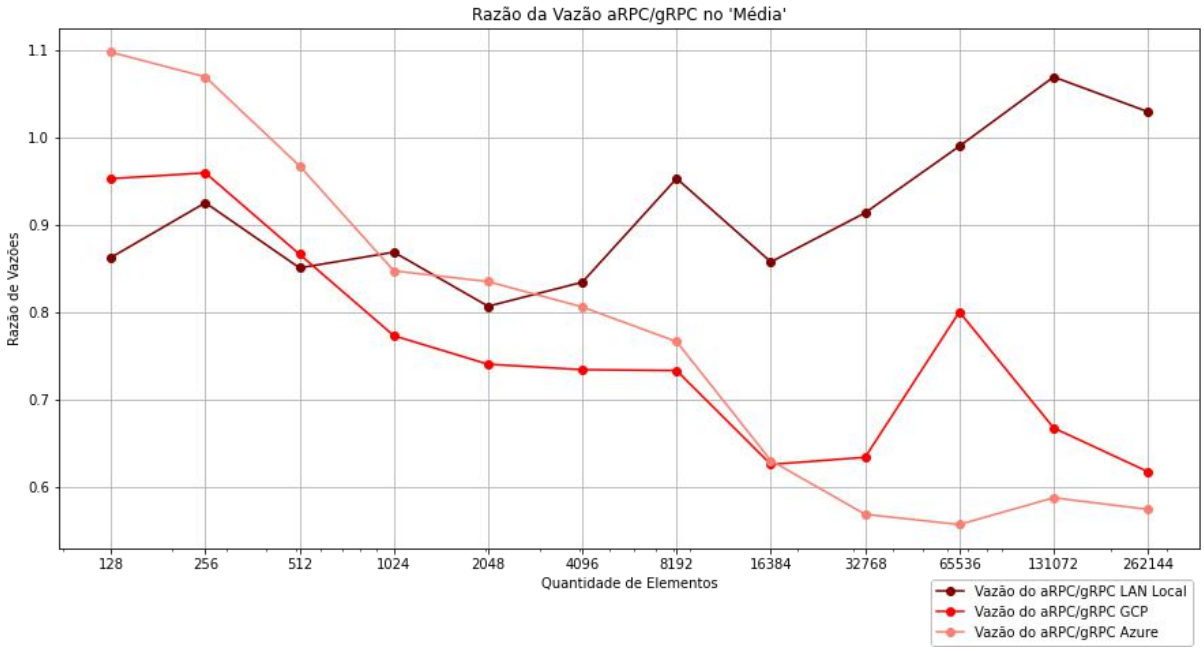
```
raphael@tcc-client:~/tcc$ sudo iperf3 --udp -b 0 -t 300 -c ignite
Connecting to host ignite, port 5201
[  5] local 10.128.0.6 port 34937 connected to 10.128.0.7 port 5201
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   419 MBytes  3.51 Gbits/sec  311760
[  5]   1.00-2.00   sec   419 MBytes  3.51 Gbits/sec  311730
[  5]   2.00-3.00   sec   419 MBytes  3.52 Gbits/sec  312070
[  5]   3.00-4.00   sec   420 MBytes  3.52 Gbits/sec  312750
[  5]   4.00-5.00   sec   419 MBytes  3.51 Gbits/sec  312000
[  5]   5.00-6.00   sec   421 MBytes  3.53 Gbits/sec  313390
[  5]   6.00-7.00   sec   421 MBytes  3.53 Gbits/sec  313340
[...]
[  5] 291.00-292.00 sec   430 MBytes  3.61 Gbits/sec  320060
[  5] 292.00-293.00 sec   427 MBytes  3.58 Gbits/sec  318230
[  5] 293.00-294.00 sec   428 MBytes  3.59 Gbits/sec  318830
[  5] 294.00-295.00 sec   428 MBytes  3.59 Gbits/sec  318820
[  5] 295.00-296.00 sec   429 MBytes  3.60 Gbits/sec  319820
[  5] 296.00-297.00 sec   428 MBytes  3.59 Gbits/sec  318920
[  5] 297.00-298.00 sec   429 MBytes  3.60 Gbits/sec  319350
[  5] 298.00-299.00 sec   429 MBytes  3.60 Gbits/sec  319710
[  5] 299.00-300.00 sec   427 MBytes  3.58 Gbits/sec  317730
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-300.00 sec   124 GBytes  3.56 Gbits/sec  0.000 ms  0/94874030 (0%)  sender
[  5]   0.00-300.04 sec   120 GBytes  3.42 Gbits/sec  0.002 ms  3726530/94874030 (3.9%)  receiver

iperf Done.
```

conforme visto na Figura 38. Nela é possível averiguar que a razão entre as vazões cai conforme a quantidade de elementos aumenta, isso ocorre pois nos ambientes de nuvem existem configurações de sistema limitando a vazão do UDP (Seção 3.2.2). Devido a essa prática, o aRPC não é adequado para esses ambientes, pois apesar de possuírem maior largura de banda, o protocolo acaba tendo seu desempenho limitado. É importante ressaltar que com a crescente adoção do QUIC, essas limitações devem ser removidas ou modificadas (ROSSOW, 2014), como justificado na Seção 3.2.3.

Figura 37 – Resultados do **iperf3** executado no servidor na Google Cloud Platform para protocolo UDP

```
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 10.128.0.6, port 55360
[  5] local 10.128.0.7 port 5201 connected to 10.128.0.6 port 34937
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec   387 MBytes  3.25 Gbits/sec  0.004 ms  10723/298834 (3.6%)
[  5]   1.00-2.00   sec   404 MBytes  3.39 Gbits/sec  0.004 ms  10759/311826 (3.5%)
[  5]   2.00-3.00   sec   405 MBytes  3.39 Gbits/sec  0.006 ms  10565/311927 (3.4%)
[  5]   3.00-4.00   sec   406 MBytes  3.41 Gbits/sec  0.014 ms  10388/312796 (3.3%)
[  5]   4.00-5.00   sec   407 MBytes  3.41 Gbits/sec  0.002 ms  8823/311986 (2.8%)
[...]
[  5]  99.00-100.00 sec   409 MBytes  3.43 Gbits/sec  0.006 ms  10708/315246 (3.4%)
[  5] 100.00-101.00 sec   406 MBytes  3.40 Gbits/sec  0.009 ms  9330/311400 (3%)
[  5] 101.00-102.00 sec   403 MBytes  3.38 Gbits/sec  0.004 ms  14207/314645 (4.5%)
[  5] 102.00-103.00 sec   409 MBytes  3.43 Gbits/sec  0.007 ms  11287/315657 (3.6%)
[...]
[  5] 165.00-166.00 sec   408 MBytes  3.43 Gbits/sec  0.003 ms  12870/317078 (4.1%)
[  5] 166.00-167.00 sec   407 MBytes  3.42 Gbits/sec  0.003 ms  14884/318219 (4.7%)
[  5] 167.00-168.00 sec   411 MBytes  3.44 Gbits/sec  0.006 ms  13971/319763 (4.4%)
[  5] 168.00-169.00 sec   406 MBytes  3.40 Gbits/sec  0.003 ms  16693/318705 (5.2%)
[...]
[  5] 298.00-299.00 sec   414 MBytes  3.48 Gbits/sec  0.002 ms  10969/319567 (3.4%)
[  5] 299.00-300.00 sec   412 MBytes  3.46 Gbits/sec  0.006 ms  11028/317817 (3.5%)
[  5] 300.00-300.04 sec  17.1 MBytes  3.41 Gbits/sec  0.002 ms  368/13069 (2.8%)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-300.04 sec   120 GBytes  3.42 Gbits/sec  0.002 ms  3726530/94874030 (3.9%)  receiver
```

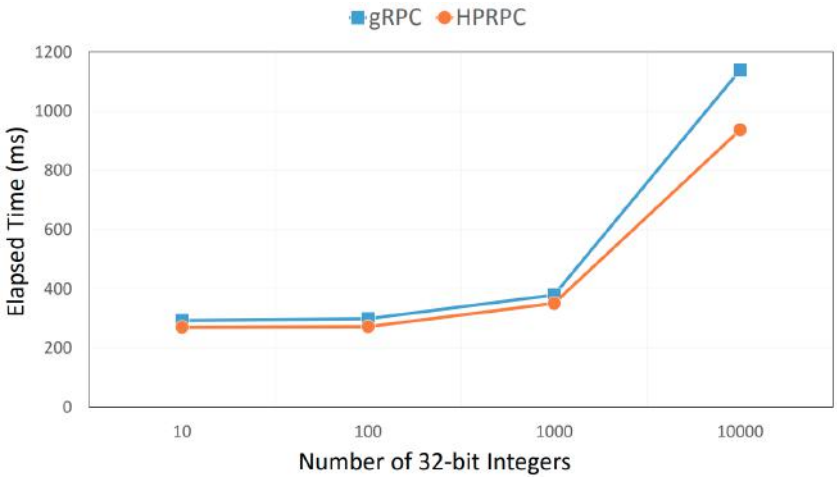
Figura 38 – Razão entre as vazões do aRPC e do gRPC, entre os três ambientes



4.4.3 *Framework*

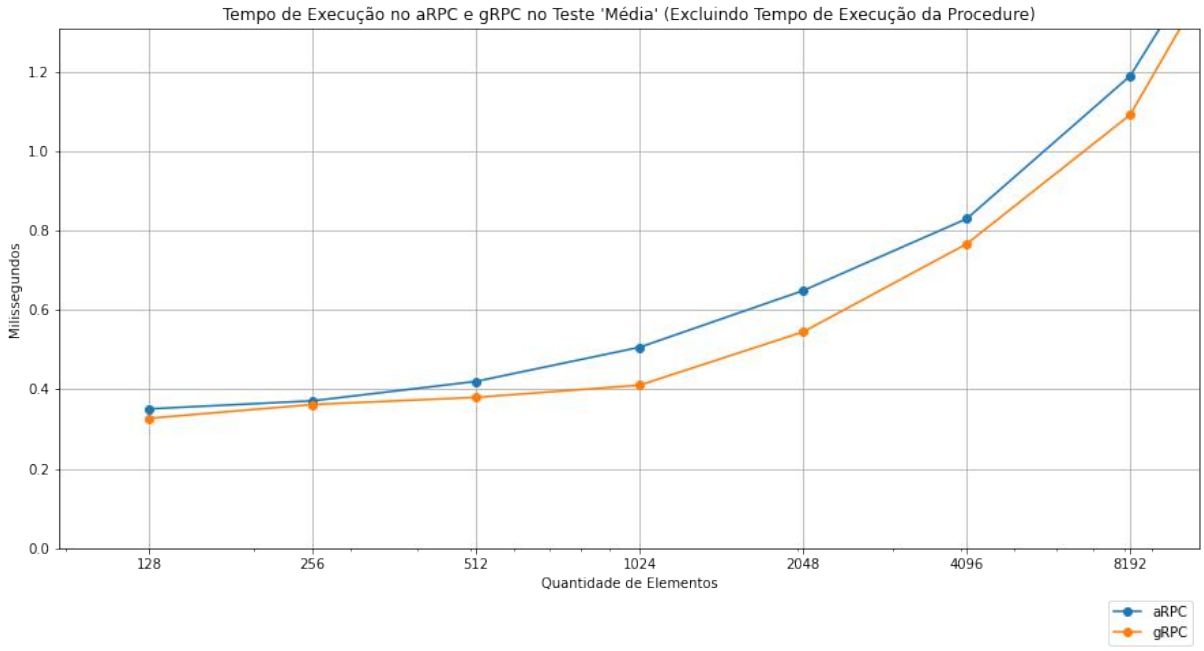
A proposta é que o aRPC seja um *framework* simples para HPC, mantendo as características da linguagem e efetuando chamadas de procedimentos remotos de forma transparente. Neste sentido, é importante que o *framework* tenha desempenho melhor que o gRPC e HPRPC para grandes quantidades de elementos e em contextos de dados mais heterogêneos.

Figura 39 – Tempo de execução do HPRPC e do gRPC



Fonte: (BAGCI; KARA, 2016)

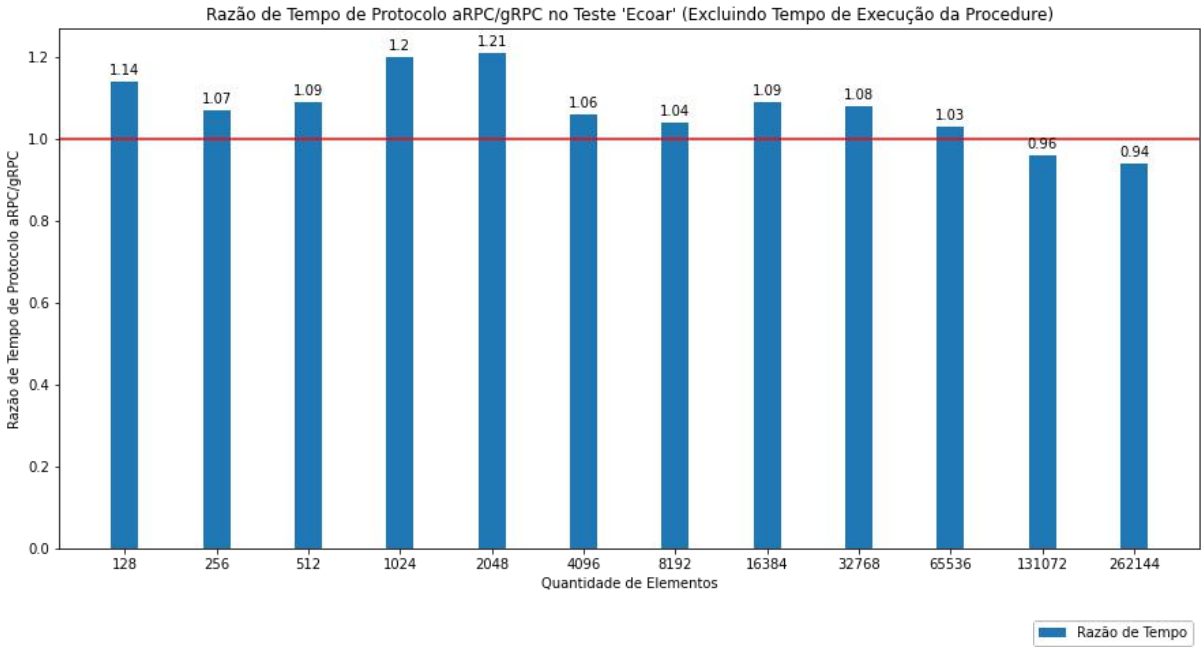
Figura 40 – Tempo de execução do aRPC e do gRPC (Foco em $x \in [0; 1000]$)



No artigo do HPRPC, o autor disponibiliza gráficos de tempo de execução são utilizados para comparar o desempenho de HPRPC e gRPC (BAGCI; KARA, 2016) como disposto na Figura 39. Neste trabalho a mesma abordagem comparativa foi adotada, como mostrado na Figura 40. Na medição realizada nesse trabalho foi observada uma divergência em relação à unidade dos dados apresentados pelos autores no eixo Y. No artigo não é informada a largura de banda da rede e nem o *hardware* utilizado, entretanto, para que 128 elementos **int32** e sua resposta **float64** sejam transferidos são necessários em torno de 520 bytes.

Na Figura 39 o tempo de execução foi cerca de 300 ms tanto para HPRPC e gRPC. Entretanto, esses valores apontam para uma vazão de 1,3 KB/s e esse é um valor suspeito, uma vez que seria necessário um *hardware* muito antigo para que o gRPC tenha esse tempo de execução. Como pode ser visto na Figura 40 o gRPC tem um tempo de transferência por volta de 0,3 ms para essa ordem de grandeza de dados. Isso levanta possíveis indícios de que as medições estão em microssegundos e não em milissegundos como indicado pelos autores. Sendo assim, não foi possível realizar os testes comparativos planejados com o HPRPC.

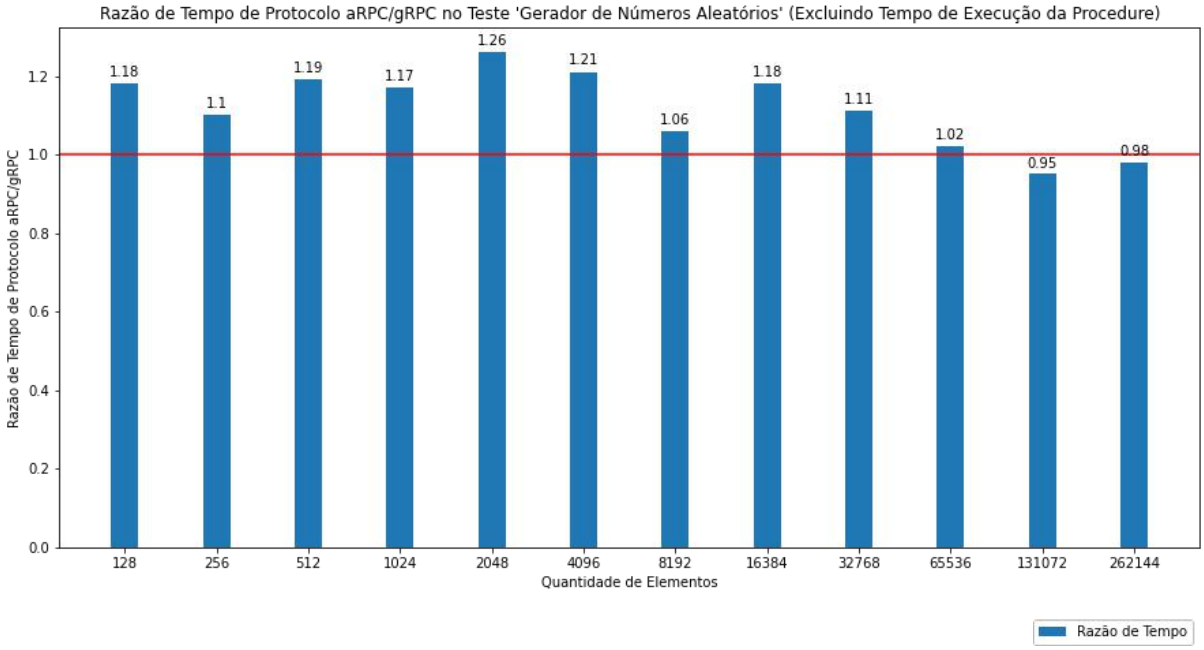
Figura 41 – Razão do Tempo de execução do aRPC e do gRPC para o caso de teste **Ecoar**



Como pode ser visto nas Figuras 41, 42 e 43, referentes respectivamente aos testes **Ecoar**, **Gerador de Números Aleatórios** e **Média**, o aRPC possui um desempenho pior que o gRPC para uma quantidade de elementos inferior a 65536. A partir desse valor o aRPC se iguala e então começa a ficar mais rápido que o gRPC com ganhos de 2% a 7%. A partir de uma quantidade grande suficiente de números, os ganhos de *speedup* do serializador, somado às reduções de sobrecarga proporcionadas pelo QUIC, fazem com que o *framework* escale melhor que o gRPC. Os ganhos mencionados tendem a aumentar conforme a quantidade de elementos aumenta até que a banda da rede esteja inteiramente saturada.

A Figura 44 é referente ao teste **Todos os Tipos** que, como apresentado anteriormente, representa melhor o uso esperado do *framework*. O resultado obtido foi favorável mesmo para uma quantidade pequena de elementos, sendo que o ganho nessas condições advém de uma serialização mais eficiente, em termos de quantidade de bytes, realizada

Figura 42 – Razão do Tempo de execução do aRPC e do gRPC para o caso de teste **Gerador de NúmerosAleatórios**



pelo serializador Colfer em comparação com o serializador Protobuffers, conforme exemplificado na Figura 25.

A partir da análise dos dados coletados acima é possível concluir que, caso a serialização do Colfer seja mais eficiente ou igual ao Protobuffers, o aRPC pode ter um desempenho melhor ou igual ao gRPC.

Já para grandes quantidades de elementos as vantagens do aRPC são mais evidentes e, nesses casos, mesmo em uma situação de serialização desvantajosa, o aRPC é mais eficiente, atingindo maior vazão.

Figura 43 – Razão do Tempo de execução do aRPC e do gRPC para o caso de teste **Média**

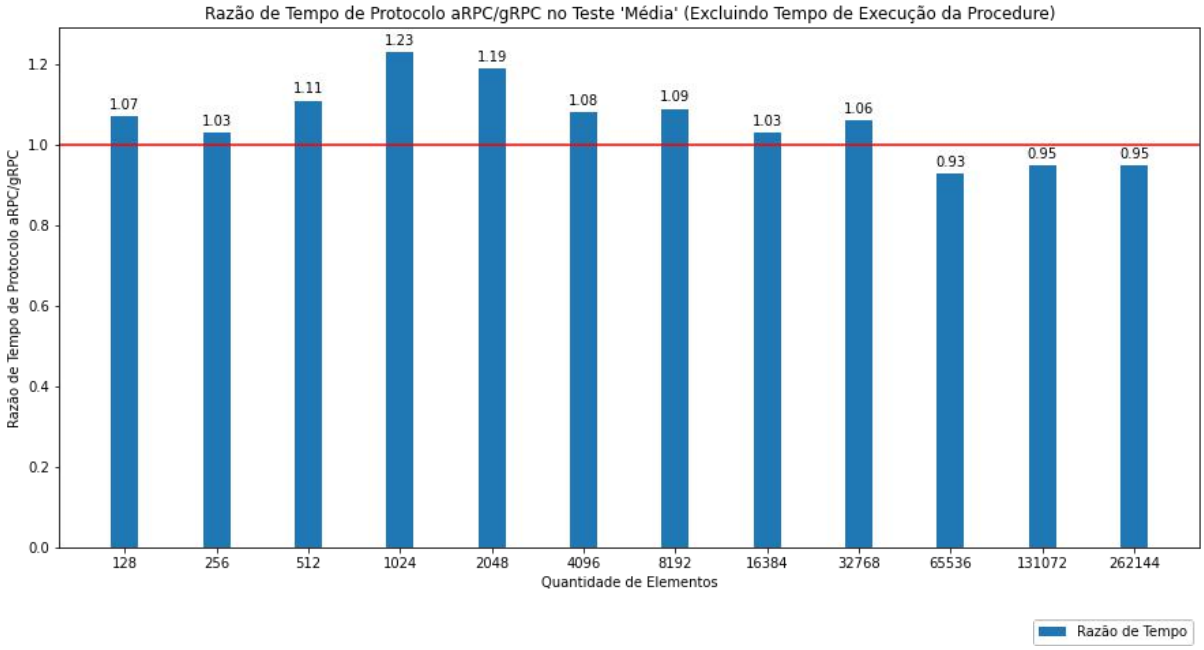
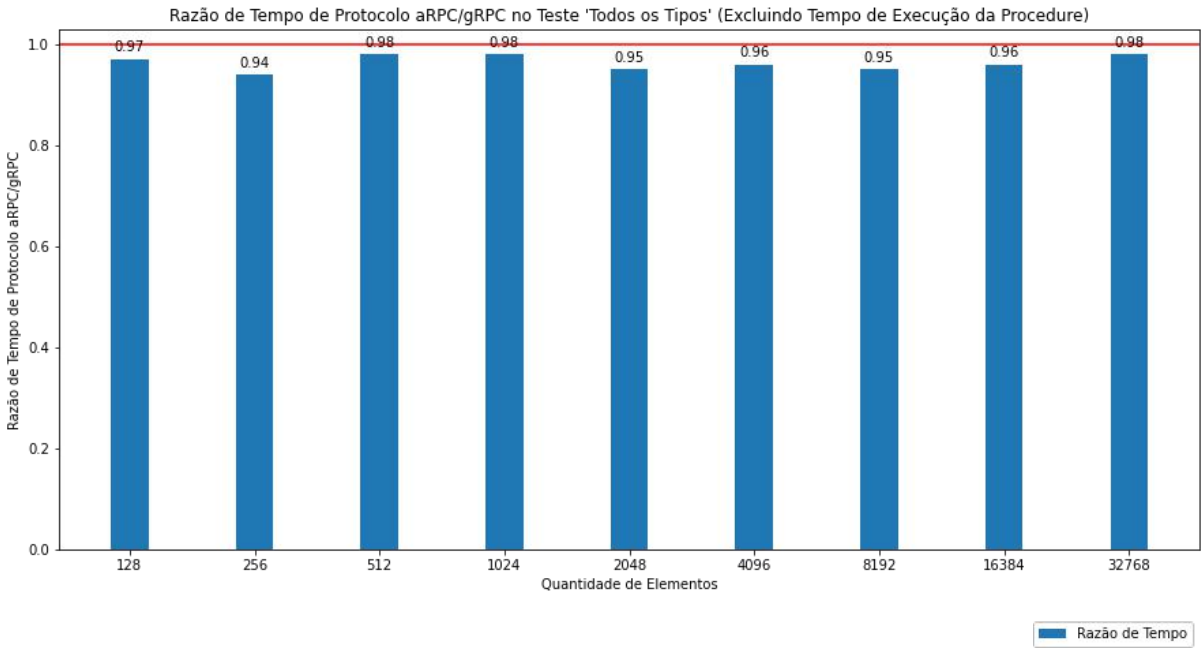


Figura 44 – Razão do Tempo de execução do aRPC e do gRPC para o caso de teste **Todos os Tipos**



5 CONCLUSÃO

Neste trabalho foi apresentado o aRPC, um *framework* que procura oferecer ergonomia na escrita do código para o desenvolvedor e um bom desempenho de execução, priorizando ambientes de computação de alto desempenho (HPC). Foram também apresentados os seus resultados quando comparado com o gRPC. O projeto do *framework* foi disponibilizado em https://github.com/almeida-raphael/arpc, o compilador para a IDL proposta é encontrado em https://github.com/almeida-raphael/arpc_code_generator e por fim, os casos de teste com as lógicas de coleta de métrica e exemplos de implementação podem ser encontrados em https://github.com/almeida-raphael/arpc_examples.

A proposta do aRPC tem simplicidade como princípio, então, fazendo uso das capacidades presentes no QUIC para a camada de transporte e no Colfer para a serialização. A implementação do aRPC ficou muito enxuta, contando com cerca de 1500 linhas de código, incluindo o *framework* e o compilador. O projeto foi desenvolvido de maneira colaborativa pelos integrantes deste trabalho. Com base nos resultados positivos obtidos na comparação com o gRPC e na quantidade total de linhas de código produzida, é possível afirmar que a arquitetura proposta atingiu o princípio de simplicidade desejado.

A obtenção dos resultados necessitou de uma extensa bateria de testes. Entretanto, devido ao longo tempo de execução necessário para avaliar todos os aspectos desejados, e considerando a grande quantidade de métricas requeridas para a validade estatística dos resultados, foi desenvolvido um mecanismo de automação dos testes. Esse mecanismo de testes abstraiu as lógicas de coleta de métricas de maneira genérica entre os exemplos utilizando o aRPC e os utilizando o gRPC. Os testes envolvidos totalizaram por volta de 4900 linhas de código. Cabe levantar que ainda há espaço para explorar o aRPC em mais cenários, como para aplicações de IoT, dada a boa resiliência apresentada pelo QUIC em ambientes com elevadas perda de pacotes. Tal aplicação do aRPC para cenários de IoT pode ser considerada para um trabalho futuro.

Quanto à implementação do *framework*, algumas dificuldades foram encontradas. Pode-se citar como exemplo o problema ocasionado pela estratégia de serialização com tamanho variável implementada pelo Colfer que impossibilita a separação correta do cabeçalho e dados na *stream* de comunicação. Tornando-se necessário a introdução de um byte antes do cabeçalho, que discrimina seu tamanho, possibilitando sua leitura completa antes de começar a processar os dados da chamada ou resultado do procedimento.

Nas seções a seguir são descritos os principais aspectos levantados para os blocos lógicos que compõem o aRPC, que são o serializador, o transporte e o próprio *framework* de maneira geral.

5.0.1 Serializador

Simplicidade ímpar e extrema facilidade de uso, somados a um ótimo desempenho de serialização, demonstraram que utilizar o Colfer como o serializador do aRPC foi uma escolha acertada. O Colfer pode gerar ganhos em termos de *speedup* de 1,47 a 2,87 e, apesar dos dados serializados serem ligeiramente maiores do que os do Protobuffers (de 1,1% a 1,6% maiores), isso não chegou a ser prejudicial para o desempenho do *framework* em geral, tendo em vista que em diversos casos o aRPC obteve desempenho superior ao gRPC, principalmente se tratando de dados heterogêneos ou de estruturas com muitos elementos, conforme visto na Seção 4.4.2.1.

Um estudo posterior pode tentar combinar os benefícios do Colfer com os benefícios do Protobuffers, de maneira a ter um maior desempenho de codificação com dados serializados em tamanhos menores. Um exemplo de melhoria para o Protobuffers seria na serialização de números inteiros negativos, onde desnecessariamente são usados 11 bytes como indicado no Capitulo 4.

5.0.2 Transporte

Utilizar o QUIC como protocolo da camada de transporte trouxe diversos benefícios para o *framework*, desde a facilidade de implementação, até a resiliência para transmissão de dados em cenários com perda de pacotes na rede.

Foi também possível constatar que mesmo transmitindo dados ligeiramente maiores, na faixa de 1,1% a 1,6%, por conta da serialização do Colfer, a vazão na transmissão dos dados ainda conseguiu ser superior à do gRPC em diversos casos, implicando em menor tempo total de execução, o que indica que o desempenho do QUIC (no aRPC) é maior que o do TCP (no gRPC).

Infelizmente, os bons resultados encontrados pelo QUIC no ambiente de máquinas pessoais se comunicando numa LAN fechada não puderam ser replicados num ambiente de nuvem, possivelmente devido a restrições na taxa de transferência de datagramas UDP, impostos pelos provedores desses serviços. Entretanto, esse cenário tem o potencial de melhorar conforme a adoção do QUIC ganha impulso e o protocolo se torna mais conhecido.

Tendo em vista o cenário encontrado ao se utilizar aplicações com o aRPC em ambientes de nuvem, é possível se considerar um trabalho futuro realizando novos testes em *clusters* privados que possuam redes com largura de banda elevada, porém sem as restrições impostas para comunicação via datagramas UDP. Isso irá permitir entender melhor o desempenho do *framework* em um contexto de redes mais velozes.

Outro possível trabalho futuro, relacionado com a camada de transporte, é a implementação do protocolo TCP para o aRPC. Isso forneceria mais informações para comparação

com o gRPC, uma vez que ambos os *frameworks* estariam utilizando o mesmo protocolo de transporte.

5.0.3 Framework

O aRPC foi comparado principalmente ao gRPC, neste contexto faz sentido explicitar as principais diferenças entre os dois *frameworks*. O gRPC é um *framework* maduro e completo, com foco em oferecer uma solução de uso geral com bom desempenho. Dessa forma, ele suporta muitas funcionalidades, tais como: argumentos opcionais, versionamento dos serviços, suporte a resposta de dados em *stream*, balanceamento de carga, etc. O aRPC é um *framework* direcionado a HPC com princípio na simplicidade e ergonomia, possuindo um conjunto de funcionalidades reduzidas, não suportando nenhuma das capacidades mencionadas anteriormente. Entretanto, com o foco em HPC e uso do QUIC como protocolo base, apresenta maior desempenho para dados grandes, que é característico das cargas de trabalho de HPC, além de ter maior resiliência a instabilidades de conexão.

Em relação às linguagens suportadas, o gRPC tem a vantagem de seu longo tempo de existência e suporte da comunidade de *software* aberto. Nesse sentido, possui vasta compatibilidade, porém, devido a sua proposta de ser um pacote completo, não é tão fácil implementá-lo em outras linguagens quanto o aRPC. Atualmente o aRPC suporta somente o Go, mas, como mencionado anteriormente, a simplicidade deste *framework* proporciona uma enorme vantagem na implementação para novas linguagens que pode eventualmente levá-lo a um suporte mais amplo que o gRPC.

No contexto de dependências, o gRPC é um projeto grande, com muitas linhas de código e junto ao tamanho e complexidade, soma-se o número de dependências. O gRPC depende hoje de doze bibliotecas, algumas do próprio Google, assim como outras externas. Por outro lado, o aRPC possui somente uma dependência, visto que a CLI do Colfer é usada exclusivamente na etapa de geração de código, não sendo uma dependência de tempo de execução do *framework*. Por essa razão, hoje o aRPC depende somente de uma implementação do QUIC disponível como biblioteca.

Como dito anteriormente, o gRPC é um projeto maduro, com muitos anos de desenvolvimento, muitos colaboradores e pronto para uso em ambientes de produção, já o aRPC por ser muito novo e ainda não testado extensivamente, não é recomendado ainda para uso em ambientes de produção.

Na comparação de desempenho entre gRPC e aRPC, existe também a questão do resultado abaixo do esperado do aRPC em ambientes de nuvem devido a restrições na taxa de transferência de datagramas UDP impostas pelos provedores. Esse fator faz com que hoje o gRPC tenha desempenho superior nesse cenário, entretanto, como mencionado, existe uma expectativa que essas restrições sejam aliviadas, ou removidas no futuro com o crescimento do QUIC.

Considerando os resultados apresentados no Capítulo 4, os objetivos iniciais propostos foram alcançados, o aRPC obteve ganhos de até 7% em relação ao gRPC. Como foi demonstrado, o aRPC teve desempenho superior ao gRPC de forma consistente para ambos os casos onde a quantidade de dados trafegada era grande o suficiente, situação comum em aplicações em ambientes HPC, e casos com estruturas de dados heterogêneas que melhor representam dados utilizados em situações reais. Tendo em vista a maturidade do gRPC e sua grande adoção no mercado, esses resultados para um protocolo novo em suas versões iniciais são muito motivantes e demonstram grande espaço para crescimento com evoluções futuras.

O foco deste trabalho se restringiu às premissas básicas do *framework* e sua comparação com uma das referências de mercado no contexto de RPC. Nesse sentido, no contexto de evoluções e trabalhos futuros, fica a recomendação de validação dos resultados e análises do *framework* em redes de alto desempenho com 2,5 Gbps, 10 Gbps ou maiores larguras de banda. Além disso, como na arquitetura do aRPC foi previsto o conceito de **channel** para abstrair outros protocolos, outra análise interessante seria o teste do aRPC usando TCP, HTTP/2 e outros protocolos de transporte. Os experimentos com outros protocolos gerariam dados interessantes a respeito de quanto do desempenho é obtido por cada bloco lógico do aRPC. Além disso, outro experimento relevante seria substituir o Colfer como serializador, possibilitando dessa forma validar a contribuição individual de serializadores distintos para o resultado final. Por fim, uma configuração relevante para análise, seria utilizar os diversos tipos de otimizações do Protobuffers, dentre eles: redução na utilização de recursos; geração de dados pequenos; entre outros. Tendo o propósito de gerar resultados comparativos relativos a todos os modos de otimização oferecidos pelo gRPC e aRPC, implicando em uma análise mais completa.

O uso das tecnologias escolhidas, QUIC e Colfer, mostram que avanços na área de protocolos e bibliotecas para uso em RPC são possíveis. Estudos e implementações de novas técnicas de serialização no contexto de RPC são bem vindos e podem gerar bons resultados, como o exemplo do aRPC evidencia. Dessa forma, esse trabalho termina apontado que mais desenvolvimentos na área são bem vindos, e que a expectativa é que o aRPC fique como estudo de caso para impulsionar mais pesquisas nesse ramo.

REFERÊNCIAS

BAGCI, H.; KARA, A. A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication:. In: **Proceedings of the 11th International Joint Conference on Software Technologies**. Lisbon, Portugal: SCITEPRESS - Science and Technology Publications, 2016. p. 117–124. ISBN 978-989-758-194-6. Disponível em: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005931201170124.

BELSHE, M.; PEON, R.; THOMSON, M. **Hypertext Transfer Protocol Version 2 (HTTP/2)**. RFC Editor, 2015. RFC 7540. (Request for Comments, 7540). Disponível em: https://rfc-editor.org/rfc/rfc7540.txt.

BERGSTROM, E.; PANDEY, R. Anycast-rpc for wireless sensor networks. In: IEEE. **2007 IEEE International Conference on Mobile Adhoc and Sensor Systems**. [S.l.], 2007. p. 1–8.

BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 2, n. 1, p. 39–59, 1984.

BISHOP, M. **Hypertext Transfer Protocol Version 3 (HTTP/3)**. [S.l.], 2021. Work in Progress. Disponível em: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34.

BLANTON, E.; PAXSON, D. V.; ALLMAN, M. **TCP Congestion Control**. RFC Editor, 2009. RFC 5681. (Request for Comments, 5681). Disponível em: https://rfc-editor.org/rfc/rfc5681.txt.

CARDWELL, N. **networking:netem [Wiki]**. 2021. (Acessado em: 26/05/2021). Disponível em: https://wiki.linuxfoundation.org/networking/netem.

CERF, V.; KAHN, R. A Protocol for Packet Network Intercommunication. **IEEE Transactions on Communications**, v. 22, n. 5, p. 637–648, maio 1974. ISSN 1558-0857. Conference Name: IEEE Transactions on Communications.

CLOCK_GETTIME(2) - Linux manual page. Linux Man Pages, 2021. (Acessado em: 26/05/2021). Disponível em: https://man7.org/linux/man-pages/man2/clock_gettime.2.html.

COOK, S. et al. QUIC: Better for what and for whom? In: **2017 IEEE International Conference on Communications (ICC)**. [S.l.: s.n.], 2017. p. 1–6. ISSN: 1938-1883.

FATEMIAN, M. **Why load balancing gRPC is tricky?** 2020. Disponível em: https://majidfn.com/blog/20201222-grpc-load-balancing/.

FORD, B.; HIBLER, M.; LEPREAU, J. Using annotated interface definitions to optimize RPC. In: **Proceedings of the fifteenth ACM symposium on Operating systems principles**. New York, NY, USA: Association for Computing Machinery, 1995. (SOSP '95), p. 232. ISBN 978-0-89791-715-5. Disponível em: https://doi.org/10.1145/224056.225833.

GOLANG/GO. Go, 2021. (Acessado em: 26/05/2021). Disponível em: https://github.com/golang/go/blob/03761ede028d811dd7d7cf8a2690d4bfa2771d85/src/runtime/time_linux_amd64.s.

GOOGLE. **Protobuffers**. 2008. (Acessado em: 26/05/2021). Disponível em: https://developers.google.com/protocol-buffers.

gRPC. 2015. (Acessado em: 26/05/2021). Disponível em: https://grpc.io/.

IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2019 (Revision of IEEE 754-2008)**, p. 1–84, 2019.

LAKSHMAN, T.; MADHOW, U. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. **IEEE/ACM transactions on networking**, IEEE, v. 5, n. 3, p. 336–350, 1997.

LANGLEY, A. et al. The quic transport protocol: Design and internet-scale deployment. In: **Proceedings of the conference of the ACM special interest group on data communication**. [s.n.], 2017. p. 183–196. Disponível em: https://dl.acm.org/doi/abs/10.1145/3098822.3098842.

Addressing the complexity of HPC in the cloud: Emergence, self-organisation, self-management, and the separation of concerns. In: LYNN, T. (Ed.). **Heterogeneity, High Performance Computing, Self-Organization and the Cloud**. Cham: Palgrave Macmillan, Cham, 2018. p. 1–30. ISBN 978-3-319-76037-7 978-3-319-76038-4. Disponível em: http://link.springer.com/10.1007/978-3-319-76038-4.

NELSON, B. J. **Remote procedure call**. [S.l.]: Carnegie Mellon University, 1981.

NEUMAN, B. C. ord. Scale in Distributed Systems. **ISI/USC**, p. 68, 1994.

NEWMARCH, J. Data Serialization. In: NEWMARCH, J. (Ed.). **Network Programming with Go: Essential Skills for Using and Securing Networks**. Berkeley, CA: Apress, 2017. p. 57–86. ISBN 978-1-4842-2692-6. Disponível em: https://doi.org/10.1007/978-1-4842-2692-6_4.

POSTEL, J. **User Datagram Protocol**. RFC Editor, 1980. (Request for Comments, 768). Published: RFC 768. Disponível em: https://rfc-editor.org/rfc/rfc768.txt.

RESCORLA, E.; DIERKS, T. **The Transport Layer Security (TLS) Protocol Version 1.2**. RFC Editor, 2008. RFC 5246. (Request for Comments, 5246). Disponível em: https://rfc-editor.org/rfc/rfc5246.txt.

ROSSOW, C. Amplification hell: Revisiting network protocols for ddos abuse. In: **NDSS**. [S.l.: s.n.], 2014.

SAXCé, H. de; OPRESCU, I.; CHEN, Y. Is HTTP/2 really faster than HTTP/1.1? In: **2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)**. [S.l.: s.n.], 2015. p. 293–299.

SCHARF, M.; KIESEL, S. Nxg03-5: Head-of-line blocking in tcp and sctp: Analysis and measurements. In: IEEE. **IEEE Globecom 2006**. [S.l.], 2006. p. 1–5.

SLEE, M.; AGARWAL, A.; KWIATKOWSKI, M. Thrift: Scalable Cross-Language Services Implementation. p. 8, 2007.

SOUMAGNE, J.; CARNS, P. H.; ROSS, R. B. Advancing rpc for data services at exascale. **IEEE Data Eng. Bull.**, v. 43, n. 1, p. 23–34, 2020.

TIME - The Go Programming Language. 2021. (Acessado em: 26/05/2021). Disponível em: https://golang.org/pkg/time/#Time.

UNICODE 6.0.0. 2010. (Acessado em: 26/05/2021). Disponível em: https://www.unicode.org/versions/Unicode6.0.0/.

WANG, J. et al. An experimental study of bitmap compression vs. inverted list compression. In: **Proceedings of the 2017 ACM International Conference on Management of Data**. [S.l.: s.n.], 2017. p. 993–1008.

ZAYTSEV, V.; BAGGE, A. H. Parsing in a Broad Sense. In: DINGEL, J. et al. (Ed.). **Model-Driven Engineering Languages and Systems**. Cham: Springer International Publishing, 2014. (Lecture Notes in Computer Science), p. 50–67. ISBN 978-3-319-11653-2.