PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE UMA BIBLIOTECA DE CHAMADA DE PROCEDIMENTO REMOTO NÃO BLOQUEANTE

LEONARDO BARBOSA DA ROSA

Trabalho de Conclusão I apresentado como requisito parcial à obtenção do grau de Bacharel em Engenharia da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Sergio Johann Filho

Porto Alegre
2022

“Se eu vi mais longe, foi por estar sobre ombros de gigantes.”
(Isaac Newton)

AGRADECIMENTOS

Primeiro, gostaria de agradecer ao professor Sergio Johann Filho, pela orientação dada neste trabalho e por proporcionar-me aprendizados de extrema valia para a vida acadêmica e profissional. Segundo, agradeço a participação da banca, composta pelo professor Cesar Augusto Missio Marcon.

Quero agradecer aos meus excelentes amigos de curso, principalmente ao Diego Oliveira e ao Ícaro Stumpf, pois me proporcionaram momentos e ensinamentos de extrema importância ao longo de minha trajetória.

Além disso, gostaria de agradecer a minha família e namorada, por me incentivarem a dispor o melhor de mim em tudo que realizo e por me acompanharem em cada etapa de minha construção profissional.

Por fim, quero expressar meus sentimentos à instituição por proporcionar contato com pessoas e profissionais incríveis ao longo dos últimos anos, bem como permitir e auxiliar-me na construção de uma empresa, hoje sedeada no Tecnopuc.

DESENVOLVIMENTO DE UMA BIBLIOTECA DE CHAMADA DE PROCEDIMENTO REMOTO NÃO BLOQUEANTE

RESUMO

Em ambientes distribuídos, a utilização de chamadas de procedimentos remotos tem-se tornado cada vez mais comum, disponibilizando a capacidade de processamento de forma remota e com facilidade de utilização para o desenvolvedor. Neste trabalho, é construída uma biblioteca RPC, com o propósito de prover uma interface de extrema facilidade de utilização, mas, ainda assim, embasada em técnicas robustas de otimização de envio e capacidade de processamento.

Para tanto, são empregados métodos de serialização binária, biblioteca de transporte construída sob UDP com controle de tráfego, utilização de diretivas de paralelização para otimização dos canais de envio e, por fim, a construção de uma arquitetura simples e intuitiva pra o desenvolvedor usufruir.

Em posse de uma biblioteca com essa estrutura, torna-se possível que o desenvolvedor utilize tecnologias e métodos complexos e de alta performance. Dessa forma, é possível obter um desempenho consideravelmente alto com uma interface de programação de fácil acesso e utilização.

Palavras-Chave: RPC, Serialização, Paralelização, UDP, Intuitiva.

DEVELOPMENT OF A NON-BLOCKING REMOTE PROCEDURE CALL LIBRARY

ABSTRACT

In distributed environments, remote procedure calls have become increasingly common, providing the processing capacity remotely and with ease of use for the developer. In this work, an RPC library is built to provide a straightforward interface based on robust techniques to optimize sending and processing capacity.

We implement the RPC library using binary serialization methods and a transport library built on UDP with traffic control and parallelization directives that optimize the send channels. This approach allowed us to build a simple and intuitive architecture for the developer.

In possession of a library with this structure, it becomes possible for the developer to use complex and high-performance technologies and methods. This way, considerably high performance can be achieved with a programming interface that is easy to access and use.

Keywords: RPC, Serialization, Parallelization, UDP, Intuitive.

LISTA DE FIGURAS

3.1 Camadas do Modelo TCP/IP em relação ao Modelo OSI 19
3.2 Formato de Cabeçalho do Datagrama do Usuário. 20
3.3 Chamada de Procedimento Remoto Bloqueante. 22
3.4 Chamada de Procedimento Remoto Não Bloqueante. 23
3.5 Diagrama representando a subdivisão dos intervalos na codificação aritmética 24
5.1 Formato dos metadados de uma estrutura 29
5.2 Estrutura na memória da construção do objeto *father* 30
5.3 Estrutura na memória da construção do objeto *son* 31
5.4 Resultado da serialização 31
5.5 Diagrama do macro funcionamento da biblioteca de transporte 32
5.6 Serialização do *payload* de envio 33
5.7 Diagrama de funcionamento da Biblioteca RPC 36
5.8 Representação do *payload* final enviado ao servidor 37
5.9 Representação do *payload* final enviado ao cliente 38
6.1 Teste com variação no tamanho do payload de 16 *bytes* a 1024 *bytes*, sem compressão de dados. 41
6.2 Teste com variação no tamanho do payload de 1K bytes a 64K bytes, sem compressão de dados. 42
6.3 Teste com variação no tamanho do payload de 64K *bytes* a 2M *bytes*, sem compressão de dados. 43
6.4 Teste com variação no atraso de rede, de 25ms a 200ms 44
6.5 Representação do *payload* enviado ao servidor com paralelização 44
6.6 Teste com variação na perda de pacote, de 3% a 24% 45
6.7 Comparação entre testes com compressão e sem compressão 46

LISTA DE ALGORITMOS

3.1 Exemplo de serialização em C# 21
5.1 Declaração da classe People 30
5.2 Inicialização dos objetos *father* e *son* 31
5.3 Estrutura de gerenciamento do tráfego de informações 33
5.4 Concatenação do cabeçalho de controle e a informação trafegada 34
5.5 Exemplo de instânciação da classe cliente 34
5.6 Alteração no método de decodificação 35
5.7 Alteração no método de codificação 36
5.8 Utilização da biblioteca de RPC no lado do servidor 38
5.9 Utilização da biblioteca de RPC no lado do cliente 39

LISTA DE SIGLAS

RPC – Remote Procedure Call

OSI – Open System Interconnection

IP – Internet Protocol

UDP – User Datagram Protocol

RMI – Remote Method Invocation

EBCDIC – Extended Binary Coded Decimal Interchange Code

ASCII – American Standard Code for Information Interchange

HTTP – Hypertext Transfer Protocol

IDL – Interface Definition Language

API – Application Programming Interface

ARPANET – Advanced Research Projects Agency Network

REST – Representational State Transfer

IBM – International Business Machines Corporation

TCP – Transmission Control Protocol

SMP – Shared Memory Parallelism

STL – Standard Template Library

SUMÁRIO

1 INTRODUÇÃO . . . 11
1.1 MOTIVAÇÃO . . . 12
1.2 OBJETIVO GERAL . . . 12
1.3 OBJETIVOS ESPECÍFICOS . . . 12

2 TRABALHOS RELACIONADOS . . . 13
2.1 NINF-G: RPC PARA COMPUTAÇÃO EM GRID . . . 13
2.2 DTN-RPC: CHAMADAS DE PROCEDIMENTO REMOTO PARA REDE TOLERANTE A INTERRUPÇÕES . . . 14
2.3 RPC PARA SISTEMAS DE TEMPO REAL . . . 14
2.4 GOOGLE GRPC . . . 15

3 FUNDAMENTAÇÃO TEÓRICA . . . 16
3.1 MODELO DE INTERCONEXÃO DE SISTEMAS ABERTOS . . . 16
3.2 MODELO DE REFERÊNCIA TCP/IP . . . 18
3.3 PROTOCOLO DE DATAGRAMA DO USUÁRIO . . . 19
3.4 SERIALIZAÇÃO . . . 20
3.5 PROTOCOLO DE CHAMADA REMOTA . . . 21
3.6 OPERAÇÕES BLOQUEANTES E NÃO-BLOQUEANTES . . . 22
3.7 OPENMP . . . 22
3.8 COMPRESSÃO DE DADOS UTILIZANDO CODIFICAÇÃO ARITMÉTICA . . . 23

4 METODOLOGIA . . . 25

5 DESENVOLVIMENTO . . . 27
5.1 SERIALIZAÇÃO DE DADOS . . . 27
5.1.1 MÉTODO DE UTILIZAÇÃO E DESCRIÇÃO DO FUNCIONAMENTO . . . 29
5.2 BIBLIOTECA DE TRANSPORTE . . . 32
5.3 ALGORITMO DE COMPRESSÃO . . . 35
5.4 BIBLIOTECA DE RPC . . . 35

6 RESULTADOS . . . 40
6.1 VARIAÇÃO NO TAMANHO DO *PAYLOAD* E *DATAGRAM*, SEM USO DE ALGORITMO DE COMPRESSÃO . . . 40

6.1.1 *PAYLOAD* DE 16 A 1024 *BYTES* 40
6.1.2 *PAYLOAD* DE 1K *BYTES* A 64K *BYTES* 41
6.1.3 *PAYLOAD* DE 64K *BYTES* A 2M *BYTES* 42
6.2 VARIAÇÃO DO TAMANHO DO *PAYLOAD* E ATRASO DE REDE, SEM USO DE ALGORITMO DE COMPRESSÃO 42
6.3 VARIAÇÃO DO TAMANHO DO *PAYLOAD* E PERDA DE PACOTE, SEM USO DE ALGORITMO DE COMPRESSÃO 44
6.4 VARIAÇÃO DO TAMANHO DO *PAYLOAD* E TAMANHO DO *DATAGRAM*, COM USO DE COMPRESSÃO 45

7 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS 47

REFERÊNCIAS 48

1. INTRODUÇÃO

O desenvolvimento da tecnologia da informação desde os anos 80 tem sido um avanço enorme e de grande valia para a sociedade. Na época passada, o desenvolvimento de soluções a partir da programação era vinculada a mercados específicos, com um propósito bem definido e voltado, principalmente, ao desenvolvimento de soluções de *software* embarcados ou *hardware* [dEdS22]. Com a constante e acelerada evolução dos transistores, tornou-se possível, cada vez mais, desenvolver *hardwares* robustos por um preço e tamanho menores e com isso, possibilitou que novas linguagens surgissem, com o nível de abstração cada vez mais alto [MVdCdM16].

A abstração das novas linguagens de programação, ocorre, principalmente, devido ao fato de que os desenvolvedores buscam resolver determinados problemas sem a necessidade de preocupar-se com artifícios a nível de máquina ou de funcionalidades que possam já ter sido implementadas por outros desenvolvedores. Ou seja, abstração e otimização são, normalmente, características excludentes e desejadas por muitos programadores [MVdCdM16].

Complementarmente, em um novo momento da tecnologia em que o desenvolvimento para plataformas disponibilizadas na internet são o auspício deste mercado, tem-se como consequência a necessidade de trafegar informações entre esses sistemas. No momento atual da evolução da tecnologia, existem protocolos para que essa comunicação seja possível com ênfase na facilidade de utilização para o desenvolvedor, no entanto, tornam viáveis problemas de otimização e segurança nas aplicações que as utilizam.

Com a evolução das linguagens de programação, foram desenvolvidos métodos de comunicação de fácil utilização, como a comunicação por JSON ou XML por meio do protocolo HTTP. Protocolos de comunicação e serialização como esses, fornecem facilidade de utilização e intercomunicabilidade, mas por padrão, não fornecem métodos de criptografia embutidos, assim como não se preocupam com um protocolo de serialização que otimize o tráfego de informação.

A necessidade da otimização do tráfego das informações dar-se-á ao fato de que, no presente, a internet está evoluindo para o uso de plataforma de contratação no modelo elástico, ofertados por grande empresas do mercado da tecnologia, como Google, Amazon, Oracle e outras. Essa modalidade de serviço, também conhecida como *pay-per-use*, tem se tornado cada vez mais comum no mercado, fornecendo ao contratante a característica de cobrança elástica, diretamente proporcional ao prevenimento de recursos pela contratada, em função de diversos fatores, como o volume de informações, armazenamento e processamento. Por essa razão, recursos com a possibilidade de realizar compressão de dados e reduzir o volume de tráfego ocasionam, por consequência, a redução de custo ao cliente.

1.1 Motivação

A partir do problema identificado, entende-se que o cuidado com a otimização da quantidade de fluxo da informação tem-se tornado, cada vez mais, uma necessidade. Pois, ao reduzir a quantidade de informação trafegada, mantendo a integridade desta, e ainda aplicando algoritmos de compressão de informação, torna-se possível aplicar a complexidade do processamento a nível de aplicação, ao invés de manter esse processamento na camada de transporte. Por essa razão, a motivação desse desenvolvimento é o aperfeiçoamento no que se refere à utilização dos serviços citados, diminuindo os custos, aumentando a segurança e o desempenho, bem como a abstração da utilização dessas bibliotecas de comunicação para os desenvolvedores.

1.2 Objetivo Geral

O objetivo geral do presente trabalho é desenvolver uma biblioteca de comunicação remota, também conhecida como RPC, para a linguagem C++. Essa biblioteca aplicará métodos de serialização, bem como algoritmos de compressão de dados bem quistos pelo mercado da tecnologia.

Portanto, tal implementação prestaria importante auxílio aos desenvolvedores que façam uso da linguagem de programação C++ para protocolos de comunicação na internet na obtenção de maior performance e segurança, bem como na redução de custos na utilização de serviços reconhecidos no mercado.

1.3 Objetivos Específicos

- Realizar um levantamento dos trabalhos relacionados do estado da arte;
- Desenvolver uma biblioteca de comunicação remota;
- Avaliar os resultados de maneira comparativa.

2. TRABALHOS RELACIONADOS

Perante a constante evolução da internet e a necessidade de métodos de comunicação direcionados à utilização do desenvolvedor, foram desenvolvidas tecnologias e bibliotecas com o propósito de suprir essa necessidade. A necessidade da abstração provém da evolução dos sistemas para um ambiente de programação distribuída, isso pois configurar e utilizar esses ambientes de desenvolvimento não é, de forma alguma, banal.

Conforme é possível verificar com a criação de diversas bibliotecas de RPC, há uma busca constante por novas tecnologias e soluções que proveem a capacidade de realizar, remotamente, chamadas de processamento de maneiras intuitivas e robustas para os desenvolvedores. Isso pois, perante a grande diversidade de linguagens de programação e interfaces de comunicação disponibilizadas para uso em aplicações, tem-se tornado cada vez mais comum a criação de bibliotecas que abstraiam parte da codificação, uma vez que a chave para o desenvolvimento rápido, eficiente e com a capacidade de fácil manutenção é amplificado pelo nível de abstração disponibilizado por boa parte das bibliotecas desenvolvidas [Str03].

Com base neste contexto, no presente trabalho foi construída uma biblioteca RPC, cujo propósito é prover uma interface intuitiva, robusta e otimizada aos desenvolvedores. Para tanto, são empregados métodos de serialização binária, biblioteca de transporte construída sob UDP, com controle de tráfego, utilização de diretivas de paralelização para otimização dos canais de envio, e por fim, a construção de uma arquitetura simples e intuitiva pra o desenvolvedor usufruir.

Em posse de uma biblioteca com essa estrutura, torna-se possível ao desenvolvedor utilizar tecnologias e métodos avançados para o desenvolvimento de software distribuído de alto desempenho. Dessa forma, tem-se um desempenho consideravelmente alto com uma interface de programação de facil acesso e utilização. que anteriormente eram demasiadamente utilizados para computação de alto desempenho, cada vez mais estão sendo utilizados para conceitos e aplicações comuns na internet; por essa razão, existem diversas implementações de bibliotecas de RPC com propósitos diferentes que possuem extrema relevância para a construção do presente trabalho.

2.1 Ninf-G: RPC para computação em Grid

Como citado, a computação distribuída é resultado da necessidade de ambientes de alta performance. Nesse trabalho relacionado, foi desenvolvida uma estrutura para computação em Grid [JBF+05], com o propósito de facilitar a utilização de uma tecnologia já difundida na comunidade de desenvolvimento.

Por essa razão, Tanaka, Nakada, Seikiguchi, Suzumura e Matsuoka criaram uma implementação do RPC, no modelo cliente-servidor no qual o cliente executa uma chamada remota, enquanto o servidor fornece ferramentas para construir e invocar a chamada, essa biblioteca foi nomeada de Ninf-G [TNS+03]. O grande diferencial dessa implementação é o fato de que os autores prezaram pela simplicidade na utilização, quando comparado a outras bibliotecas utilizadas como base, como a própria Ninf [SNS+97] ou NetSolve [SYAD05].

2.2 DTN-RPC: Chamadas de Procedimento Remoto para Rede Tolerante a Interrupções

Há implementações diversas em se tratado de chamada de procedimento remoto, isso pois é fato que nesse processo de comunicação em ambientes distribuídos ainda há muito a ser aperfeiçoado. Diferente do Ninf-G, os autores John Backan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove e Hadia Ahmed [SBM+17], implementaram uma biblioteca de RPC tolerante a atrasos e interrupções.

Cenários com atrasos e falhas podem ocorrer devido a interrupções de comunicações em links físicos ou rupturas de rede. Tais acontecimentos comuns, por exemplo, em desastres naturais. Por essa razão, os autores propõem a implementação de uma arquitetura de redes de computadores que vise a não ocorrência de problemas técnicos em redes heterogêneas por falhas de conectividade, nomeadas de DTN.

2.3 RPC para Sistemas de Tempo Real

O contexto da utilização do protocolo de comunicação altera consideravelmente as exigências técnicas das chamadas remotas. No trabalho desenvolvido por Han Namgoong e Myung-Joon Kim [NK99], a preocupação foi quanto à otimização em chamadas de procedimento remoto em sistemas de tempo real.

Essa necessidade de implementação surgiu a partir do fato de que os serviços já desenvolvidos para esse tipo de sistema atuam de forma em que o cliente realiza a chamada ao servidor e aguarda a resposta ou a expiração por um determinado tempo. Por mais que seja possível executar essas ações de forma assíncrona, ou seja, não bloqueante, ainda assim tornava-se um problema, visto que o cliente havia de esperar o *timeout* para realizar a chamada a outro servidor e, assim, conseguir a sua resposta; o servidor também desperdiçava tempo de processamento caso o tempo limite fosse atingido.

Por essa razão, os autores implementaram uma biblioteca cuja atuação rege uma resposta do servidor ao cliente, confirmando que a execução será realizada e, com isso,

dando maior certeza ao cliente que o *timeout* não será atingido. Caso o servidor não responda ao cliente em um tempo determinado, o cliente realiza outra chamada ao servidor avisando-o que o procedimento executado anteriormente não é mais de seu interesse, e com isso, libera o servidor para processar outras requisições.

2.4 Google gRPC

Ao mesmo tempo em que existem soluções específicas para determinadas situações, há grandes empresas do mercado da tecnologia visam a criação de soluções genéricas, capazes de funcionar em diversas linguagens de programação e com uma estrutura própria e extremamente funcional e otimizada. Esse é o caso da Google Inc e sua iniciativa o gRPC. O gRPC é uma estrutura de chamada de procedimento remoto de código aberto e de alto desempenho, com ênfase na modernização da sua utilização e a interoperabilidade de sistemas e linguagens de programação [Inc22].

A estrutura desse RPC foi montada sob a necessidade de uma definição simples de serviço, por meio de um conjunto de ferramentas e linguagem de serialização binário próprio da Google, o Protocol Buffers [Inc12], com a capacidade de serializar dados estruturados por meio de uma linguagem de descrição de interface (IDL).

A IDL definida pelo Google permite que seja descrito perfeitamente estruturas de envio e recebimento de mensagens, informando os tipos do objeto de resposta, bem como, os parâmetros que serão enviados a uma chamada remota. Além disso, é possível definir outras estruturas, como listas e chamadas recursivas - funcionalidades não comuns em recursos RPCs.

O gRPC, desde a sua concepção, em 2015, foi redigido sob o protocolo de transporte HTTP/2. Como consequência dessas tecnologias de ponta, é possível fornecer uma série de funcionalidades, tais como a própria interface de descrição de interfaces, autenticação, comunicação bidirecional, tráfego de *streaming* de dados, controle de fluxo, chamadas bloqueantes ou não bloqueantes e cancelamentos de chamadas por tempo excedido. Com todas as funcionalidades e facilidades fornecidas, atualmente várias organizações mundialmente reconhecidas adotam a utilização desse conjunto de ferramentas, tais como a Uber [Inc19], Netflix, IBM, Docker, Cisco, Spotify [grp19], DropBox [IG20], entre outras.

3. FUNDAMENTAÇÃO TEÓRICA

Nessa seção serão abordados conceitos intrínsecos às necessidades técnicas dispostas no presente trabalho, tais como conceitos de rede, serialização, compressão, chamada remota, aplicações paralelas e outros aspectos relevantes. Dessa forma, define-se uma série de conceitos importantes para a elaboração deste trabalho de conclusão de curso.

3.1 Modelo de Interconexão de Sistemas Abertos

Com o notório avanço da tecnologia da computação, foi disposto um movimento natural de padronizações, como protocolos de comunicação, práticas de segurança, desenvolvimento de código e trabalho em software, entre outros. O *Open System Interconnection* (OSI), foi criado pela Organização Internacional de Normalização com o objetivo de padronizar o modelo de comunicação entre sistemas de computadores. O modelo OSI é baseado no conceito de 7 camadas abstratas, no qual cada camada protocolo implementa uma funcionalidade assinalada a uma determinada camada [DZ83].

Camada Física

A camada física, a primeira do modelo OSI, é a que diz respeito aos meios de conexão que irão trafegar as informações, como as interfaces seriais ou os tipos de cabos e protocolos que serão seguidos a nível matemático. De maneira simplificada, uma das maneiras de se transmitir informações é a partir da utilização de tensão, ou corrente, que são propriedades físicas dos meios de propagação, como os fios. É possível, no entanto, transformar essas propriedades em função do tempo, transformando-as em um modelo que seja capaz de utilizar análise matemática [Tan11].

Camada de Enlace

A camada de enlace, também conhecida como camada de ligação de dados ou de link de dados, é a segunda camada do modelo OSI. Essa, possui a finalidade de detectar e, possivelmente, corrigir erros que ocorram na camada inferior. No demais, faz-se também o controle da recepção, delimitação e transmissão dos *frames* de informação, assim

como estabelece o protocolo de comunicação entre os sistemas que estiverem diretamente conectados [Tan11].

Camada de Rede

A camada de rede trata as transferências dos pacotes entre a origem e o destino. Ao contrário das camada de enlace, que não apenas transferem os dados de um ponto ao outro, mas sim realizam saltos entre os roteadores dispostos no caminho do tráfego da informação. Esses saltos, denominados como *hops*, ocorrem devido à topologia em que a rede está atuando, e com isso, permite que sejam determinados caminhos para evitar a sobrecarga da rede [Tan03].

Camada de Transporte

A etapa responsável pela transferência das informações, isto é, a de transporte, trata dos objetos independente da aplicação, da topologia e dos meios físicos utilizados. Essa camada oferece serviços para o nível superior com o objetivo de permitir a criação e utilização de aplicações de forma independente da implementação da rede.

Há dois tipos de serviços estabelecidos nessa camada, ditos como com conexão e sem conexão. Serviços orientados à conexão possuem um protocolo de maior confiabilidade, pois é realizado o controle do fluxo de informações, de modo a evitar congestionamento na rede e a perda de pacotes, assim como é garantido a ordenação das informações pelo receptor. Por outro lado, protocolos não orientados à conexão não possuem controle de tráfego embutido, todavia a falta do fornecimento destes serviços de controle acarretam em um fator compensador: a velocidade. Dessa maneira, recomenda-se a utilização de procolos não orientados à conexão, como UDP, quando o objetivo é agilidade na entrega das informações, ou seja, esse modelo de comunicação é preferencial para aplicações dependentes de desempenho [Tan11].

Camada de Sessão

Em um nível acima da camada de transporte, encontra-se a camada de sessão, a qual controla os processos que gerenciam a transferência das informações, gerenciando os erros e os registros de tráfego. Como dito por Gallo e Hancock [Gal03], essa camada é responsável por controlar o fluxo entres os nós. Dessa forma, são empregadas diversas metodologias de sincronização das trocas de informações.

Diferente da camada de transporte, essa não gerencia a comunicação entre duas máquinas, mas sim entre duas aplicações em uma mesma máquina. Isso pois, em função desse mecanismo é possível prover gerenciamento de conexões, e com isso, maior segurança, por meio de autenticação, por exemplo, bem como a sincronia entre as aplicações envolvidas.

Camada de Apresentação

Antes de chegar na última camada, é necessário que haja o reconhecimento e tratamento das informações repassadas, para que com isso, seja possível traduzir os dados de modo a serem cabíveis de interpretação pelas aplicações envolvidas. Por essa razão, existe a camada de apresentação.

Essa formatação é necessária, pois o modelo de tráfego das informações e o modelo, sintático, da apresentação destes para as aplicações envolvidas são divergentes. Um exemplo disso é o trabalho de conversão realizado para trafegar um arquivo de texto codificado no formato EBCDIC, que se trata, resumidamente, de um formato de codificação de caracteres em 8 bits, criado pela IBM [Mac80], para a codificação ASCII, passível de leitura e identificação pelo usuário e aplicações finais;

Camada de Aplicação

Por fim, a última camada consistem em uma abstração para contemplar protocolos que efetuam a comunicação ponto a ponto entre aplicações. Essa abstração, existe para prover serviços para os programas de modo a decernir a existência de comunicação em rede entre processos de diferentes máquinas.

Essa camada, por ser a mais próxima ao usuário, torna-se responsável por prover serviços quando o cliente acessa suas aplicações, como páginas na internet, e-mail, troca de mensagens, entre outras. Os protocolos utilizados nesse nível sempre atuam em conjunto com aqueles da camada de transporte, como TCP/IP e UDP, isso pois, juntos definem como as partes trocarão mensagens entre si.

3.2 Modelo de referência TCP/IP

Esse modelo pode ser conceituado como um conjunto de protocolos de comunicação entre máquinas. Seu nome é proveniente de dois protocolos, TCP e IP. Assim como

o modelo OSI, seu modo de operação pode ser compreendido como camadadas, em que cada uma dessas fornece um conjunto de serviços definidos para o nível superior.

O TCP/IP teve sua história iniciada em 1969, quando criado pelo U.S *Department of Defense Advanced Research Projects Agency*, para que fosse possível realizar um projeto, até então experimental, denominado como ARPANET (*Advanced Research Project Agency Network*). Esse projeto, em 1972, após grande crescimento na comunidade internacional, tornou-se o que hoje é conhecido como Internet.

Essa definição é o que hoje domina a internet, isto é, utiliza-se no mundo de comunicação o Modelo de referência TCP/IP ao invés do modelo OSI. O funcionamento, em comparação com o modelo teórico, é semelhante, porém utilizam apenas de 4 camadas, ao invés de 7, com uma certa equivalência entre os níveis, como é possível visualizar na Figura 3.1.

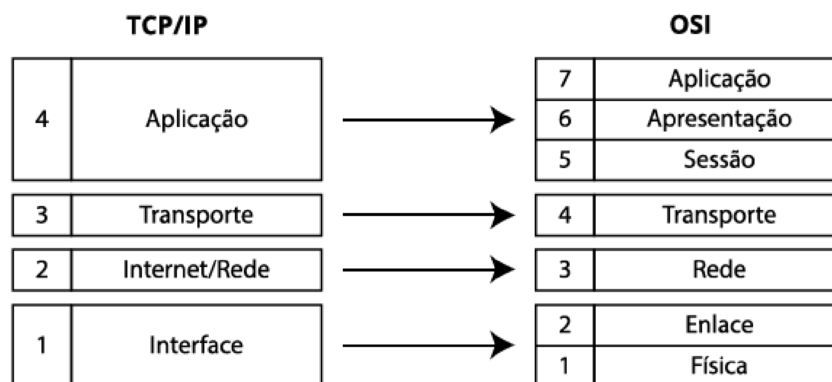


Figura 3.1: Camadas do Modelo TCP/IP em relação ao Modelo OSI
Fonte: Autor

3.3 Protocolo de Datagrama do Usuário

Utilizando o *Internet Protocol* (IP) como protocolo subjacente, implementou-se o *User Datagram Protocol* (UDP), definido para disponibilizar um modo de datagrama de comunicação de computador comutada por pacote no ambiente de um conjunto interconectado de redes de computadores.

Este protocolo disponibiliza um procedimento para que um programa tenha a capacidade de enviar mensagens para outros com mínimo de mecanismos de protocolo, [rfc80]. O UDP é orientado ao envio de *datagram*, sem manter uma conexão entre as partes, por essa razão, não garante a entrega ou proteção das informações.

O UDP fornece serviços que não são fornecidos pela camada IP, tais como o número da porta, auxiliando na distinção das diferentes solicitações de usuários, bem como um recurso de validação da integridade dos dados a partir de uma funcionalidade de soma, resultando na verificação da correta transferência dos dados, como é possível visualizar na

Figura 3.2. Esse protocolo é utilizado, na maioria dos casos, em comunicações sensíveis ao tempo, isso pois, pode ser mais vantajoso descartar pacotes do que esperar a correção de um determinado envio, como exemplos de informações sensíveis ao tempos temos voz e video, que são projetadas para lidar com pequenos níveis de perda.

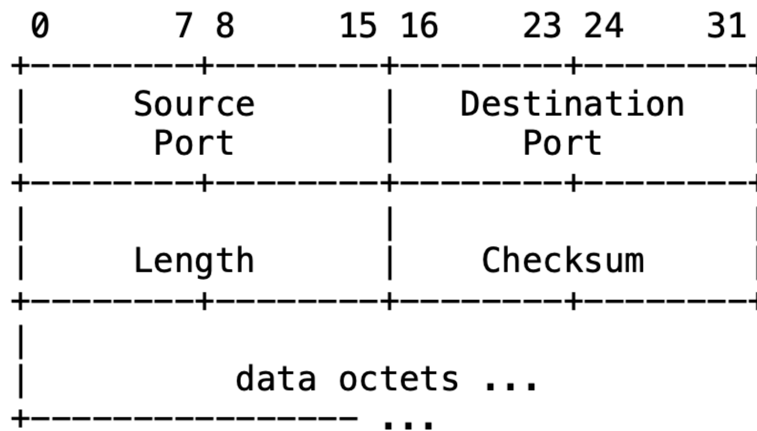


Figura 3.2: Formato de Cabeçalho do Datagrama do Usuário.
Fonte: [rfc80]

3.4 Serialização

A serialização é um processo utilizado em muitas linguagens de programação para converter uma determinada informação em um fluxo de *bytes* com o objetivo de armazenar e transmitir essas informações de maneira independente de arquitetura do sistema operacional envolvido. Esse processo consiste em uma padronização de um conjunto de informações, com determinadas informações e tipos de dados.

Um processo de serialização é um modelo de codificação das informações em um determinado padrão. Em função disso, faz-se necessário, então, o processo reverso para aquele que for realizar a leitura ou interpretação deste fluxo de dados. Tal processo acarreta em diversas possibilidades de utilização, tais como: trafegar dados entre diferentes aplicações, até mesmo em diferentes máquinas e sistemas operacionais; salvar informações em banco de dados; entre outras. Além disso, há um conjunto de soluções provenientes ao processo de serialização, como a representação compacta e não ambígua da informação, que possibilita a interoperabilidade entre sistemas construídos para diferentes plataformas, tanto de *hardware* quanto de *software*, assim como diferentes linguagens.

Dentre as linguagens de programação existentes, há aquelas que não possuem tal funcionalidade implementada nativamente, como é o caso de C e C++, bem como, há os casos em que isso é disponibilizado nas bibliotecas originais da linguagem, como é o caso de C# e Java. No Algoritmo 3.1, é possível visualizar um pequeno trecho de código que

ilustra a facilidade de serializar uma determinada classe em C#, dada a instância prévia, conforme é possível verificar no trecho comentado.

```
1  /*
2  [Serializable]
3  public class People {
4      public int age;
5      public String name;
6  }
7  */
8
9  People sergio = new People();
10 sergio.age = 40;
11 sergio.name = "Sergio";
12
13 IFormatter formatter = new BinaryFormatter();
14 Stream stream = new FileStream("people.bin", FileMode.Create, ...
       FileAccess.Write, FileShare.None);
15 formatter.Serialize(stream, sergio);
16 stream.Close();
```

Algoritmo 3.1: Este algoritmo trata-se de um exemplo simples de serialização na linguagem de programação C#. Fonte: Autor

3.5 Protocolo de Chamada Remota

Baseado no protocolo IP e no padrão de comunicação estabelecido por UDP, sem conexão e por datagramas, é viável utilizar tecnologias para possibilitar que uma das partes acione a outra, de forma remota, isto é, um modelo de comunicação entre uma ou mais partes com propósito diferente de apenas o envio de trechos de informação, como o caso do UDP. Um dos modelos que permite esse tipo de intercomunicação é o Protocolo de Chamada Remota (RPC), uma tecnologia de comunicação entre processos que permite que uma parte acione remotamente outra qualquer, ou seja, que um programa de computador chame um procedimento remoto em outro espaço de endereçamento.

Esse modelo, trata-se de um paradigma popular para comunicação entre processos em diferentes computadores na rede, vastamente utilizado em Sistemas Distribuídos. Embora conceitualmente seja de fácil utilização, há diversos problemas diferentes envolvidos, e por essa razão, existem tantas variações de RPC, como Xerox Courier RPC, Sun ONC/RPC, gRPC e outros [TA90].

3.6 Operações Bloqueantes e Não-Bloqueantes

Em programação, operações bloqueantes ou não bloqueantes referem-se a chamadas de recursos que exigem um determinado tempo de computação. Dessa maneira, as chamadas que bloqueiam a execução do algoritmo enquanto o resultado não é obtido é denominado como chamadas síncronas. Por outro lado, operações que não bloqueiam a execução em função do tempo de processamento, são denominadas como assíncronas.

Métodos não bloqueantes podem ser tratados de diversas formas, tais como a utilização de *callbacks*, que são métodos atribuídos a uma função bloqueante na sua execução, para que, ao final de sua computação, o método *callback* seja executado. Outra maneira de tratamento é a utilização de interrupções via software, isto é, no momento em que o método não bloqueante finalizar ele executará uma ação de interrupção com o propósito de executar um determinado trecho de código no momento desta interrupção. Para fácil compreensão, abaixo estão dispostas figuras representativas sobre o modelo de execução dos tipos de chamadas bloqueantes e não bloqueantes.

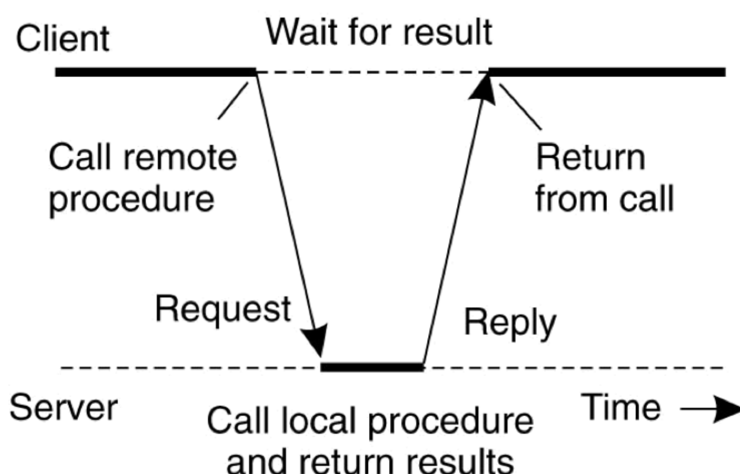


Figura 3.3: Chamada de Procedimento Remoto Bloqueante.
Fonte: [Fig21]

3.7 OpenMP

Na década de 1980, o computador era construído sobre a ideia de que o um processador de maior eficiência era o que movia a computação para maiores poderes computacionais [Alr21]. Ao longo dos anos, demonstrou-se que a quebra desse paradigma para o processamento paralelo foi o que essencialmente resolveu o problema de computação, fazendo com que tenhamos evoluído tanto em relação a capacidade de processamento.

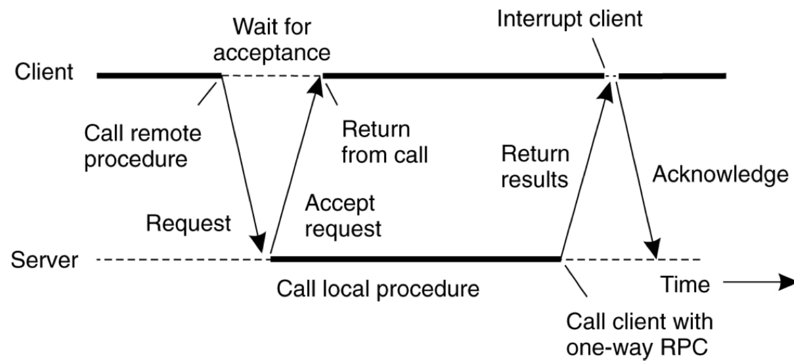


Figura 3.4: Chamada de Procedimento Remoto Não Bloqueante.
Fonte: [Fig21]

A abordagem da programação paralela, significa conectar vários núcleos com memória compartilhada, para aumentar conjuntamente a velocidade e a eficiência. Todavia, ao escolher um modelo de programação paralela, deve ser considerado o método em que essa capacidade de processamento é explorada [Alr21].

Em função da necessidade da época, o OpenMP foi criado com o intuíto de ser uma interface de programação de aplicativo para a programação multi-processo de memória compartilhada em múltiplas plataformas, ou como dito por David Clark, trata-se de um conjunto de diretivas de compilador e biblioteca de tempo de execução que pode ser chamado através de rotinas que aproveitam os ambientes de multiprocessador de memória compartilhada (SMP) [Cla98].

3.8 Compressão de dados utilizando codificação aritmética

A codificação aritmética pode ser entendida como uma tecnologia de compressão de informações sem a ocorrência de perdas em uma sequência de símbolos em distribuições independentes e identicamente distribuídas, isto é, cada variável aleatória possui a mesma distribuição de probabilidade da demais ao mesmo tempo que são mutuamente independentes (i.i.d.) [LGH19].

Essa técnica de compressão não é baseada em tabelas ou símbolos. Em função da eliminação da associação entre símbolos individuais e códigos, é capaz de praticamente igualar a entropia da fonte em todos os casos. Por meio de um modelo estatístico, constrói-se uma tabela que contenha a informação da probabilidade do próximo símbolo lido ser um dos símbolos possíveis, de maneira simplificada, nada mais é do que a contagem da ocorrência do símbolo dado todos os símbolos, como é definido pela fórmula abaixo:

$$P(\sigma) = \frac{n_\sigma}{N}$$

Nessa representação, P(σ) é a probabilidade da ocorrência de σ enquanto n_σ é o número de ocorrências e N é o tamanho da informação original. Dessa maneira, partindo de um intervalo [0,1), é identificado um sub-intervalo em que um símbolo é representado, dividido por representações probabilísticas, conforme a tabela de intervalos gerada no início do algoritmo, como é possível visualizar na Figura 3.5.

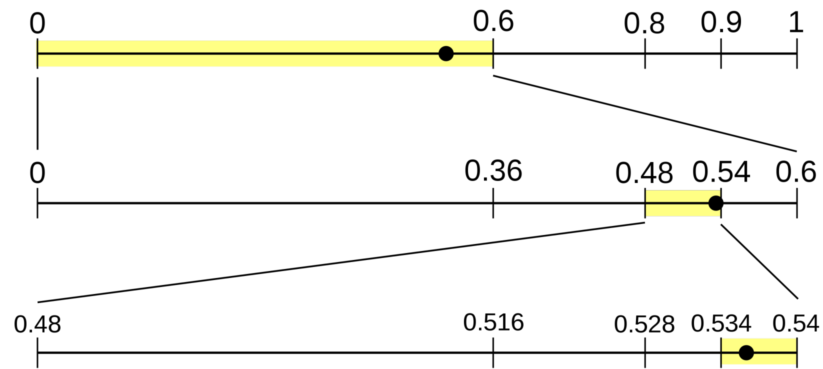


Figura 3.5: Diagrama representando a subdivisão dos intervalos na codificação aritmética
Fonte: [Wik22a]

4. METODOLOGIA

O trabalho decorre por meio de uma revisão bibliográfica, do método qualitativo e explicativo, através da identificação, por parte do autor, de uma possibilidade de melhoria em ferramentas já disponibilizadas no mercado da tecnologia. Para tanto, fez-se a utilização de uma metodologia dedutiva. Conforme Prodanov e Freitasc [PDF13], o método dedutivo parte de um entendimento geral até o particular. Isso, por meio de princípios, leis ou teorias e por meio da lógica, concebe casos particulares.

Abaixo está descrito a metodologia seguida para o desenvolvimento do presente Trabalho de Conclusão de Curso:

1. **Determinação do tema**: Nesta fase, foi verificada a necessidade, dado um subconjunto de um contexto já explorado, cujo detalhe se encontra no início das atividades referente a esse trabalho, bem como, avaliado em conjunto com o orientador do presente trabalho;

2. **Estudo de bibliografia**: Uma vez organizado o propósito do trabalho, foi efetuado um levantamento sobre os recursos bibliográficos disponíveis para a fundamentação teórica do assunto proposto;

3. **Estudo de documentação**: Após o estudo das referências bibliográficas necessárias para a estruturação do trabalho, foi necessário o estudo das documentações existentes das bibliotecas propostas por grandes empresas do mercado da tecnologia. Essa etapa teve como finalidade entender quais eram os pontos estratégicos descritos por essas empresas, desde o modelo de formalização, descrição do protocolo, camada de abstração disposta ao desenvolvedor e as funcionalidades específicas de cada implementação;

4. **Estruturação lógica**: Conforme a evolução do estudo do trabalho, tornou-se necessária a construção de artefatos lógicos para a construção da escrita formal do TCC, bem como o modelo de execução, dado que o trabalho proposto consiste em um conjunto de funcionalidades que devem ser coerentes e passíveis de implementação em conjunto, com um propósito bem definido, em nível de estrutura e de ordem de implementação;

5. **Construção do documento**: Com o conteúdo estruturado, iniciou-se a construção do documento do trabalho. Nesse, o propósito foi de ambientar-se com o modelo de escrita e pesquisa, bem como, com os artifícios formais de expressão da pesquisa. Com isso, fora concebido a partir da introdução e fundamentação teórica, em conjunto com a referência bibliográfica, e por fim, a construção do capítulo de metodologia;

6. **Desenvolvimento do projeto**: Após a construção do último capítulo citado e a organização do trabalho, teve inicio o desenvolvimento das ferramentas e codificações necessárias para futuros testes da biblioteca.

7. **Conclusão e comparações**: Por fim, definiu-se a conclusão do projeto, a partir das comparações dos resultados obtidos, pela biblioteca desenvolvida e bibliotecas e *frameworks* de terceiros. Dessa maneira, foi possível visualizar o quanto o trabalho foi coerente com a hipótese concebida inicialmente, bem como se houve, de fato, resultado sobre o desenvolvimento.

5. DESENVOLVIMENTO

5.1 Serialização de dados

Para que seja possível realizar o tráfego de informações de modo sucinto e com maior segurança, foi desenvolvido um protocolo de serialização, cujo objetivo é permitir a fácil composição ou decomposição dos tipos primitivos de dados em C++ no formato de maior economia de espaço possível. Para tornar viável essa implementação, inicialmente foi necessário validar a implementação de outras linguagens de programação, como já mencionadas, e os seus modos operante, de maneira generalizada, encontrou-se dois propósitos gerais para a serialização:

1. Interobabilidade entre sistemas operacionais: a capacidade de trafegar dados entre sistemas operacionais distintos, porém em uma mesma linguagem de programação. Dessa forma, torna-se possível otimizar o processo de serialização, de modo a diminuir as informações de controle e diminuir o tráfego entre os sistemas. Esse abrange os casos das serializações nativas de C# e Java;

2. Ainda, há as serializações cujo propósito é não apenas a comunicação entre sistemas operacionais, mas também entre linguagens de programação, como é o caso de XML e JSON. Esses, apesar de seus benefícios, acarretam em prejuízo no tráfego de informação, uma vez que necessitam de uma alta quantia de informações de controle em seus dados serializados.

Com os modelos de serialização visualizados, fez-se a decisão de implementar algo que contemplasse o melhor de ambos os métodos, a interobabilidade entre sistemas e linguagens. Por essa razão, descreveu-se um protocolo binário com formato bem estruturado, com suporte para os seguintes tipos de dados:

- Inteiro positivo de 8 bits;
- Inteiro positivo de 16 bits;
- Inteiro positivo de 32 bits;
- Inteiro positivo de 64 bits;
- Inteiro de 8 bits;
- Inteiro de 16 bits;
- Inteiro 32 bits;

- Inteiro 64 bits;
- Caractere, no padrão ascii;
- Valores booleanos;
- Float, de 32 bits;
- Double, de 64 bits;
- Lista de caracteres, também conhecido como String;
- Lista de qualquer um dos tipos primitivos citados acima.

Para cada um dos tipos de dados, criou-se um método de serialização específico, com o propósito de disponibilizar uma representação compacta e portável entre diversos sistemas. Os tipos primitivos mais simples, como o conjunto dos números inteiros positivos, simplesmente foram convertidos para binário, para os inteiros sinalizados, fora utilizado o padrão de complemento de dois. Essas mesmas regras, simples, aplicam-se a conversão de caractere e valores binários.

Para os tipos de ponto flutuante simples e duplo, *Float* e *Double*, respectivamente, utilizou-se os padrões especificados pelo IEEE 754 [Ins85]. Além disso, o tipo lista, que contempla os demais tipos, assim como um conjunto de caracteres, trata-se da concatenação do tamanho das informações, de no máximo 32 bits, em conjunto com a quantidade de dados em sequência.

A serialização dar-se-á por meio de uma estrutura de dados, como *struct* ou *class* em C++, contendo propriedades restritas aos tipos primitivos. Dessa maneira, essa estrutura de dados terá como propriedade uma outra, cujo objetivo é disponibilizar a manipulação dos metadados[1] da estrutura principal, disponibilizando o que será nomeado como "contrato". O contrato pode ser entendido como uma inteligência embarcada para reconhecimento das informações dispostas na estrutura, bem como a sua interpretação e referência a informação original.

Na Figura 5.1 é possível verificar como o contrato utiliza estruturas da biblioteca de modelos padrão, STL de C++, para referenciar os tipos primitivos de uma estrutura. A estrutura contempla uma lista com as *String* de cada uma das propriedades da *Classe*, bem como duas unidades de contêiner que armazenam elementos em pares chave-valor. A primeira dessas unidades contempla a *String* com a chave da propriedade e o valor contendo uma informação numérica referente a qual tipo a informação pertence. Por último, a outra unidade é formada também com a chave do tipo *String* com o nome da propriedade e valor sendo o ponteiro que referencia à informação primitiva na estrutura de dados. A partir disso, é possível serializar todas as informações apenas com os metadados.

[1]Metadados, ou Metainformação, são dados sobre outros dados. Um item de um metadado pode dizer do que se trata aquele dado, geralmente uma informação inteligível por um computador [Wik22b]

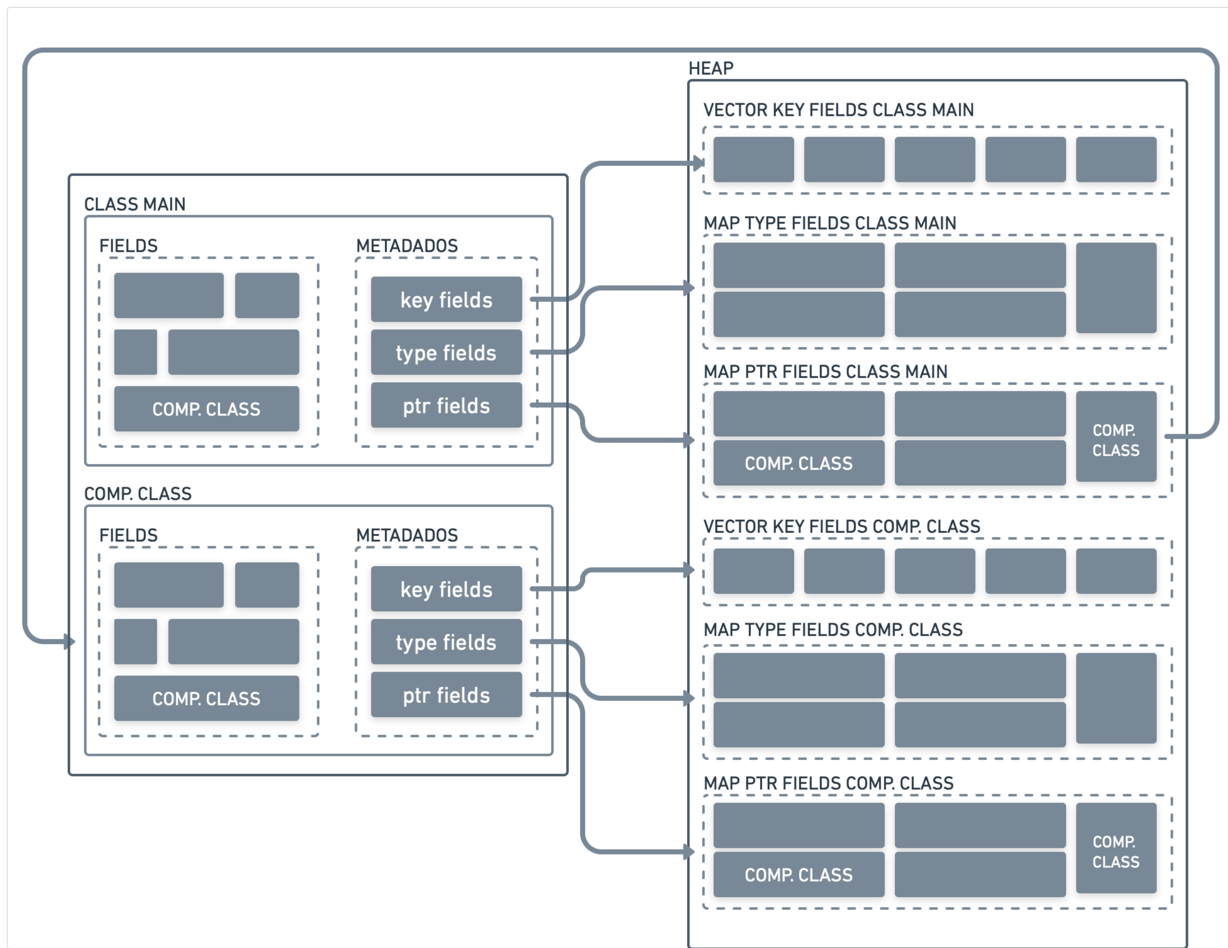


Figura 5.1: Formato dos metadados de uma estrutura
Fonte: O Autor

Para serializar, é realizado uma iteração por todos os nomes de propriedades da *struct*. Em posse do nome da estrutura, obtém-se o tipo e a referência na memória do *field*. Por fim, para cada um é executado dinamicamente a função de serialização ou desserialização, conforme seu tipo.

5.1.1 Método de utilização e descrição do funcionamento

Ao considerar um objeto denominado *People*, constituído de três propriedades: *age*, *name* e a referência a outro objeto igual, cujo nome é *father*, a instanciação do objeto faz-se por meio da declaração do trecho de código 5.1. Para exemplificar um caso de uso da serialização construída, é realizado a construção de dois objetos, um denominado *son* e outro *father*.

Para o primeiro objeto, *father*, é inicializado com as propriedades *name* e *age*, resultando na seguinte construção na memória, conforme Figura 5.2, ao qual é possível

```
1  class Person {
2      public:
3          struct metadatas meta;
4          Person(uint8_t age, std::string name, Person &father);
5          Person(uint8_t age, std::string name);
6          Person();
7          Person(Person& father);
8
9          std::string serialize();
10         void deserialize(std::string data);
11
12         uint8_t age;
13         std::string name;
14         Person* father;
15 };
```

Algoritmo 5.1: Declaração da classe People. Fonte: Autor

observar o funcionamento da estrutura de controle. Nessa estrutura, é mantido as *string* com os nomes das propriedades, *age* e *name*, bem como de qual tipo pertencem e a referência ao valor no objeto.

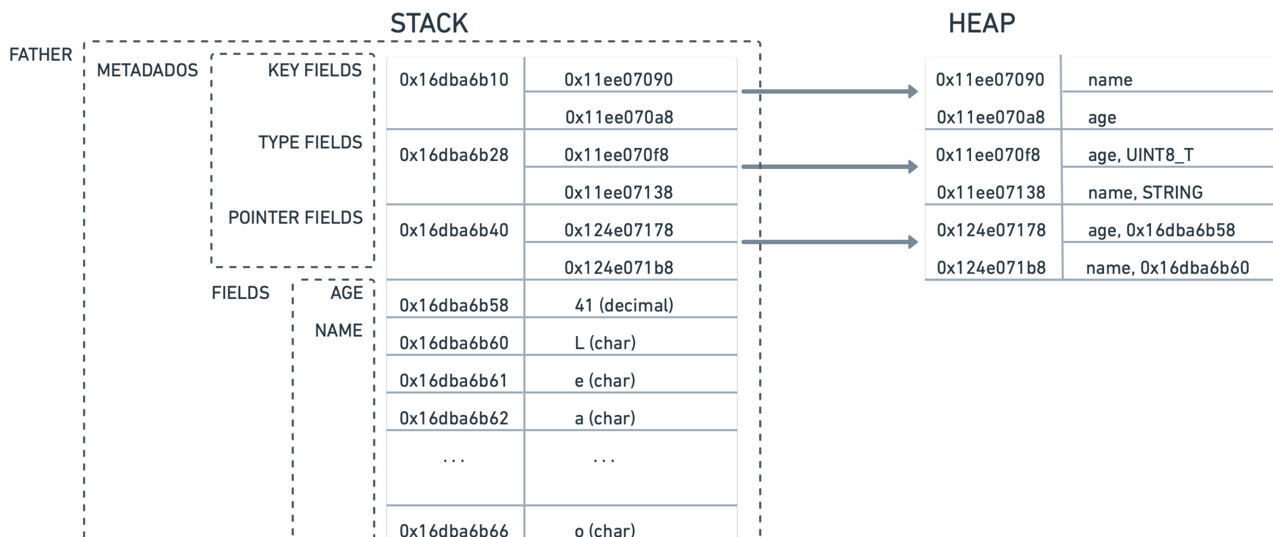


Figura 5.2: Estrutura na memória da construção do objeto *father*
Fonte: O Autor

Além desse objeto simples, é construído outro do mesmo tipo, porém com declaração da propriedade *father*. Nesse, no entanto, assim como o descrito na Figura 5.2, é informado os valores de *age* e *name*. Todavia, além dessas, faz-se referência para o primeiro objeto criado, denominado *father*, com a finalidade de relacionar esse segundo objeto com o primeiro, como uma relação familiar.

Dessa maneira, a disposição na memória desse segundo objeto é semelhate, com poucos acréscimos de informação, como é possível visualizar na Figura 5.3. Através do encadeamento das estruturas na memória, construído pela informação de controle, *ptr_fields*,

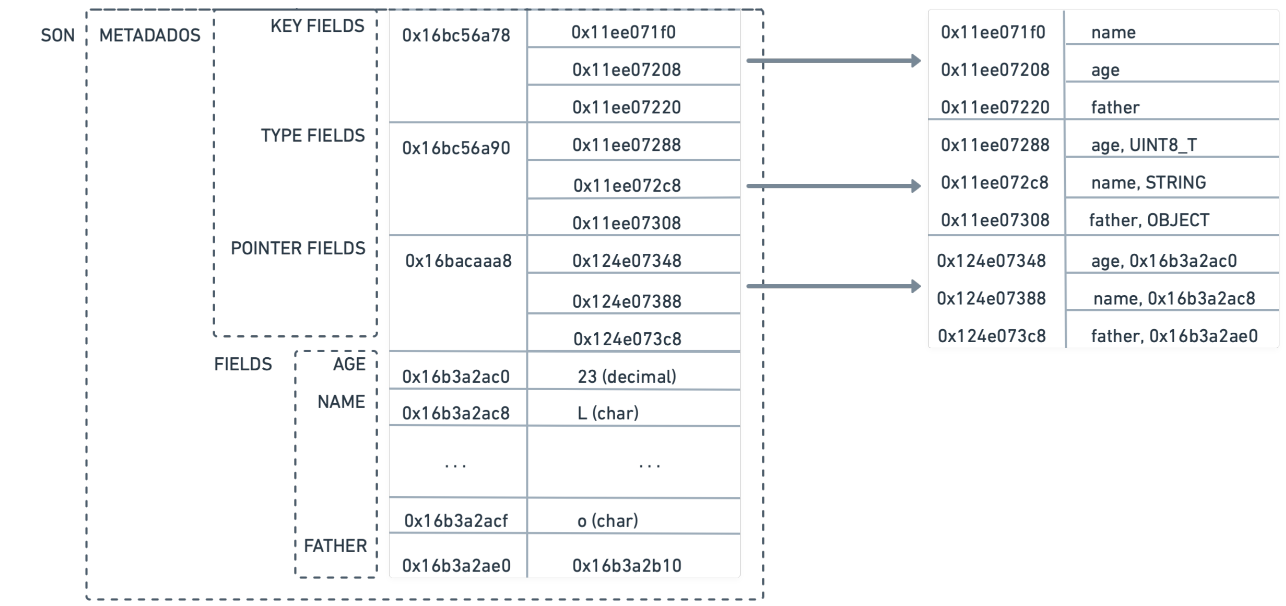


Figura 5.3: Estrutura na memória da construção do objeto *son*
Fonte: O Autor

```
1  #include "people.hpp"
2
3  int main(int argc, char** argv){
4
5      People father(43, "Leandro");
6      People son(23, "Leonardo", &father);
7
8  }
```

Algoritmo 5.2: Inicialização dos objetos *father* e *son*. Fonte: Autor

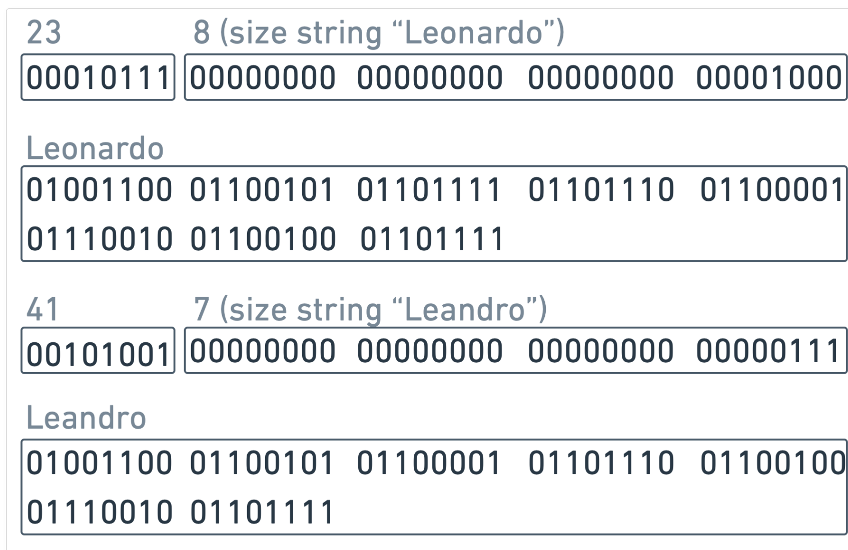


Figura 5.4: Resultado da serialização
Fonte: O Autor

que relaciona o nome da propriedade com o espaço de memória em que essa se encontra, é possível iterar as propriedades do objeto de exemplo, denominado *father*, como é visível na Figura 5.4.

5.2 Biblioteca de transporte

Com o propósito de abstrair o tratamento das informações trafegadas entre as partes envolvidas na biblioteca RPC, utilizou-se UDP para a biblioteca de transporte de informações. Fora escolhido o protocolo de transporte orientado à datagrama para o presente trabalho, pois, ao contrário desse, o protocolo orientado à conexão não possui tempo de retransmissão definido, assim como não inclui nenhum mecanismo para explorar o paralelismo no envio das informações. Com isso, a utilização do UDP nesse trabalho permitirá a configuração de *timeout*, bem como a implementação de métodos para explorar o paralelismo e o tratamento das informações não recebidas.

O protocolo de rede UDP é um protocolo da camada de transporte na estrutura do sistema TCP/IP. Uma vez que se trata de um protocolo sem conexão, isto é, não existe conexão entre o dispositivo de origem e o dispositivo de destino, ocasionando em um serviço de transmissão não confiável [LSS+15]. Em contrapartida, a característica da abstenção do controle do envio na camada de transporte, ocasiona uma alta taxa de transmissão de informações, por essa razão esse fora o protocolo selecionado para a implementação da biblioteca.

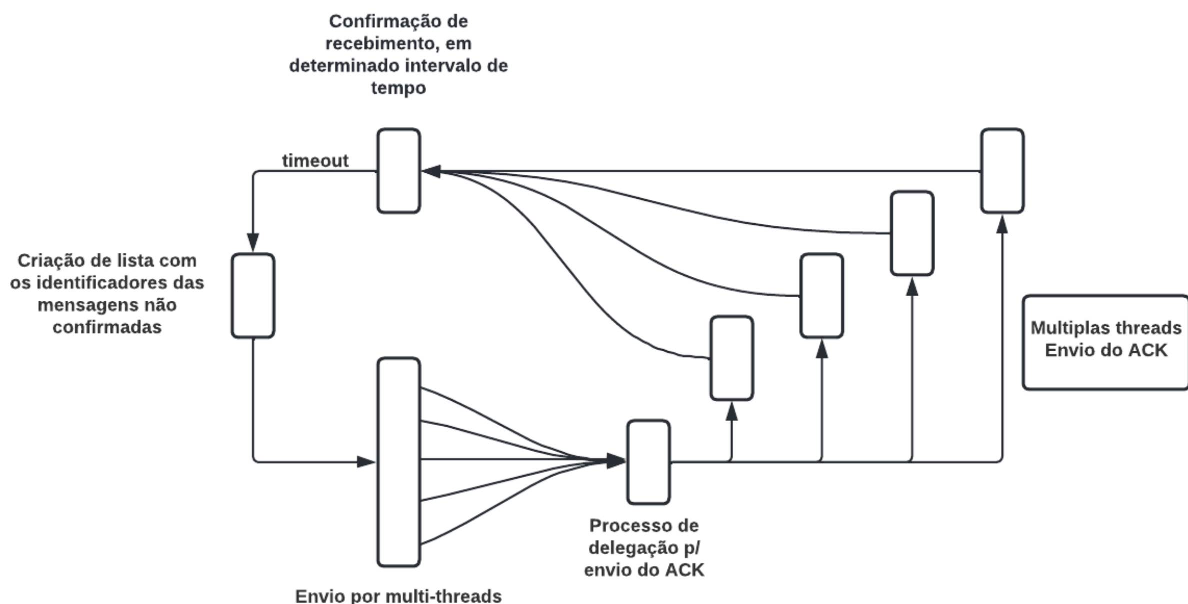


Figura 5.5: Diagrama do macro funcionamento da biblioteca de transporte
Fonte: O Autor.

Uma vez que a gestão do tráfego das informações não ficou sob responsabilidade da camada de transporte, fez-se necessário esta implementação no nível de aplicação. Dessa maneira, a implementação determinada funciona de maneira a aproveitar a capacidade de envio em paralelo do processador, permitindo que as informações sejam enviadas de forma desordenada, paralelamente. Conforme Figura 5.5, cujo propósito é apresentar, de maneira macro, a implementação da biblioteca, é possível visualizar que o cliente, aquele que envia as informações, dispõe de um número limitado de *threads*[2]. O servidor - que recebe o tráfego - não dispõe de múltiplas *threads* para gerenciar o recebimento, mas sim uma única cujo funcionamento é o mais simplório possível, bastando receber a informação e abrir uma nova tarefa concorrente para tratar do recebimento.

Para cada tarefa concorrente iniciada pelo servidor, ocorre o tratamento das informações, validação do trecho recebido e envio de uma mensagem de confirmação ao cliente. O cliente, por sua vez, possui uma outra *thread* em funcionamento para o recebimento das mensagens de confirmação. No caso do não recebimento de determinadas mensagens, por um período de tempo configurável, o cliente agrupa àquelas não enviadas e refaz o processo de envio, até que todos os trechos sejam recebidos com sucesso.

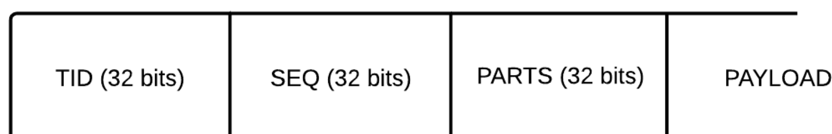


Figura 5.6: Serialização do *payload* de envio
Fonte: O Autor

Com o propósito de possibilitar a gestão do tráfego, foi anexado um cabeçalho de 12 *bytes* para cada envio. Essas informações, assim como descrito na Figura 5.6, tratam-se, respectivamente, de um número aleatório gerado para identificação da mensagem; o número de sequência, cuja finalidade é indicar sobre qual trecho é a mensagem; o número de partes que a mensagem. Logo após essas informações de controle, tem-se anexado a informação.

```
1 struct data_s {
2     uint32_t tid;
3     uint32_t seq;
4     uint32_t parts;
5 };
```

Algoritmo 5.3: Estrutura de controle de tráfego. Fonte: Autor

[2]Thread é uma forma como um processo/tarefa de um programa de computador é divido em duas ou mais tarefas que podem ser executadas concorrentemente.

No Algoritmo 5.3 é possível visualizar a criação da estrutura de dados supracitada, com o propósito do controle de fluxo das informações. Além disso, no Algoritmo 5.4, tem-se a utilização desse formato de gerenciamento, uma vez que se faz a criação de uma região de memória com o tamanho da informação somada ao tamanho do cabeçalho e, por fim, referencia-se o cabeçalho ao começo dessa região de memória, enquanto a informação é referenciada a posição inicial defasada com o tamanho do cabeçalho.

```
1 char buf[sizeof(struct data_s) + this->payload_size];
2 struct data_s *ptr = (struct data_s *)buf;
3
4 ptr->parts = parts;
5 ptr->seq = i + send;
6 ptr->tid = random;
7
8 char *data = (char *)buf + sizeof(struct data_s);
```

Algoritmo 5.4: Concatenação do cabeçalho de controle e a informação trafegada. Fonte: Autor

A partir desses artifícios, possibilitou-se a utilização de uma alta abstração ao desenvolvedor ao utilizar uma biblioteca própria de desempenho considerável no envio de informações. Esse, por sua vez, precisa apenas informar a quantidade de *threads* que serão disponibilizadas para o envio paralelo das informações, o tamanho de cada trecho enviado e o tempo limite, em milissegundos, da resposta do servidor, além das informações óbvias como o IP e porta do servidor, conforme é possível visualizar no Trecho de Código 5.5. Por padrão, o tempo de espera limite é de 10 milissegundos, e a quantidade de *bytes* no *payload* UDP em cada envio é 512 [FS11], além desses 512 *bytes*, é adicionado ao *payload* as informação de cabeçalho UDP, IP e também dos níveis de enlace e físico.

```
1 #include <client.hpp>
2
3 int main(int argc, char** argv){
4     Client client(8888, "127.0.0.1", 16, 10, 512);
5     client.send("hello world");
6 }
```

Algoritmo 5.5: Exemplo de instânciação da classe cliente. Fonte: Autor

A escolha da arquitetura assíncrona e paralela originou-se por meio de pesquisas, como a de Torsten Hoefler e Andrew Lumsdaine, que demonstrou que modelos que permitam a comunicação e a computação sobrepostas, resultam em uma maior probabilidade de desempenho superior [HL08]. Isso pois , por meio de interfaces de programação paralela, torna-se possível a melhor utilização da capacidade de *hardware*, otimizando a capacidade de envio de informações, e, consequentemente, diminuindo o tempo de processamento das

informações [HL08]. Para possibilitar a implementação dessa arquitetura, utilizou-se a API para programação multi-processo de memória compartilhada, OpenMP [CDK+01].

5.3 Algoritmo de compressão

Com o intuito de otimizar os recursos de rede oferecidos, já descritos, foi implementado o algoritmo de Codificação Aritmética, utilizando como referência a implementação original [WNC87]. Ao utilizar a compressão de maneira antecedente ao envio, é possível observar compressões relevantes, ocasionando, assim, a otimização de recursos de rede.

Não obstante, imprescindivelmente, fora realizado a alteração do algoritmo original para ao invés de realizar a codificação e decodificação sob o fluxo de entrada e saída, também conhecidos como *stdin* e *stdout*, fosse realizado sob cadeias de caracteres, como é possível visualizar nos trechos de código 5.7 e 5.6.

```
1  std::string decode(std::string str) {
2      start_model();
3      start_inputing_bits();
4      start_decoding(str);
5      std::string out;
6
7      for(;;){
8          int ch; int symbol;
9          symbol = decode_symbol(cum_freq);
10         if (symbol == EOF_symbol) break;
11         ch = index_to_char[symbol];
12         out += ch;
13         update_model(symbol);
14     }
15     return out;
16 }
```

Algoritmo 5.6: Alteração no método de decodificação. Fonte: Autor

5.4 Biblioteca de RPC

Por fim, com o objetivo de disponibilizar a abstração das funcionalidades fornecidas ao desenvolvedor, foi preparada uma biblioteca de RPC. Por meio dessa biblioteca, tornou-se possível a execução de chamadas remotas, com o cliente, aquele que solicita o processamento, apenas realizando a chamada do método de RPC, enquanto o servidor, ao receber, processa a requisição e devolve àquela que a solicitou.

```
1  std::string encode(std::string str){
2      start_model();
3      start_outputing_bits();
4      start_encoding();
5      for(auto i : str){
6          int ch; int symbol;
7          ch = i;
8
9          if (ch == EOF) break;
10         symbol = char_to_index[ch];
11         encode_symbol(symbol, cum_freq);
12         update_model(symbol);
13     }
14     encode_symbol(EOF_symbol, cum_freq);
15     done_encoding();
16     return done_outputing_bits();
17 }
```

Algoritmo 5.7: Alteração no método de decodificação. Fonte: Autor

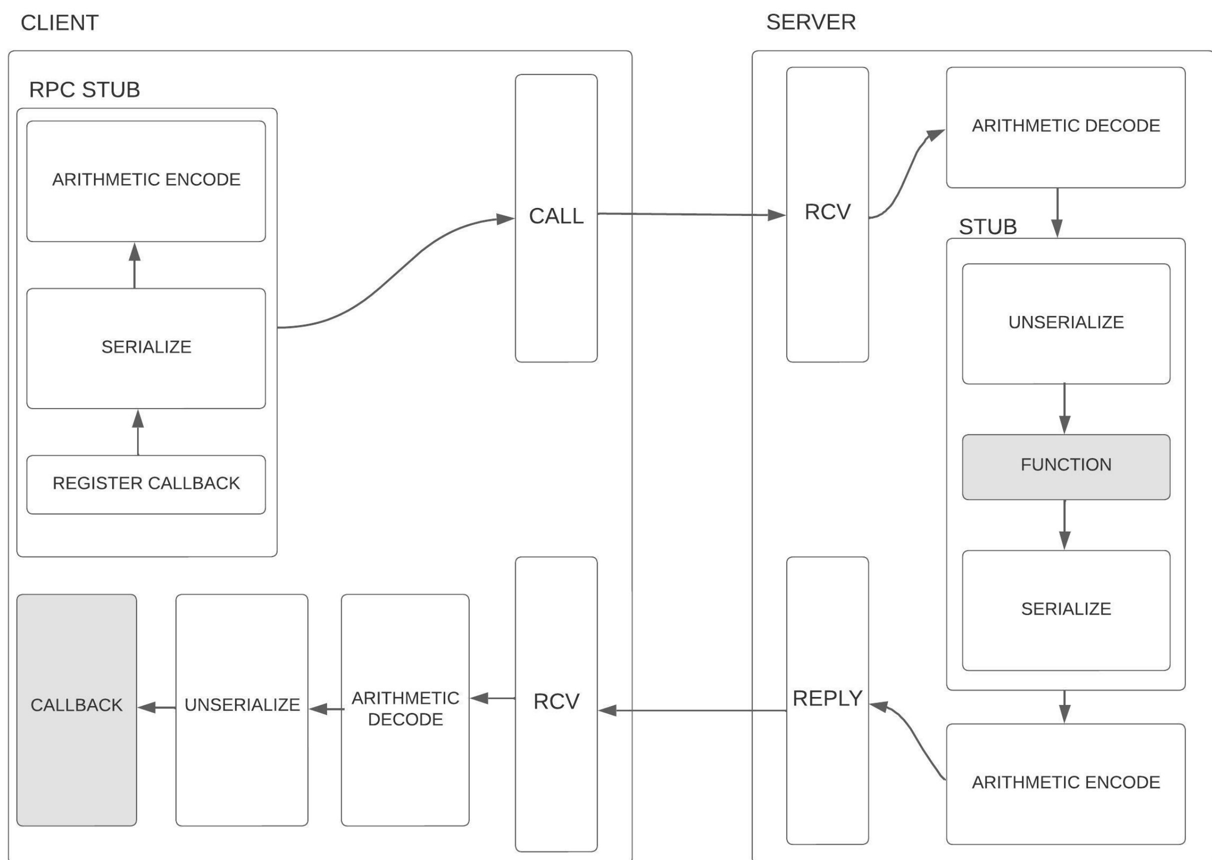


Figura 5.7: Diagrama de funcionamento da Biblioteca RPC
Fonte: O Autor

Na Figura 5.7, é possível visualizar o diagrama de funcionamento completo da biblioteca RPC desenvolvida, de modo que, explicitamente, o cliente realiza a chamada do

procedimento remoto, informando os parâmetros de envio e uma função para ser executada no momento do retorno da chamada assíncrona. Ao efetuar esta chamada, é realizado o registro da função *callback*, para que seja possível distinguir quais funções devem ser executadas perante determinados retornos. Após isso, faz-se a serialização dos parâmetros de requisição, a decodificação por meio do algoritmo de Codificação Aritmética, como visto na Seção 5.3. Apesar disso, ressalta-se que a utilização da Codificação Aritmética é configurável, por meio de um parâmetro na construção da biblioteca de RPC. Por fim, por meio da Biblioteca de Transporte, vista na Seção 5.2, faz-se o envio da *payload* ao cliente.

A aplicação servidora, ao receber a solicitação de procedimento remoto, recebe as informações por meio da Biblioteca de Transporte, realiza a decodificação da compressão aritmética, caso houver, e executa um método genérico, cujo propósito é executar a desserialização das informações e ajustar o formato do *payload* para então, processar a requisição da função registrada no servidor. O resultado do processamento passará pelo processo inverso, primeiro a serialização das informações e posterior compressão, caso houver. Por fim, realiza o envio do processamento ao cliente.

A aplicação no lado do cliente, ao receber a requisição na *thread* assíncrona, realiza a decodificação das informações. Em seguida, realiza-se a desserialização e o *casting*[3] das informações para o formato esperado no recebimento do método registrado como *callback*, para então, executá-las.

Figura 5.8: Representação do *payload* final enviado ao servidor
Fonte: O Autor

Além disso, houve a necessidade da inclusão de novas informações no dado trafegado entre as partes. Conforme Figura 5.8, percebe-se a inclusão de duas novas informações, o valor inteiro não sinalizado de 16 bits, para a representação da porta do cliente a ser devolvido o processamento da chamada remota, assim como um identificador criado pelo desenvolvedor da função remota executada, em díade com a rotina a ser invocada no retorno da chamada. Não obstante, diferente da Figura 5.8, a resposta da aplicação servidora ao cliente, não trás a necessidade da informação de 16 bits para localizar a porta, uma vez que o cliente não efetuará nenhuma outra solicitação remota a este servidor, mantendo o identificador da chamada finalizada, mas sim, como representado na Figura 5.9. Isso é requerido, pois o cliente necessita da identificação da função remota a ser executada, para que dessa forma, consiga informá-la na chamada ao servidor, bem como identificar o seu retorno na resposta.

[3]Casting é uma transformação aplicada em valores numéricos para modificar seu tipo de dado.

Figura 5.9: Representação do *payload* final enviado ao cliente
Fonte: O Autor

Por fim, para que o desenvolvedor consiga utilizar a biblioteca de chamada remota, é necessária apenas a inclusão da biblioteca do RPC, e a descrição do método a ser executado, no servidor, como é possível verificar no Algoritmo 5.8. Na função principal, instância-se a classe responsável pelo RPC, informando a porta em que será executada a biblioteca de transporte e por fim, registra-se o método que será executado com um número de identificação única.

De maneira análoga, o trecho de código de cliente é diferente em poucos aspectos, como a instância do objeto de RPC do lado do cliente, ao invés do servidor e a chamada do método da classe RPC com mesmo nome do método cadastrado no servidor, ao qual, receberá um tipo genérico como primeiro parâmetro que deve possuir o padrão de classe descrita no tópico de serialização, e o segundo parâmetro será um ponteiro para um método *callback* que será executado quando houver o retorno assíncrono da função processada, conforme Algoritmo 5.9. Dessa maneira, o desenvolvedor possuirá todos os artifícios necessários para executar chamadas de procedimento remotos facilmente, com trecho de código em alto nível de abstração e com demasiada facilidade de execução e utilização.

```
1  #include "rpc_server.hpp"
2
3  Request sum(Request req){
4      Request res(req.str);
5      return res;
6  }
7
8  int main(int argc, char** argv){
9      RpcServer rpc(3000);
10     rpc.registerFn(15, (void*)sum);
11 }
```

Algoritmo 5.8: Utilização da biblioteca de RPC no lado do servidor. Fonte: Autor

```
#include "rpc_client.hpp"

void sum(Request req){
    /*
        Callback method
    */
}

int main(int argc, char** argv){
    RpcClient client(3001);
    std::string str = "Hello World";
    Request req(str);
    rpc->sum(req, (void*)sum);
}
```

Algoritmo 5.9: Utilização da biblioteca de RPC no lado do cliente. Fonte: Autor

6. RESULTADOS

Nesse capítulo, abordar-se-ão testes realizados com o propósito de visualizar a capacidade de operação da biblioteca desenvolvida. Nestes referidos testes foram feitas variações no tamanho da informação trafegada, assim como no tamanho de cada unidade de envio, *datagram*. Além desses, também foram executados experimentos em ambientes que simulam uma conexão com variação no atraso da rede e perda de pacote entre as partes.

Em todos os testes foi utilizada a ferramenta CORE (*Common Open Research Emulator*) [ADHK08]. Cujo objetivo é construir redes virtuais, por meio de uma representação de rede de computadores reais que executam em tempo real.

6.1 Variação no tamanho do *payload* e *datagram*, sem uso de algoritmo de compressão

Com o propósito de testar a integridade da biblioteca de RPC desenvolvida, inicialmente criou-se uma estrutura de dados para serialização simples, cuja informação trafegada é um texto, ou *string*, de tamanho não fixo. A partir disso, foi submetido a testes com três intervalos de operação e de incremento no tamanho da informação, para simular distintos usos da biblioteca.

Para todos os casos de testes no tamanho da informação transportada, utilizou-se um intervalo de 256 *bytes* a 2048 *bytes*, com variação de 256 *bytes* no tamanho da informação enviada, *datagram*. Isso foi feito com o intuito de simular diferentes usos que um desenvolvedor pode realizar sob o transporte das informações, permitindo que visualize os diferentes efeitos e com isso, escolha as circunstâncias e os motivos para a configuração do tamanho do *datagram*.

6.1.1 *Payload* de 16 a 1024 *bytes*

No primeiro caso, simulando usos gerais de uma biblioteca RPC, foi submetido a testes do tempo de processamento e envio das informações em ambiente local, ou seja, sem a interferência de rede, com tamanho de informação de 16 *bytes* a 1024 *bytes*, com variação linear de 16 *bytes*. Esse teste encena utilizações comuns de uma biblioteca de chamada remota, com envio de poucos parâmetros e pouco volume de informações. É possível visualizar o resultado na Figura 6.1, ao qual o resultado demonstra que para informações de tamanho tão reduzidos e ainda sem influência de atraso de rede ou perda

de pacotes, o envio de *datagrams* da configuração padrão da biblioteca, de 512 *bytes* é suficiente para o bom funcionamento e processamento das informações.

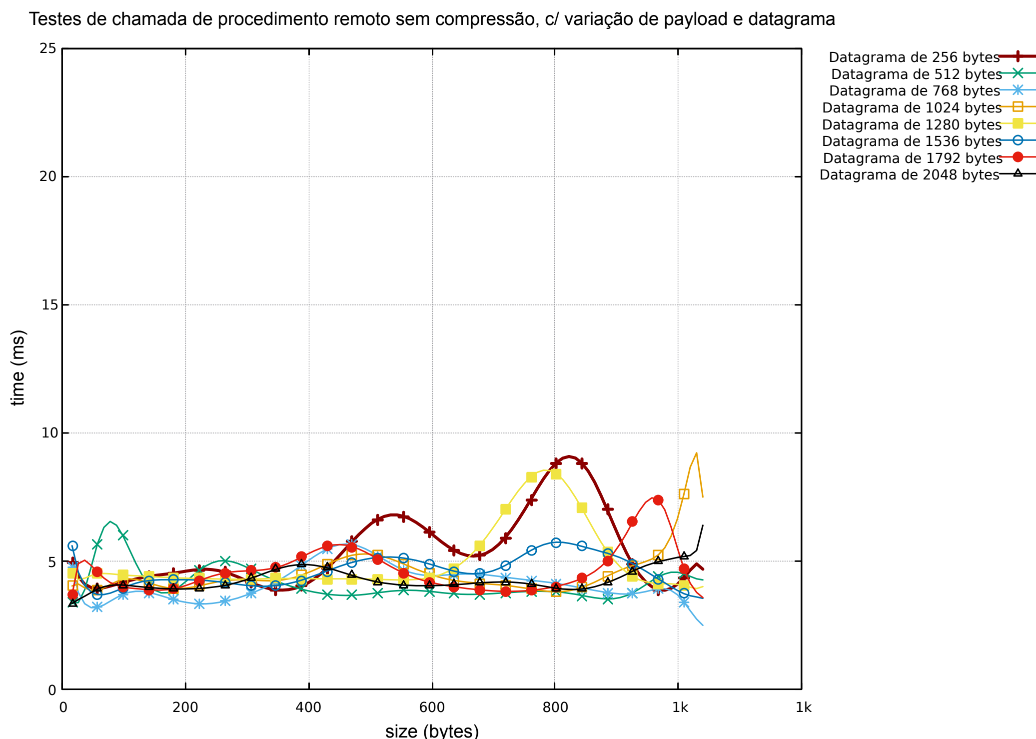


Figura 6.1: Teste com variação no tamanho do payload de 16 *bytes* a 1024 *bytes*, sem compressão de dados.

Fonte: O Autor

6.1.2 *Payload* de 1K *bytes* a 64K *bytes*

Em um segundo teste, foram executadas simulações com envio de informações de tamanhos superiores ao anterior, de 1024 *bytes* a 65.536 *bytes*, isto é de 1K a 64K *bytes*, com variação linear de 1K *bytes*. Esse experimento tem o intuito de validar usos da biblioteca com parâmetros estruturados ou pequenos trechos de textos. Nesse teste, conforme Figura 6.2, demonstra que a biblioteca estabiliza com tamanho de *datagram* superiores a 1024 *bytes*. Isso ocorre devido ao fato de que a biblioteca de transporte, descrita na Seção 5.2, funciona condicionada ao retorno de uma informação denominada como *ACK*, cujo objetivo é informar ao cliente que determinado envio foi recebido com sucesso, ou seja, as informações trafegadas com configurações de *datagram* inferiores a 1024 *bytes* necessitam de pelo menos 4 vezes mais retornos de confirmações, diminuindo a performance da execução, como é possível visualizar no teste realizado.

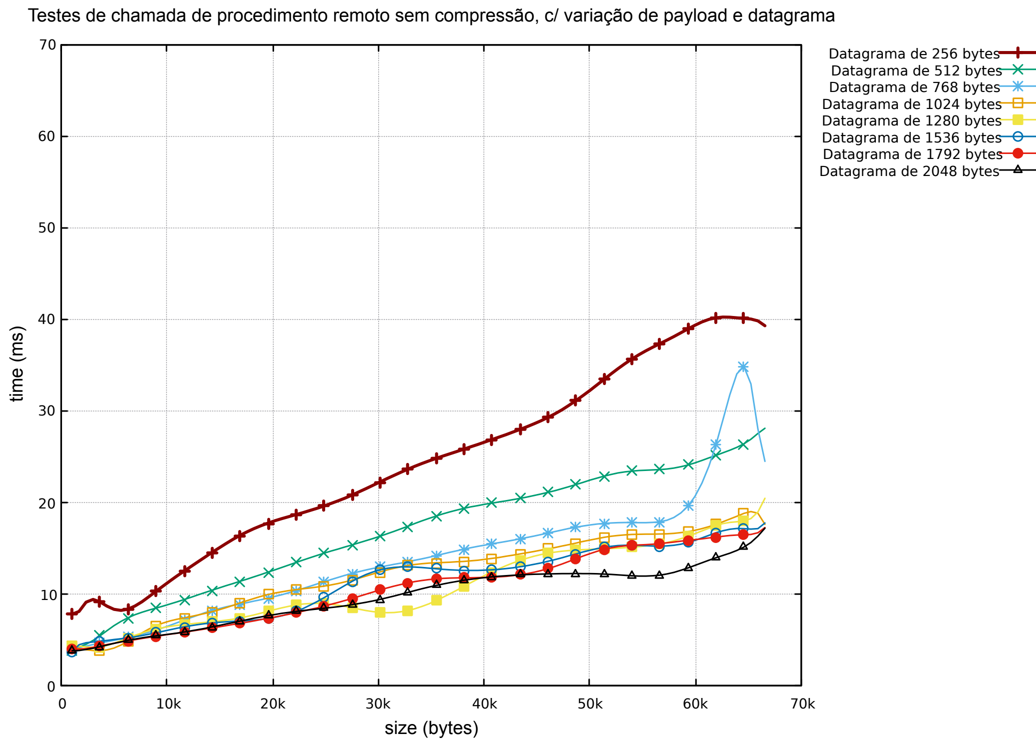


Figura 6.2: Teste com variação no tamanho do payload de 1K bytes a 64K bytes, sem compressão de dados.

Fonte: O Autor

6.1.3 *Payload* de 64K *bytes* a 2M *bytes*

No último teste de variação no tamanho da informação trafegada, simula casos de uso em que o desenvolvedor utiliza a biblioteca de chamada remota para tamanhos consideravelmente superiores, de 65.536 *bytes* a 2.097.152 *bytes*, isto é de 64K *bytes* a 2M *bytes*, com variação linear de 64K *bytes*. Essa simulação tem como propósito de figurar envio de informações binárias, como arquivos de texto, imagens e semelhantes.

Conforme Figura 6.3, verifica-se que a biblioteca comporta-se semelhantemente ao teste com informações de 1K a 64K *bytes*, isso porque o tempo de processamento para tamanhos de *datagram* superiores a 1024 *bytes* estabiliza o angulo de inclinação da relação linear entre o tamanho da informação transportada e o tempo total.

6.2 Variação do tamanho do *payload* e atraso de rede, sem uso de algoritmo de compressão

Em um próximo teste, com a intenção de validar a estrutura de paralelização da biblioteca, conforme descrito na Seção 3.7, submeteu-se a utilização em um ambiente de rede com condições de atraso de rede estáveis, por meio da ferramenta nativa dos sistemas

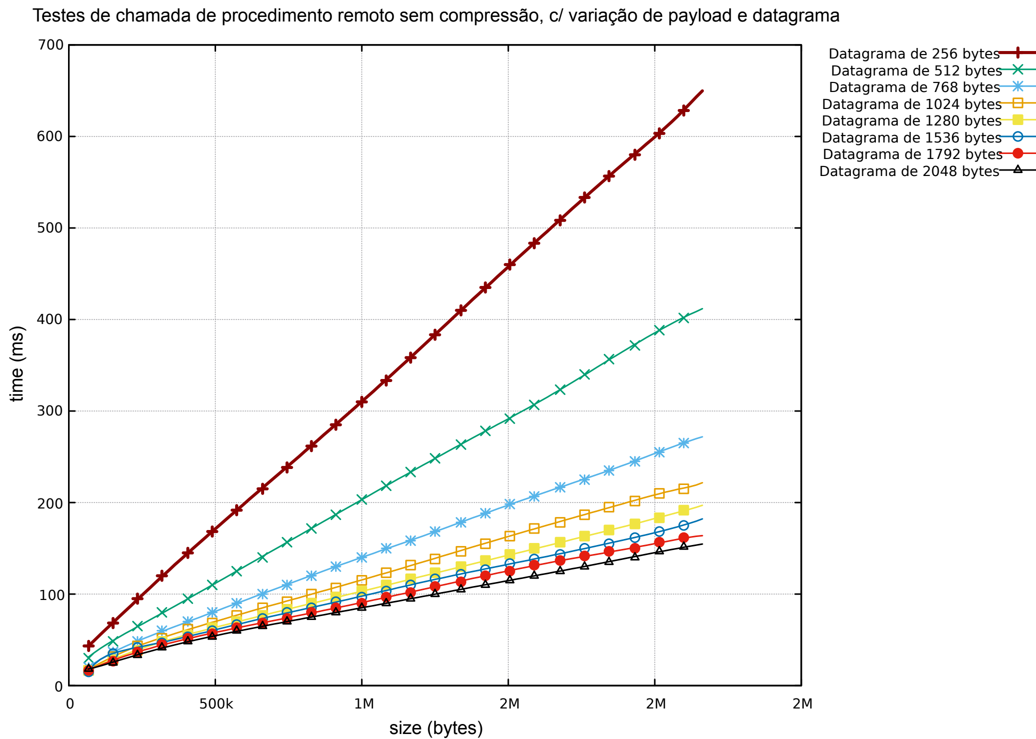


Figura 6.3: Teste com variação no tamanho do payload de 64K *bytes* a 2M *bytes*, sem compressão de dados.

Fonte: O Autor

Linux, chamada de *Traffic Controll* (tc), um utilitário que permite a configuração do agendador de pacotes do *kernel*. Utilizou-se dessa ferramenta para configurar, neste teste, atrasos no tempo de resposta na rede.

Para essa simulação, empregou-se intervalos de 1K *bytes* a 32K *bytes*, com variação linear de 1K *bytes*, assim como atrasos de 25ms a 200ms, também lineares, com intervalos de 25ms. Conforme Figura 6.4, é possível verificar o bom funcionamento da estrutura de paralelização, uma vez que mesmo com considerável atraso de rede o tempo de execução permanece estável, apenas com a adição do tempo de atraso, isso ocorre devido as janelas de envio paralelo, por padrão configurada com 16 envios paralelos, dado que o tamanho do *datagram*, neste teste, fixou-se em 512 *bytes*, ou seja 16 janelas de envio significam 8K *bytes* de envio paralelo, portanto é necessário 4 tempos de envio para uma informação de 32K *bytes*.

Uma vez que para cada tempo de envio, não é bloqueado a estrutura de envio, mas sim configurado um tempo de *timeout* para o recebimento assíncrono da confirmação do recebimento, resulta com que o atraso total não seja proporcional ao atraso de rede multiplicado pela quantidade de envio, mas sim próximo ao atraso de de rede acrescentado ao tempo de processamento das informações sem o atraso de rede, funcionamento este exemplificado na Figura 6.5.

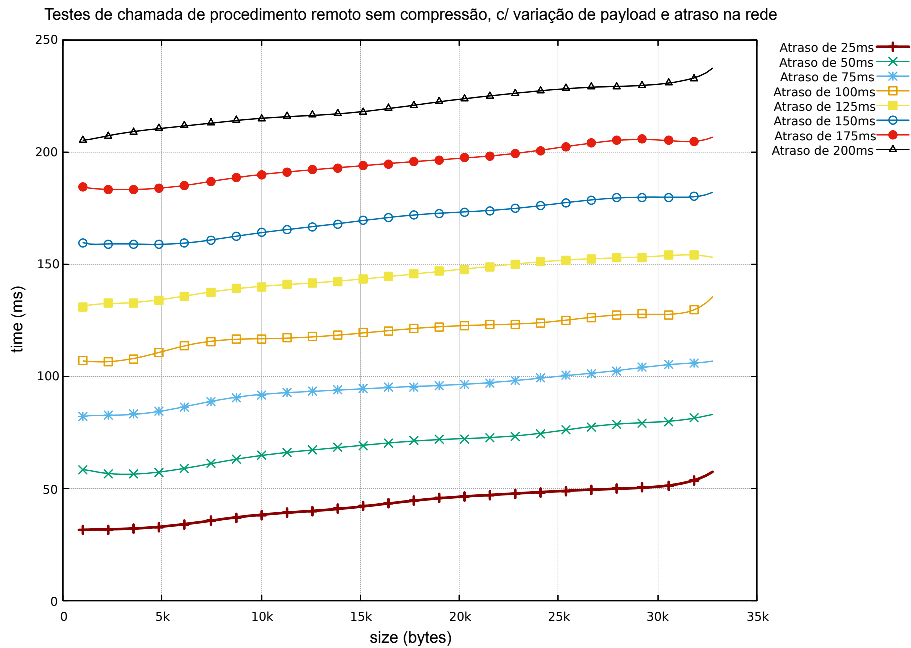


Figura 6.4: Teste com variação no atraso de rede, de 25ms a 200ms
Fonte: O Autor

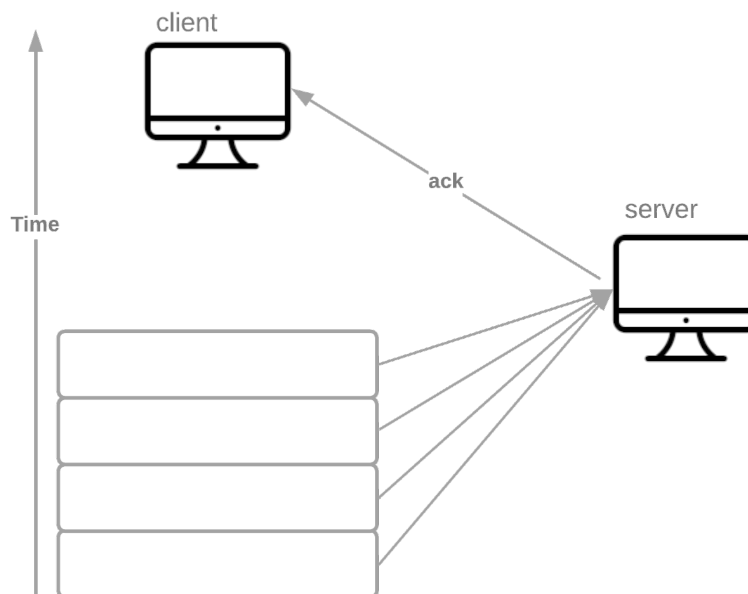


Figura 6.5: Representação do *payload* enviado ao servidor com paralelização
Fonte: O Autor

6.3 Variação do tamanho do *payload* e perda de pacote, sem uso de algoritmo de compressão

Com o objetivo de verificar a estrutura de reenvio de informações assíncronas, executou-se ensaios com ambientes configurados com percentuais de perdas de informa-

ções trafegadas. Assim como a avaliação anterior, empregou-se tamanhos de informações de 1K *bytes* a 32K *bytes* com variação constante de 1K *bytes*, porém acrescentou-se configurações de *loss* no utilitário TC de 3% a 24%, com variação linear de 3%.

A amostragem resultante, visível na Figura 6.6, demonstra que o funcionamento de reenvio de informações incluso na biblioteca de transporte, descrita na Seção 5.2 funcionou corretamente. Isso, pois quando o cliente envia um *datagram* e não recebe um mensagem com a confirmação do recebimento, denominada ACK, no tempo limite configurado na criação do objeto cliente, este realizará o reenvio do trecho de informação não confirmado pelo servidor. Dessa maneira, é possível visualizar que conforme o aumenta o percentual de perda de informações, resulta em um tempo maior para a finalização do processamento e envio das informações.

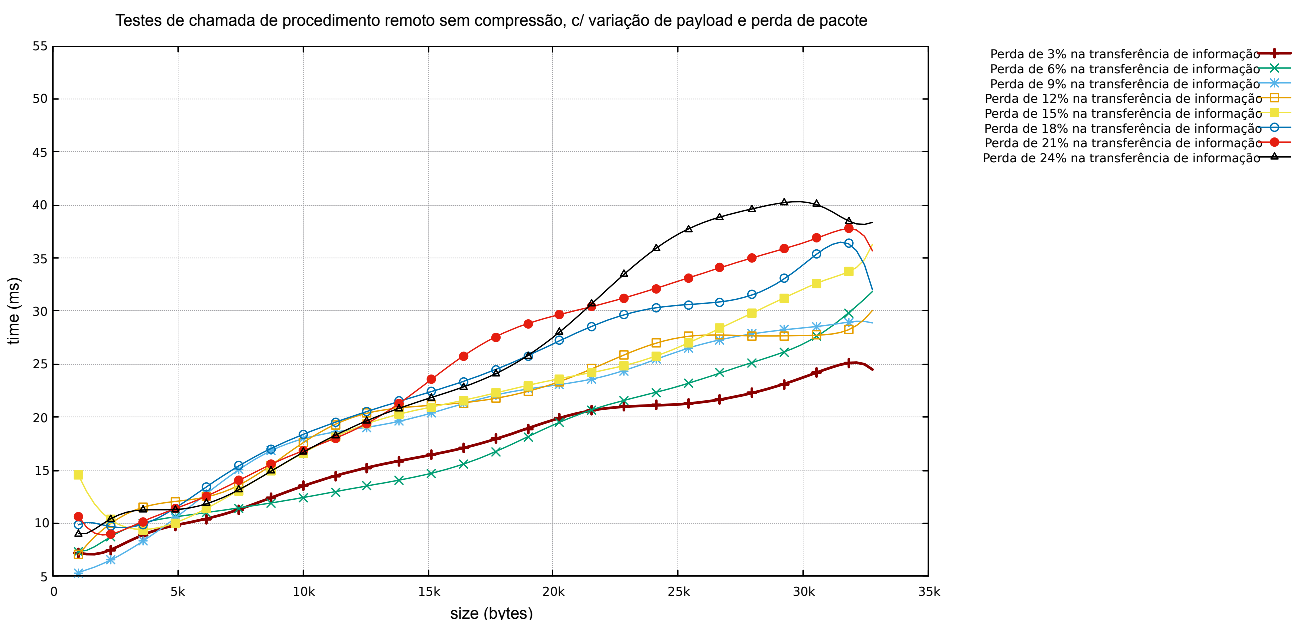
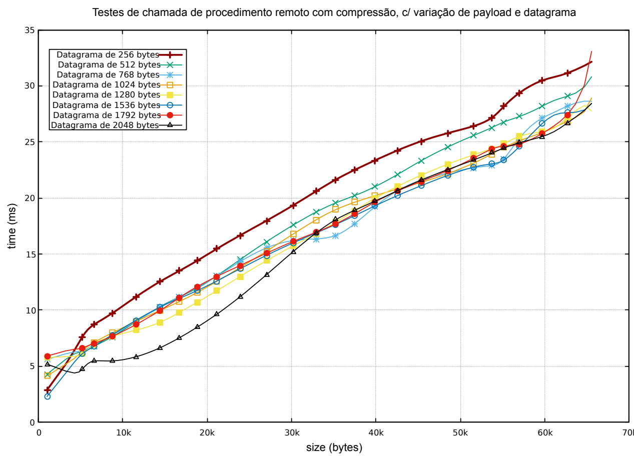


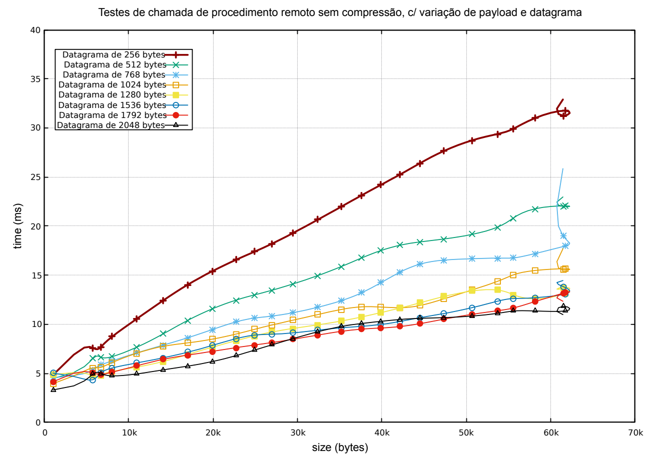
Figura 6.6: Teste com variação na perda de pacote, de 3% a 24%
Fonte: O Autor

6.4 Variação do tamanho do *payload* e tamanho do *datagram*, com uso de compressão

Por fim, com a finalidade de validar o funcionamento e utilidade do algoritmo de compressão incluído, conforme descrito na Seção 5.3, executou-se dois testes com variação no tamanho do *payload* de 1K *bytes* a 64K *bytes*, com variação linear de 1K *bytes*, assim como variação no tamanho do *datagram* de 256 *bytes* a 2048 *bytes*. A Figura 6.7a representa os resultados obtidos com o uso do método de compressão, enquanto a Figura 6.7b representa os resultados sem uso do algoritmo de compressão.



(a) Teste com variação no tamanho do *payload* e *datagram* com compressão



(b) Teste com variação no tamanho do *payload* e *datagram* sem compressão

Figura 6.7: Comparação entre testes com compressão e sem compressão

Por meio dessas amostragem, é possível verificar que o uso da codificação aritmética não acarreta em melhorias no desempenho no trafego de *payload* com alta entropia, de tamanhos de até 64K *bytes*, conforme teste realizado. No entanto, no caso de tamanhos de mensagens elevados, é indicado a realização de testes para verificar a possibilidade da utilização da compressão, isso pois haverá um tempo de compressão, no entanto, se a entropia da mensagem for baixa, ou seja, se ocasionar em um nível relevante de compressão, poderá ocasionar em tempos de trafego inferiores, devido ao tamanho reduzido da mensagem, dessa forma, compensando o tempo de processamento adicionado.

7. CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O desenvolvimento do presente estudo possibilitou uma compreensão profunda sobre o funcionamento de uma biblioteca de chamada de procedimentos remotos, bem como de grande parte das tecnologias utilizadas nesse âmbito, como a compressão, serialização, paralelização e a interface de programação disponibilizada ao usuário.

Ao realizar testes em ambientes com características distintas, sob a premissa da averiguação da integridade da biblioteca em modelos reais de utilização, tornou-se possível verificar que a construção da proposta deste trabalho foi concluída com sucesso. Isso, pois conforme o Capítulo 6 todos os testes realizados obtiveram resultados satisfatórios, demonstrando as diversas opções de utilização ao desenvolvedor tanto em ambientes locais quanto em aplicações que se comuniquem por meio da internet.

Quanto a desenvolvimentos futuros, alguns detalhes de otimização poderiam ser aperfeiçoados, principalmente por meio da paralelização já utilizada em diversos processamentos realizados na biblioteca. Além dessa evolução, poderia ser considerada a construção de um pré-processador, utilizando Lex e Yacc [LS75], para disponibilizar a interface de utilização para o desenvolvedor totalmente pronta, sem que seja necessário a sua configuração e a escrita de código repetitivo, como na versão atual desse trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[ADHK08] Ahrenholz, J.; Danilov, C.; Henderson, T. R.; Kim, J. H. “Core: A real-time network emulator”. In: MILCOM 2008 - 2008 IEEE Military Communications Conference, 2008, pp. 1–7.

[Alr21] Alrawais, A. “Parallel programming models and paradigms: Openmp analysis”. In: 2021 5th International Conference on Computing Methodologies and Communication (ICCMC), 2021, pp. 1022–1029.

[CDK+01] Chandra, R.; Dagum, L.; Kohr, D.; Menon, R.; Maydan, D.; McDonald, J. “Parallel programming in OpenMP”. Morgan kaufmann, 2001.

[Cla98] Clark, D. “Openmp: a parallel standard for the masses”, *IEEE Concurrency*, vol. 6–1, 1998, pp. 10–12.

[dEdS22] das Empresas de Software(ABES), A. B. “Estudo mercado brasileiro de software - panorama e tendências 2022”. [Online; accessed 19-Outubro-2022], Capturado em: https://abes.com.br/dados-do-setor/, 2022.

[DZ83] Day, J. D.; Zimmermann, H. “The osi reference model”, *Proceedings of the IEEE*, vol. 71–12, 1983, pp. 1334–1340.

[Fig21] Figueiredo. “Computação distribuída”. Capturado em: https://www.cos.ufrj.br/~daniel/sd/slides/aula_10.pdf, Ago 2022.

[FS11] Fall, K. R.; Stevens, W. R. “TCP/IP illustrated, volume 1: The protocols”. addison-Wesley, 2011.

[Gal03] Gallo, M. A. “Comunicação entre computadores e tecnologias de rede”. Cengage Learning Editores, 2003.

[grp19] “Adopting grpc at spotify”. Capturado em: https://www.jfokus.se/jfokus19-preso/Adopting-gRPC-at-Spotify.pdf, Ago 2022.

[HL08] Hoefler, T.; Lumsdaine, A. “Message progression in parallel computing-to thread or not to thread?” In: 2008 IEEE International Conference on Cluster Computing, 2008, pp. 213–222.

[IG20] Ivanov, A.; Guba, O. “How we migrated dropbox from nginx to envoy”. Capturado em: https://dropbox.tech/infrastructure/how-we-migrated-dropbox-from-nginx-to-envoy, Ago 2022.

[Inc12] Inc, G. “Protocol buffers”. Capturado em: https://en.wikipedia.org/wiki/Protocol_Buffers, Ago 2022.

[Inc19] Inc, U. T. “The architecture of uber’s api gateway”. Capturado em: https://eng.uber.com/architecture-api-gateway, Ago 2022.

[Inc22] Inc, G. “Documentation | grpc”. Capturado em: https://grpc.io/docs/, Ago 2022.

[Ins85] Institute of Electrical and Electronics Engineers. “Ieee standard for binary floating-point arithmetic”, 1985.

[JBF+05] Jacob, B.; Brown, M.; Fukui, K.; Trivedi, N.; et al.. “Introduction to grid computing”, *IBM redbooks*, 2005, pp. 3–6.

[LGH19] Lin, S.-J.; Gao, Z.; Han, Y. S. “Arithmetic coding based on reflected binary codes”. In: 2019 Ninth International Workshop on Signal Design and its Applications in Communications (IWSDA), 2019, pp. 1–5.

[LS75] Lesk, M. E.; Schmidt, E. “Lex: A lexical analyzer generator”. Bell Laboratories Murray Hill, NJ, 1975.

[LSS+15] Lin, H.; Su, H.; Shu, Z.; Wu, Z.-G.; Xu, Y. “Optimal estimation in udp-like networked control systems with intermittent inputs: Stability analysis and suboptimal filter design”, *IEEE Transactions on Automatic Control*, vol. 61–7, 2015, pp. 1794–1809.

[Mac80] Mackenzie, C. E. “Coded-Character Sets: History and Development”. Addison-Wesley Longman Publishing Co., Inc., 1980.

[MVdCdM16] Martins, N.; Vidal, E. L.; de Christo, N. T. S.; de Mello, A. V. “Comparativo de linguagens de programação”, *Anais do Salão Internacional de Ensino, Pesquisa e Extensão*, vol. 8–2, 2016.

[NK99] Namgoong, H.; Kim, M.-J. “Improved hard acknowledgement deadline protocol in real-time rpc”. In: Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium, 1999, pp. 375–382.

[PDF13] Prodanov, C. C.; De Freitas, E. C. “Metodologia do trabalho científico: métodos e técnicas da pesquisa e do trabalho acadêmico-2ª Edição”. Editora Feevale, 2013.

[rfc80] “User Datagram Protocol”. Capturado em: https://www.rfc-editor.org/info/rfc768, Ago 1980.

[SBM+17] Sterz, A.; Baumgärtner, L.; Mogk, R.; Mezini, M.; Freisleben, B. “Dtn-rpc: Remote procedure calls for disruption-tolerant networking”. In: 2017 IFIP Networking Conference (IFIP Networking) and Workshops, 2017, pp. 1–9.

[SNS+97] Sato, M.; Nakada, H.; Sekiguchi, S.; Matsuoka, S.; Nagashima, U.; Takagi, H. “Ninf: A network based information library for global world-wide computing infrastructure”. In: International Conference on High-Performance Computing and Networking, 1997, pp. 491–502.

[Str03] Stroustrup, B. “Abstraction, libraries, and efficiency in c+”, 2003.

[SYAD05] Seymour, K.; YarKhan, A.; Agrawal, S.; Dongarra, J. “Netsolve: Grid enabling scientific computing environments”, *Grid Computing and New Frontiers of High Performance Processing*, vol. 14, 2005, pp. 33–51.

[TA90] Tay, B. H.; Ananda, A. L. “A survey of remote procedure calls”, *ACM SIGOPS Operating Systems Review*, vol. 24–3, 1990, pp. 68–79.

[Tan03] Tanenbaum, A. S. “Redes de computadores. ed”, *Campus-Tradução da Terceira Edição, Rio de Janeiro*, 2003, pp. 55–120.

[Tan11] Tanenbaum, A. S. “Redes de computadores. ed”, *Campus-Tradução da Terceira Edição, Rio de Janeiro*, 2011, pp. 227–231.

[TNS+03] Tanaka, Y.; Nakada, H.; Sekiguchi, S.; Suzumura, T.; Matsuoka, S. “Ninf-g: A reference implementation of rpc-based programming middleware for grid computing”, *Journal of Grid computing*, vol. 1–1, 2003, pp. 41–51.

[Wik22a] Wikipédia. “Codificação aritmética”. [Online; accessed 8-Outubro-2022], Capturado em: https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_aritm%C3%A9tica, 2022.

[Wik22b] Wikipédia. “Metadados”. [Online; accessed 10-Novembro-2022], Capturado em: https://pt.wikipedia.org/wiki/Metadados, 2022.

[WNC87] Witten, I. H.; Neal, R. M.; Cleary, J. G. “Arithmetic coding for data compression”, *Communications of the ACM*, vol. 30–6, 1987, pp. 520–540.