Remote Procedure Call: análise comparativa entre diferentes implementações do sistema de chamadas de procedimento remoto

Alexandre Farias Santos
Universidade de São Paulo
alexandre.farias.santos@usp.br

Anderson Kistner
Universidade de São Paulo
anderson.kistner@usp.br

I. Implementações e bibliotecas utilizadas

Os mecanismos de *remote procedure call* (RPC) utilizados para este experimento foram **gRPC** e **RPyC**, sabe-se que o primeiro mecanismo tem um uso mais genérico em termos de linguagens de programação, pois pode ser utilizado em uma grande gama de linguagens[1], ao contrário do segundo que se limita a implementações feitas em python. Todavia, deseja-se testar o desempenho de ambas as implementações de RPC em um sistema do tipo cliente-servidor programado na linguagem de programação python.

A. gRPC

Atualmente mantido pela Cloud Native Computing Foundation, segundo a própria documentação, nesta implementação, uma aplicação cliente pode chamar vários métodos diretamente de uma aplicação servidor, tendo assim por objetivo, facilitar a criação de sistemas e serviços distribuídos.

O gRPC tem por base a ideia de definir um serviço através de um protocolo onde as mensagens de envio, retorno e os métodos são especificados.

Com este protocolo devidamente descrito, é criada uma implementação no servidor que atende o protocolo, e que quando executada, pode gerenciar as chamadas de aplicações clientes.

No lado do cliente, também existe uma implementação que fornece uma interface para a chamada de procedimentos remotos, também conhecida como *stub*, assim como ilustra a figura 1.

Além disso, toda aplicação cliente-servidor gRPC pode se comunicar em uma grande variedade de ambientes, desde pequenas redes locais, até mesmo redes externas como serviços de computação em nuvem.

Outra grande vantagem deste mecanismo está na variedade de linguagens de programação suportadas, citando por exemplo Java, Go, Python, Ruby, C++, PHP e outras.

A comunicação do gRPC é baseada em buffers de protocolo, uma definição de protocolos do Google *OpenSource* para definição dos dados estruturados que serão serializados. Em geral, o primeiro passo na descrição destes buffers de protocolo

[1]A documentação com a lista completa das linguagens pode ser acessada através desta URL: https://grpc.io/docs/

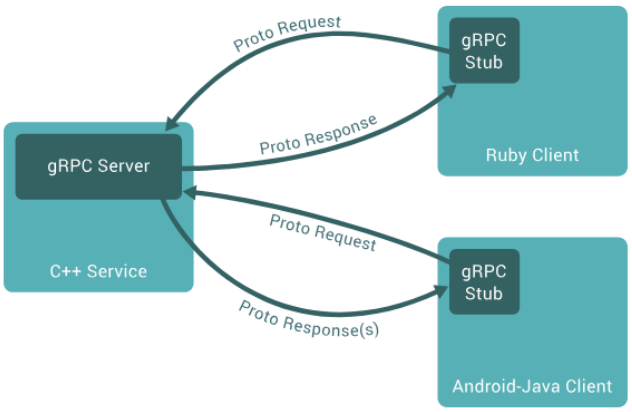

Figura 1. Mecanismo geral de funcionamento do gRPC. Fonte: https://grpc.io/docs/guides/index.html

é a definição das estruturas de dados que serão serializadas pelas aplicações, sejam tipos primitivos, *arrays* ou mensagems mais complexas como objetos. Tendo definido as estruturas de dados, descreve-se então a assinatura dos métodos junto ao seu tipo de retorno, para a implementação do servidor. Finalmente utiliza-se um compilador de buffer de protocolo para gerar as classes que gerenciam as chamadas dos clientes, e fornecem as interfaces necessárias para que as aplicações clientes usem os métodos através de chamadas remotas.

A implementação gRPC, fornece mecanismos como prazos e cancelamento de chamadas de modo que nenhum trabalho adicional seja feito para o procedimento remoto que está sendo cancelado, além de mecanismos de autenticação integrados.

Para este experimento, utilizou-se a versão de buffer de protocolo **proto3**, e a biblioteca **grpcio 1.13.0** para implementação do grpc em python.

B. RPyC

De acordo com a documentação da implementação, o grande foco do RPyC é a transparência, elimando a necessidade de escrever arquivos de definição de protocolos.

Para esta implementação, é levado em consideração o conceito de simetria, logo ambas as partes de uma aplicação cliente-servidor, na verdade, podem atender solicitações e

enviar respostas. Logo, clientes e servidores são praticamente idênticos.

Um conceito importante para a implementação do RPyC é o *boxing*, responsável por transferir objetos entre as duas pontas de uma conexão. No geral, o boxing permite duas maneiras de serialização, por **valor** e por **referência**.

Na serialização por valor, objetos simples do python como strings, inteiros e tuplas são passados por valor, de modo que uma cópia dos dados é enviada. Para outros tipos de objetos, a passagem é feita via refência, de modo que alterações feitas em uma ponta são refletidas na outra.

Portanto, em passagem de parâmetros por valor, o outro lado da conexão deserializa e converte em objetos locais, enquanto por referência, os dados são convertidos em *object proxies*.

A técnica de *object proxying*, permite transparência ao acessar um objeto remoto, na prática, tem-se um objeto especial chamado de *proxy*, este objeto tem a mesma aparência em termos de atributos e comportamento que o objeto passado por referência, de modo que qualquer operação feita no proxy, seja espelhada de maneira transparente no objeto passado por referência, independente do objeto ser local ou remoto.

Esta implementação também oferece mecanismos de segurança, sendo neste caso um módulo *secure socket layer* (SSL)[2].

Além disso, também oferece mecanismos de *timeouts* programados, assim como o gRPC.

Para este experimento, foi utilizada a biblioteca **RPyC Versão 4.0.1**, lançada em 12/06/2018.

C. Outras bibliotecas utilizadas

Além das bibliotecas principais dos mecanismos de RPC, biblotecas auxiliares para a realização dos testes também foram utilizadas, sendo estas:

1) **numpy 1.14.4**: para testes com operações em vetores.
2) **pandas 0.23.0**: para criação e manipulação do conjunto de dados para análise de desempenho.
3) **matplotlib 2.2.2**: para visualização do agregado dos dados.
4) **time** (default python 3.6): para monitorar o tempo de exeucução de cada teste.

Se o leitor estiver interessado em configurar o ambiente para testes, está disponibilizado junto deste relatório um ambiente virtual em python 3.6 cuja configuração das bibliotecas já está devidamente feita. Para sistemas operacionais Linux o *virtualenv* é nomeado como rpc.zip, já para sistemas operacionais Windows, o ambiente é nomeado como rpc_win.zip, para sistemas Windows é altamente recomendável a utilização do Power Shell para ativar o ambiente corretamente.

II. Especificações do ambiente de testes

A. Testes definidos para análise das implementações

Foram definidos 12 testes para chamadas de procedimentos remotos que são:

1) Função sem argumento com retorno vazio (void).
2) Função com argumento booleano e retorno booleano.
3) Função com argumento inteiro e retorno inteiro.
4) Função com argumento long e retorno long.
5) Função com argumento string e retorno string (32 caracteres).
6) Função com argumento string e retorno string (512 caracteres).
7) Função com argumento string e retorno string (1024 caracteres).
8) Função com array de inteiros e retorno array de inteiros (array de 200 posições).
9) Função com array de floats e retorno array de floats (array de 200 posições).
10) Função com array de strings e retorno array de strings (array de 200 posições).
11) Função com argumento objeto e retorno objeto (Objeto Pessoa com 3 atributos, nome, idade e peso).
12) Função com 2 vetores como parâmetro e retorno a distância euclidiana entre os vetores.

De modo que os testes foram realizados com a linguagem de programação Python 3.6, praticamente todos os tipos de dados especificados pelo proto3 foram utilizados nos testes, sendo eles **bool**, **int**, **long** (int64), **float** e **string**. O protocolo ainda especifica outros tipos de dados como uint32/64, sint32/64, fixed32/64 e outros[3], no entanto, em geral para a linguagem python o mapeamento desses tipos refletem em tipos já incluídos nos testes supracitados.

Para simular uma mensagem vazia, na implementação gRPC, definiu-se como retorno uma mensagem sem qualquer tipo de dados, do seguinte modo:

```
message void {

}
```

Já para a implementação do RPyC, a mensagem vazia (que não transmite nenhum dado e não retorna nada), é definida no servidor como um método que simplesmente não realizada nada e não recebe qualquer argumento, do seguinte modo:

```
def exposed_voidFunction(self):
    pass
```

Deixa-se explicíto que os testes utilizaram os mesmos dados para ambas as implementações, exatamente as mesmas strings, numeros inteiros e long, assim como os arrays, para manter o maior nível de controle possível.

Definidos os testes, passamos então para a descrição da configuração dos computadores utilizados para os testes locais e remotos.

B. Hardware

No total foram utilizados dois computadores para realização dos testes, sendo que uma dessas máquinas foi utilizada

[2]O SSL é um padrão desenvolvido em 1994 pela Netscape, seu objetivo é criptografar um canal de comunicação de modo que os dados sejam transmitidos de maneira sigilosa.

[3]A tabela completa do mapeamento de tipos pode ser melhor conferida nesta URL: https://developers.google.com/protocol-buffers/docs/proto3, na seção *Scalar Value Types*.

somente para testes remotos.

1) Máquina 1: A máquina 1 foi a escolhida para a realização de todos os testes locais, sendo cliente e servidor, fazendo para os testes remotos apenas o papel de cliente. As configurações da máquina 1 são:

- Sistema Operacional: **Ubuntu 18.04 (Bionic Beaver)**.
- CPU: **Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz**.
- Memória RAM: **16GB**.
- Memória Secundária: **1 HD de 1TB**.

2) Máquina 2: Já máquina 2, realizou somente o papel de servidor para os testes remotos, suas especificações são:

- Sistema Operacional: **Windows 10 PRO**.
- CPU: **Intel(R) Core(TM) i3-4170 CPU @ 3.70GHz**.
- Memória RAM: **4GB**.
- Memória Secundária: **1 HD de 1TB e 1 SSD de** .

Cabe ressaltar que para a realização dos testes, ambas as máquinas estavam sob a mesma rede local, no entanto, a máquina 1 permaneceu conectada via Wi-Fi, enquanto a máquina 2 estava conectada diretamente em um roteador através de um cabo de rede. Para criar um ambiente ideal de controle, tanto nos testes locais quanto nos testes remotos, a mesma configuração das máquinas foi mantida durante todo o processo de experimentação.

III. Metodologia

Antes da realização dos experimentos, tentou-se em ambas as máquinas manter em execução em primeiro plano somente os processos necessários para a execução dos testes, no entanto, os processos em segundo plano inerentes ao funcionamento dos sistemas operacionais foram mantidos em execução.

Importante ressaltar que durante os testes locais e remotos, em nenhum momento realizou-se a utilização de navegadores web, ou acessou-se por outros dispositivos na mesma rede qualquer site de hipermídia, para assim não gerar mais interferências e aumentar a variabilidade da latência na rede.

Além disso, durante os testes, apenas as duas máquinas estavam conectadas a rede, impedindo através do roteador a conexão de qualquer outro dispositivo que não fosse a máquina 1 ou a máquina 2.

Os testes locais e remotos foram realizados de maneira serializada, portanto, apenas uma implementação foi testada por vez (remotamente e localmente), fica assim explícito que em momento algum as implementações foram executadas paralelamente uma a outra, do mesmo modo para os testes locais e remotos, sendo cada um executado individualmente, evitando potenciais interferências entre os processos.

Dadas as precauções tomadas para realização dos testes sob um ambiente controlado, para cada implementação realizou-se um teste para cada função definida anteriormente, para facilitar a identificação de outliers e prover um valor médio mais próximo do desempenho real, cada função foi executada 512 vezes para cada implementação, provendo assim um conjunto de dados minimamente robusto para a análise da performance de cada implementação.

Logo, ao todo são gerados 4 conjuntos de dados, 2 conjuntos para os testes locais em cada implementação, e mais 2 para os testes remotos.

Por fim, com os conjuntos de dados criados, realiza-se a construção da visualização dos dados através de gráficos como **boxplots** e **gráficos de barras** para o comparativo entre cada função e implementação, e também **gráficos de pontos** para visualizar mais claramente a oscilação da latência localmente e remotamente para ambas as implementações.

Vale ressaltar que antes da visualização dos dados, foram eliminados os outliers que estavam acima do percentil 98, e abaixo do percentil 2, pois estes apresentavam valores extremos com muita discrepância em relação aos valores médios.

IV. Resultados Observados

Antes de iniciar a discussão sobre os testes realizados, vale indicar que os testes para a função de cálculo da distância euclidiana entre dois vetores fram omitidos dessas primeiras analises iniciais, isso porque o desempenho da implementação gRPC foi relevantemente superior ao RPyC, como será demonstrado posteriormente na seção de comparativos, assim como também será discutida a potencial razão para esta superioridade nos testes.

A. Análise gRPC

Para os primeiros testes locais do gRPC, fica evidente que a função mais custosa localmente é a função que recebe como parâmetro uma lista de strings e retorna a mesma lista. Em todos os testes, a lista de strings é composta por 200 strings *'abcdef'*, observa-se também que a função com mensagem vazia é uma das que possui menor custo, assim como o esperado, no mais, localmente as outras funções não apresentaram uma diferença tão significativa em termos de tempo de execução, como demonstra a figura 2.

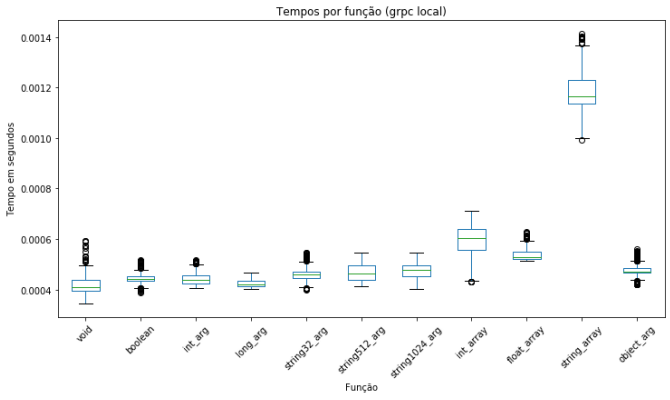

Figura 2. Boxplot para execuções locais.

Observa-se com maior riqueza de detalhes, que o erro padrão é maior justamente na função que utiliza um array de strings como parâmetro.

Outro fato interessante, ocorre nas funções com diferentes tamanhos de string, apesar de perceber uma sutil diferença

entre essas, o tempo de execução, considerando o erro apresentado, é praticamente o mesmo.

Do mesmo modo é interessante observar que o tempo de execução de uma função que serializa um objeto, é muito parecido com o tempo de uma função que serializa uma string.

Damos também destaque ao tipos mais primitivos como inteiros, longs e booleanos que em geral são mais simples e mais rápidos - característica que já era esperada pelo menor número de bytes necessário para ser transferido, como ilustra a figura 3.

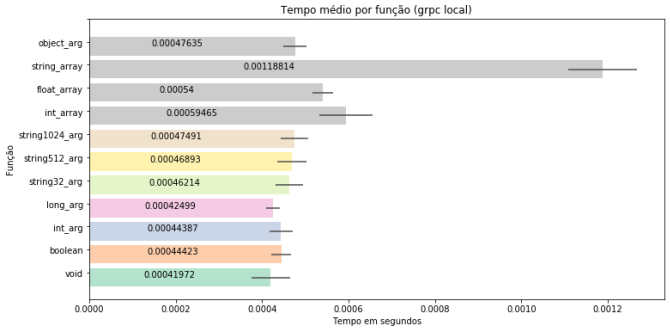

Figura 3. Barplot para execuções locais.

Já partindo para as execuções remotas, fica mais evidente o efeito da latência da rede no desempenho das funções de teste. Como se observa no boxplot para as execuções remotas, o número de outliers cresce bastante, principalmente os valores maiores, já que os outliers inferiores, diminui nos testes remotos.

Interessante observar que a função com um array de strings ainda continua tendo o maior custo entre todas, mas agora, percebe-se também, que as funções com um argumento simples de strings, mas de diferentes tamanhos parecem ter valores mais discrepantes entre si, tendo as chamadas com strings maiores, um tempo maior de execução. Neste momento, fica ainda mais evidente o efeito da latência da rede na chamada de procedimento remoto, como apresenta a figura 4.

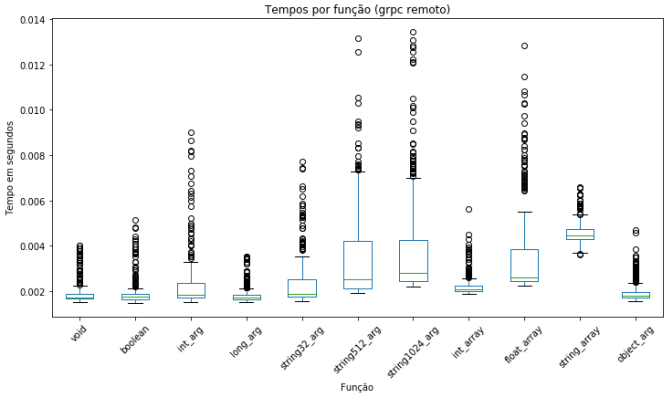

Figura 4. Boxplot para execuções remotas.

O gráfico de barras nos permite visualizar com mais clareza a diferença no tempo de execução nos métodos que utilizam strings como parâmetros em relação ao resto. Percebe-se que o erro aumenta consideravelmente para essas funções em relação ao todo.

Percebe-se também que se antes a função que passava um objeto tinha praticamente o mesmo tempo de execução que a função que passava strings, agora o cenário já é diferente, sendo tão rápida quanto a função com o menor tamanho de string testado.

Também é válido o destaque para as diferenças ressaltadas entre as funções que serializam arrays, observa-se agora que um array de inteiros leva o menor tempo para ser transferido, sendo mais rápido que a serialização de um array de floats, que por sua vez é mais rápido que a serialização do array de strings.

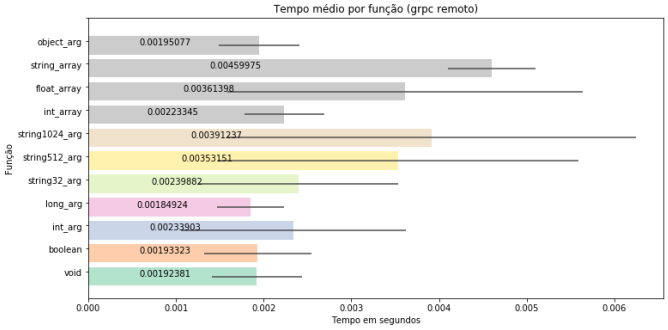

Figura 5. Barplot para execuções remotas.

Para ilustrar melhor o efeito da latência em chamadas remotas e locais, o gráfico de pontos mostra com clareza o aumento na variação do tempo de execução de uma das funções durante os testes remotos. Enquanto os pontos que representam os testes remotos, tem um comportamento mais espalhado no gráfico, os pontos que apresentam os testes locais se aproximam muito de uma reta.

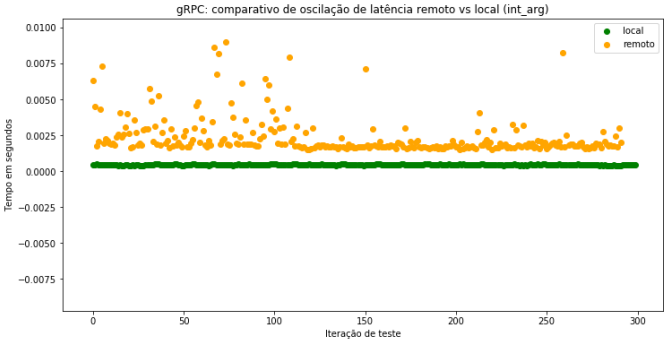

Figura 6. Latência de execuções locais vs remotas.

Finalmente, como esperado, os testes remotos são até uma ordem de grandeza mais lentos que os testes locais, a explicação para este fenômeno é autoevidente e está associada ao fato de todo o overhead adicionado para a transferência dos dados através de uma estrutura de rede, o que é mais complexo, e portanto mais caro, do que uma chamada remota para um processo local.

B. RPyC

Para a implementação RPyC, os resultados observados para os testes locais iniciaram um tanto diferentes dos observados para o gRPC. Como esperado, as funções que utilizam arrays tem um custo mais elevado que as funções que usam tipos primitivos mais simples.

O custo observado para as funções que trabalham com listas é muito similar neste caso, apesar de uma importante diferença na ocorrência de outliers nas funções que trabalham com strings.

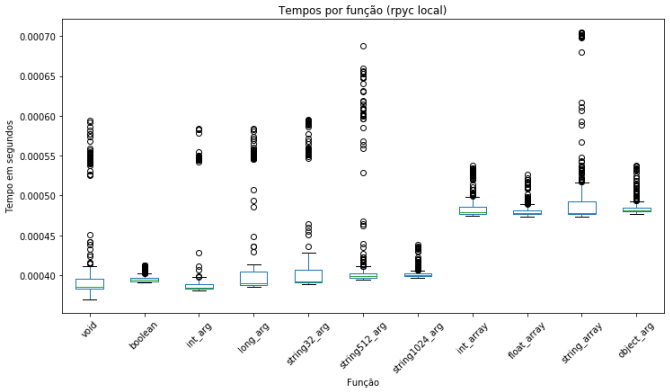

Figura 7. Boxplot para execuções locais.

No gráfico de barras, é visualizado com mais facilidade a uniformidade no tempo de duração das funções, apesar do erro aparentemente maior em relação as execuções locais do gRPC.

A revelação mais curiosa veio no fato de uma chamada vazia e sem nenhum dado estruturado como retorno ser tão custosa quanto as outras chamadas que serializam dados. Fato este que se repete no outro mecanismo de RPC.

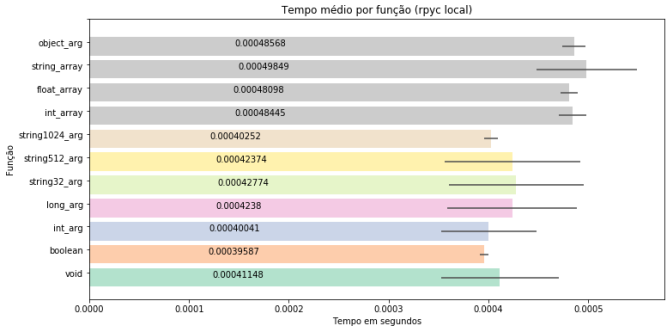

Figura 8. Barplot para execuções locais.

Partindo para a análise dos testes remotos, o boxplot revela um fato curioso em relação ao custo das funções, com o mecanismo RPyC, as chamadas remotas para as diferentes funções possuem um custo médio muito parecido, apesar de discrepâncias na ocorrência de outliers entre as funções. No entanto, é importante observar aqui que o custo das funções se eleva consideravelmente em uma ordem de grandeza.

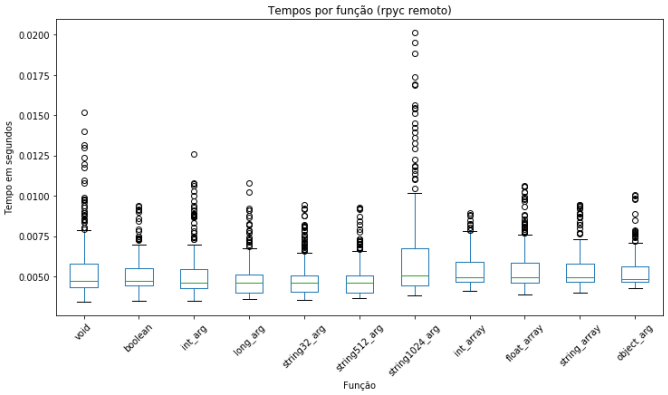

Figura 9. Boxplot para execuções remotas.

Através do gráfico de barras, damos destaque para a diferença no erro padrão que agora cresce muito mais, e mostra uma variação maior do que a observada no teste local anterior.

O gráfico da figura 10 ressalta ainda mais a uniformidade do custo das funções, durante chamadas remotas.

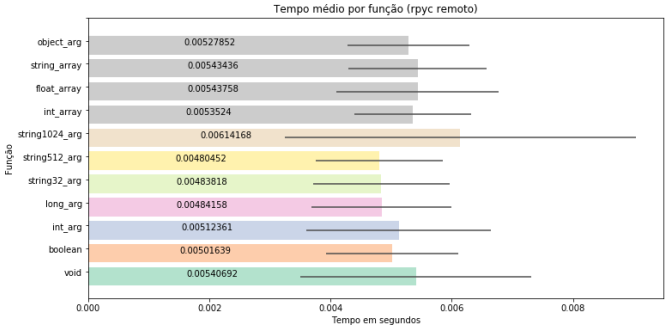

Figura 10. Barplot para execuções remotas.

Assim como no experimento realizado para a implementação anterior de RPC, observa-se agora no gráfico de latência uma diferença muito significativa entre as chamadas remotas e locais. Além disso, é importante dar destaque a diferença no tempo de execução observada, que agora é aparentemente maior em relação a implementação anterior.

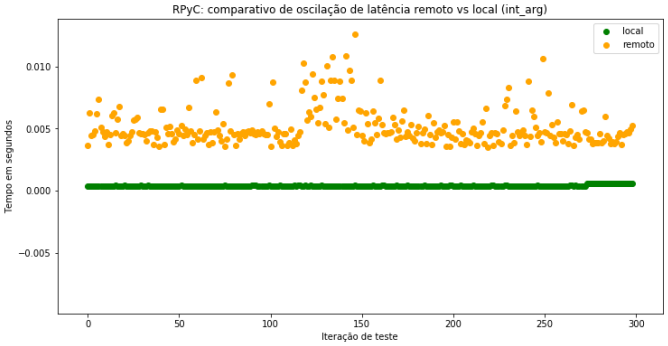

Figura 11. Latência de execuções locais vs remotas.

Fato interessante de ser observado na figura 11 é que os pontos das execuções remotas agora parecem mais espalhados, apesar dos pontos locais estarem ainda mais próximos de uma reta.

V. Comparativo: gRPC x RPyC

A. Comparativo local

O resultado para o comparativo em execuções locais, foi em geral um desempenho em termos de tempo de execução maior para a implementação RPyC, como observado na figura 12.

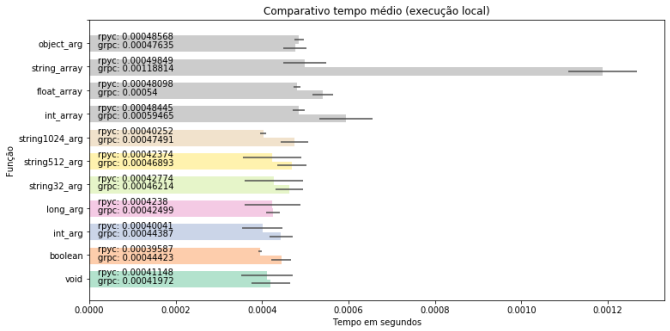

Figura 12. Comparativo local, gRPC x RPyC.

Uma das potenciais razões para esta vantagem é a não existencia de alguns *overheads* necessários para o funcionamento do gRPC, já que este tem uma implementação mais complexa pela sua natureza que permite uma implementação mais genérica e que funcione em mais de uma linguagem de programação com facilidade.

B. Comparativo remoto

Já no comparativo remoto, o gRPC apresenta um resultado relativamente superior em relação ao RPyC, apesar de neste momento os erros padrão serem maiores, fica claro que mesmo considerando a margem de erro, a primeira implementação ainda é mais eficiente.

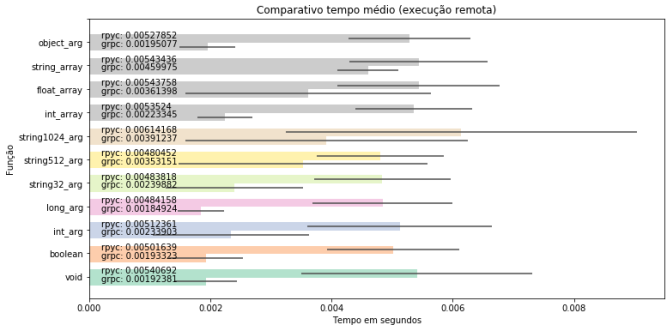

Figura 13. Comparativo remoto, gRPC x RPyC.

Este é um ponto muito positivo para a implementação da biblioteca *grpcio* e também para o buffer de protocolo para serialização dos dados estruturados, mostrando que a estratégia adotada pelo gRPC é superior a da segunda implementação quando o cenário são chamadas remotas entre máquinas distintas que podem se comunicar de algum modo através da rede.

C. A armadilha do object proxing

Como já citado anteriormente, o experimento possui 12 funções, sendo a 12ª um teste para o cálculo da distância euclidiana entre dois vetores de três dimensões.

Surpreendentemente, percebeu-se neste teste uma diferença muito discrepante entre as duas implementações, sendo os tempos diferentes em até uma ordem de grandeza, como mostra a figura 14.

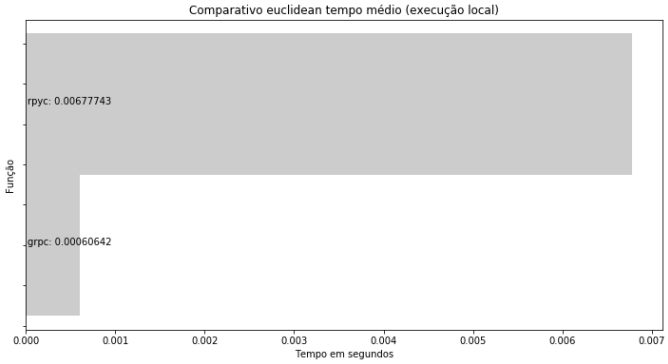

Figura 14. Comparativo local distancia euclidiana, gRPC x RPyC.

Em pesquisas posteriores, descobriu-se que um potencial problema que poderia estar afetando o desempenho da implementação RPyC neste cenário, seria a estratégia de boxing aplicando o object proxing.

Este espelhamento do objeto através do proxy poderia se tornar um gargalo no momento de conversão do array simples em um numpy array, já que as alterações no servidor nesta estratégia, precisam ser refletidas no cliente.

As estruturas de dados que não caem dentro de um object proxing são os definidos *simple types* na implementação, que são: bool, int, long, float, complex, basestring e type(None)[4]. Na implementação também existem outras estruturas declaradas como **TYPE_TUPLE** que não criam um object proxy.

Para testar as hipóteses, um novo teste foi realizado, utilizando tuplas no lugar de listas como passagem de parâmetros para calcular a distância euclidiana, o resultado é demonstrado na figura 15.

Apesar de diminuir o tempo de execução em uma ordem de grandeza, provando que o tipo de dado a ser serializado pode causar problemas em termos de tempo de exeucução, a "otimização"por assim dizer ainda não foi suficiente para superar o desempenho da implementação gRPC, como ilustra a figura 16.

[4]A implementação que mostra as estruturas citadas pode ser encontrada na URL: https://github.com/Mojang/play/blob/master/python/Lib/site-packages/Rpyc/Boxing.py

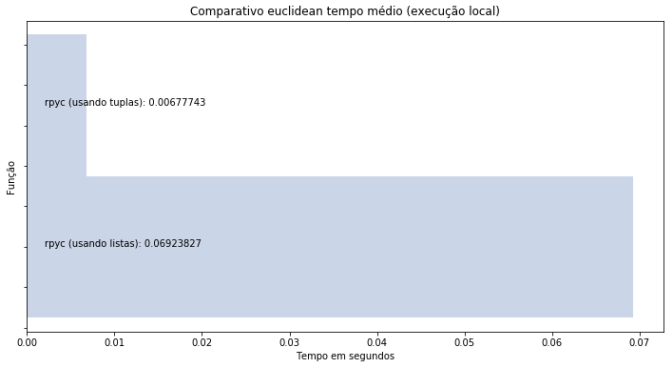

Figura 15. Comparativo local distancia euclidiana, RPyC (lista) x RPyC (tupla).

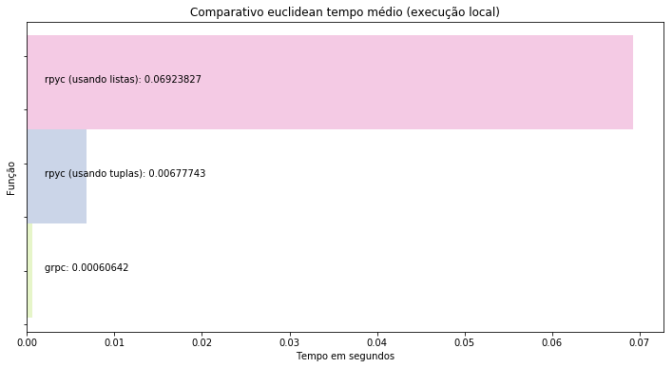

Figura 16. Comparativo local distancia euclidiana, RPyC (lista) x RPyC (tupla) x gRPC.

VI. Conclusão

Conclui-se com base nos experimentos observados, que a escolha de um mecanismo de RPC possui vários *trade-offs*, se por um lado a implementação gRPC é mais robusta em termos de linguagens e plataformas atendidas, para execuções locais, pode acrescentar um overhead que não compense o custo da implementação de um buffer de protocolo e assim aumentar a quantidade de código escrito.

Por outro lado, se a implementação RPyC é mais transparente e simples de ser implementada, apresenta-se mais ineficiente em termos de tempo de execução para chamadas remotas em computadores diferentes.

Claro que apesar dos propósitos gerais das bibliotecas serem diferentes, pois enquanto uma abrange uma grande gama de linguagens, a outra abrange de forma bem satisfatória a transparência em uma única linguagem, percebe-se que ambas possuem atuações mais eficientes em cenários mais especificos para uma mesma linguagem.

Portanto, para o usuário que deseja desenvolver um sistema distribuído em python com bom grau transparência e que o tempo de resposta não seja tão crítico, a biblioteca RPyC, poderá atender de maneira satisfatória.

Todavia, se o objetivo é ter um desempenho mais eficiente durante a execução do sistema distribuído, apesar do custo adicional da implementação do buffer de protocolo, a implementação gRPC oferece mais robustez e inclusive portabilidade para outras linguagens de programação.

Referências

[1] SCHMIDT, Douglas C. Overview of Remote Procedure Calls. Disponível em <http://www.cs.wustl.edu/ schmidt/PDF/rpc4.pdf>. Acesso em: 10/06/2018.

[2] SOARES, Patricia Gomes. On Remote Procedure Call. Disponível em <http://www.inf.puc-rio.br/ noemi/sd-09/rpc-survey.pdf>. Acesso em: 10/06/2018.

[3] WILBUR, Steve. BACARISSE, Ben. Building distributed systems with remote procedure call. Disponível em <https://ieeexplore.ieee.org/document/4807901/>. Acesso em: 10/06/2018.

[4] Language Guide (proto3). Disponível em <https://developers.google.com/protocol-buffers/docs/proto3>. Acesso em: 11/06/2018.

[5] RPyC Documentation. Disponível em <https://media.readthedocs.org/pdf/rpyc/latest/rpyc.pdf>. Acesso em: 01/07/2018.

[6] gRPC Concepts. Disponível em <https://grpc.io/docs/guides/concepts.html>. Acesso em: 01/07/2018.