UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
COLEGIADO DE CIÊNCIA DA COMPUTAÇÃO

Relatório do Trabalho – Redes de Computadores

Título: Implementação de Cliente e Servidor NTP
Alunos: Vitor Mayorca Camargo, Weberson Morelli Leite Júnior Data: 27/02/2025

---

1. INTRODUÇÃO

A internet surgiu na década de 1960 como ARPANET, inicialmente usada para comunicação militar e acadêmica nos EUA (LEINER, *et al*. 2009). Em 1989, a criação da World Wide Web democratizou o acesso e compartilhamento de informações por meio de navegadores web, permitindo que pessoas comuns usassem a internet de forma fácil e rápida (MONTEIRO, 2001).

As informações transmitidas na internet seguem protocolos rigorosos, como o TCP, o IP, e o UDP, que surgiram da necessidade de estabelecer regras e padrões para unificar as diferentes redes e máquinas que formam a internet (CORONA, 2004). O protocolo TCP (Transmission Control Protocol) foi projetado para fornecer uma comunicação confiável entre dois computadores, reduzindo ou aumentando a taxa de transmissão de dados de forma dinâmica, com o fim de evitar perda de dados. (CAMPISTA, *et al*. 2010).

Já o protocolo IPv4 (Internet Protocol) define a base para a transação, tráfego e reconhecimento de dados em uma rede. Ele é responsável por definir o "Endereço IP", que é um número único dado a cada máquina ou "host" na rede. Esse endereço é composto por quatro números (de 0 a 255) separados por pontos (de 000.000.000.000 a 255.255.255.255), e permite que uma máquina seja identificada e se comunique com outras na rede (CORONA, *et al*. 2004).

Duas versões do protocolo IP são usadas como padrão na internet. O IPv4 especifica as capacidades e protocolos básicos que toda máquina deve seguir, e usa um endereço de 32 bits (TANENBAUM, 2003). Já o IPv6, mais avançado, possui uma maior segurança e melhor comunicação entre as redes, fornecendo um endereço de 128 bits (CHANDRA, KATHING, KUMAR. 2013).

O protocolo UDP (User Datagram Protocol) é descrito na RFC 768, e possui como principal característica o envio de pacotes sem estabelecer uma conexão prévia, sendo muito rápido, mas sem garantia de que os pacotes chegarão em ordem (TANENBAUM, 2003). Por não ser orientado à conexão, ele acaba sendo um protocolo mais ágil, e é comumente utilizado em aplicações em tempo real. Porém, seu único mecanismo de controle de erros é o descarte de pacotes corrompidos (DOURADO, COSTA. 2008). Ele usa o protocolo IP para transportar mensagens. Porém, para distinguir entre diferentes destinos dentro de um host, ele usa o mecanismo de “portas”. Elas funcionam como uma interface, um multiplexador que gerencia o tráfego das informações do protocolo IP (DINIZ, JUNIOR. 2014).

Muitos outros protocolos se baseiam no funcionamento do protocolo UDP, sendo que um dos mais importantes é o protocolo NTP (Network Time Protocol). Segundo Utime (2021), o protocolo NTP foi desenvolvido para garantir a sincronização de tempo entre diferentes computadores e dispositivos numa ou mais redes. Pode-se dizer que esse protocolo é fundamental para o funcionamento de inúmeros dispositivos ao redor do mundo, já que a medição precisa de tempo é indispensável para diversas aplicações.

O protocolo NTP se coordena com o sistema UTC (Coordinated Universal Time) para

conseguir manter a sua precisão. Ele foi inicialmente idealizado por David Mills, na década de 70, e foi definido oficialmente pela primeira vez em 1985, pela RFC 958 (MILLS, 1985). Atualmente, o NTP está em sua versão 4.0, e é definido pela RFC 5905 (MILLS, *et al*. 2010). Essa NFC especifica por completo todos os detalhes da arquitetura, implementação, protocolos, estruturas de dados, e algoritmos utilizados pelo protocolo.

A arquitetura do protocolo NTP é formada por uma hierarquia, com as camadas (stratum) sendo numeradas de 0 a 16. Quanto maior o stratum, menor a precisão da máquina em média. Uma das características da arquitetura NTP é que um dispositivo pode, simultaneamente, ser um servidor primário, servidor secundário, e um cliente, fornecendo tempo para outros dispositivos e consultando o tempo para sincronizar o próprio relógio (SOARES; DANTAS, 2022).

Os diversos algoritmos definidos na RFC procuram minimizar a diferença de tempo e frequência entre o UTC e o relógio do sistema. Quando essas diferenças são reduzidas a um nível nominal, se diz que o relógio está sincronizado com o UTC. Para seu funcionamento, o protocolo NTP usa 3 tipos de dados próprios diferentes: o NTP Short (32 bit), o NTP Timestamp (64 bit), e o NTP Date (128 bit) (Figura 1). O formato de 128 bits é usado quando se há espaço de armazenamento de sobra, e seus primeiros 64 bits representam segundos (584 bilhões de anos), e cada unidade dos últimos 64 bits representa 0,5 attosegundos (0.5e-18 segundos) (MILLS, *et al*. 2010).

O Short e o Timestamp são usados em headers de pacotes, e funcionam de forma similar: a primeira metade representa segundos, e a segunda fração de segundos. O instante equivale às 00:00 de 1 de janeiro de 1900. Toda medida de tempo é relativa à esse instante 0. Eles também acontecem em “eras”: a era 0 vai do início de 1900 até algum momento em 2036, quando ocorre o wrap-around para a era 1. O mesmo pode ocorrer para o passado (MILLS, *et al*. 2010).

Figura 1: Tipos de dados NTP (Fonte: MILLS, *et al*. 2010).

O cabeçalho de um pacote NTP padrão está representado na Figura 2, e é composto por 16 campos, sendo os dois últimos opcionais, e ainda permitindo adição de dois campos de extensão caso necessário (MILLS, *et al*. 2010).

- Leap indicator sinaliza a adição/remoção de um segundo no último minuto do mês atual. Possui 2 bits;
- Version number é a versão do protocolo usada. Possui 3 bits;
- Mode é o modo de associação, que pode ser simétrico ativo ou passivo, cliente, servidor, broadcast, control, ou reservado. Possui 3 bits;
- Stratum é o nível hierárquico do servidor. Possui 8 bits;
- Poll representa o intervalo máximo entre as mensagens, em log2 segundos. Possui 8 bits;
- Precision é a precisão do clock do sistema. Possui 8 bits;
- Root delay é o delay do round-trip time ao clock de referência. É um NTP Short;
- Root dispersion é a dispersão do clock de referência. É um NTP Short;
- Reference id é um código que identifica um servidor particular ou, caso o stratum seja 0, armazena uma mensagem de debugging chamada “kiss code”. É uma string de 4 caracteres;
- Reference timestamp é o tempo onde o clock foi corrigido pela última vez; origin timestamp é o tempo quando o client enviou a requisição. É um NTP Timestamp;
- Receive timestamp é o tempo que o server recebeu a requisição. É um NTP Timestamp;
- Transmit timestamp é o tempo que o server enviou a resposta da requisição. É um NTP Timestamp;
- Destination timestamp é o tempo aonde a resposta chegou ao cliente (porém, esse campo não é normalmente incluído no header). É um NTP Timestamp;

Depois deles, podem ocorrer até dois campos de extensão, que contém uma mensagem de requisição ou resposta; Os últimos dois campos são usados apenas para fins criptográficos:

- Key identifier serve para designar uma chave armazenada pelo servidor. Possui 32 bits;
- Message Digest armazena um hash da mensagem inteira sem incluir o Key identifier nem o digest em si. Possui sempre 128 bits.

Name	Formula	Description
leap	leap	leap indicator (LI)
version	version	version number (VN)
mode	mode	mode
stratum	stratum	stratum
poll	poll	poll exponent
precision	rho	precision exponent
rootdelay	delta_r	root delay
rootdisp	epsilon_r	root dispersion
refid	refid	reference ID
reftime	reftime	reference timestamp
org	T1	origin timestamp
rec	T2	receive timestamp
xmt	T3	transmit timestamp
dst	T4	destination timestamp
keyid	keyid	key ID
dgst	dgst	message digest

Figura 2: Cabeçalho NTP (Fonte: MILLS, *et al*. 2010).

As trocas de mensagens NTP podem ser unicast (um para um) ou broadcast/multicast (um para muitos), utilizando endereços específicos para IPv4 e IPv6. O protocolo opera com quatro timestamps (T1 a T4) e três variáveis de estado (org, rec, xmt), permitindo a medição de atrasos e

offsets entre pares de máquinas. Esses quatro timestamps são trocados durante uma comunicação, e são usados para calcular o round-trip delay e o offset de uma máquina em relação à outra. A máquina A envia um pacote em T1, que é recebido por B em T2. B o processa, e envia a resposta em T3, que chega de volta à A em T4. A máquina A usa esses 4 timestamps para os cálculos, que estão representados na Figura 3. Essas variáveis estão no formato NTP Timestamp, e normalmente são convertidas para ponto flutuante antes da realização da operação. O valor final do offset pode ser utilizado para corrigir o relógio da máquina A, se esta aceditar que o relógio de B é mais preciso (MILLS, *et al*. 2010).

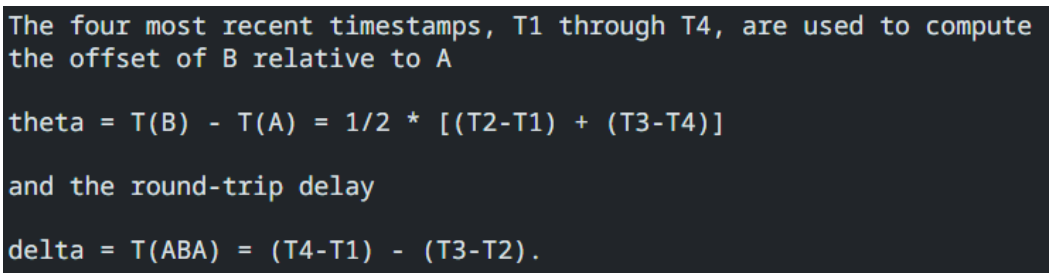

Figura 3: Cálculos de Offset e Round-Trip Delay (Fonte: MILLS, *et al*. 2010).

Também existe um algoritmo de filtragem de relógio no protocolo NTP. Nele, o peer basicamente vai fazer inúmeras classificações rigorosas entre as diferentes amostras de tempos obtidas. O algoritmo de seleção descarta servidores com tempos incorretos (falsetickers) e mantém os confiáveis (truechimers). O algoritmo de agrupamento elimina gradualmente os servidores mais imprecisos até restarem os melhores, e então calcula o tempo final com uso de uma média ponderada. O valor de tempo considerado é então definido como o padrão pelo sistema operacional, e a máquina passa a compartilhá-lo com outras máquinas na rede (MILLS, *et al*. 2010).

Existe também um processo automático de ajuste do relógio do sistema, que aplica uma correção percentual no relógio do sistema a cada X segundos. De tempos em tempos, ele então vai fazer uma requisição para se sincronizar a um servidor NTP, num processo chamado de polling. Conforme mencionado antes, no cabeçalho NTP é definido uma variável chamada “poll”. Ela basicamente define a frequência na qual será feito o polling ou, em outras palavras, a frequência na qual serão feitas requisições NTP a um servidor preciso e confiável. Se o servidor parar de responder, o tempo de polling aumentará gradativamente, e retornará ao padrao quando receber uma resposta válida (MILLS, *et al*. 2010).

Por fim, é essencial pontuar que existe uma necessidade crítica de segurança no protocolo NTP. A autenticação da veracidade e confiabilidade de um pacote ajuda a garantir um tempo correto e a evitar erros, que podem se espalhar rapidamente pela hierarquia de servidores. O uso de algoritmos de autenticação protege contra servidores maliciosos e pode ser reforçada com filtragem e criptografia, como o Autokey. No entanto, o uso de MD5 ainda persiste devido à ampla adoção na Internet (MILLS, *et al*. 2010).

Existem diversas formas de se criptografar dados que serão enviados pela internet, sendo a mais comum a criptografia simétrica. Oliveira (2012) explica que a criptografia simétrica é um método relativamente antigo, e nela a chave é a mesma para ambas as partes, e deve permanecer um segredo. É simples e rápida. Porém, a chave precisa ser previamente compartilhada entre origem e destino e, se ela for interceptada nesse momento, toda a comunicação poderá estar comprometida.

Mesmo assim, algoritmos de criptografia simétrica são muito utilizados para garantir a integridade de pacotes NTP compartilhados. Scarselli *et al* (2019) e Mkacher *et al* (2018) concordam que o algoritmo simétrico HMAC-SHA256 é um algoritmo robusto e eficaz quando usado para garantir a integridade de pacotes trocados via NTP. O fato de ser um algoritmo padronizado e facilmente disponibilizado por diversas implementações open source também o tornam mais atrativo para a tarefa.

Implementações do protocolo NTP também podem ser feitas utilizando a linguagem de programação *Python*. Segundo Neto (2007), *Python* é uma linguagem muito interessante para essa função por ser uma linguagem de altíssimo nível, com tipagem dinâmica, sintaxe simples, e por possuir inúmeras bibliotecas que facilitam o processo de desenvolvimento de diferentes sistemas de comunicação por rede.

2. OBJETIVOS

O objetivo do trabalho prático foi implementar um cliente e um servidor compatíveis com o protocolo Network Time Protocol (NTP). O sistema implementado precisou ser capaz de se comunicar com clientes e servidores NTP oficiais, respeitando o protocolo NTP na versão 3 ou superior. O foco esteve na construção manual das funcionalidades do NTP, sem o uso de bibliotecas específicas para NTP.

O cliente desenvolvido também precisou fornecer uma interface que permita informar o endereço do servidor e, na ausência deste parâmetro, utilizar um servidor oficial aleatório do NTP.br. O endereço do servidor deve ser informado ao usuário juntamente com a data e hora calculados. Também foi necessário seguir todos os requisitos especificados no documento de especificação do trabalho.

3. MATERIAL UTILIZADO

A implementação foi feita num notebook da marca DELL, modelo Vostro 3520, com um CPU Intel Core i7-1255U 1.70 GHz, 16Gb de memória RAM DDR4, placa de vídeo integrada Intel Iris Xe Graphics, placa de vídeo dedicada NVIDIA GeForce MX550, adaptador de rede modelo Intel(R) Wi-Fi 6 AX201 160MHz, unidade de disco SSD NVMe ADATA 512Gb.

Dentro dele, foi executado o sistema operacional Windows 11 Home, versão 10.0.22631. Mais especificações da máquina estão explícitas na figura 4. Na máquina foi instalado o interpretador da linguagem *python* diretamente do website oficial. Todos os pacotes e módulos utilizados no processo de desenvolvimento já vieram incluídos na instalação padrão do *python*.

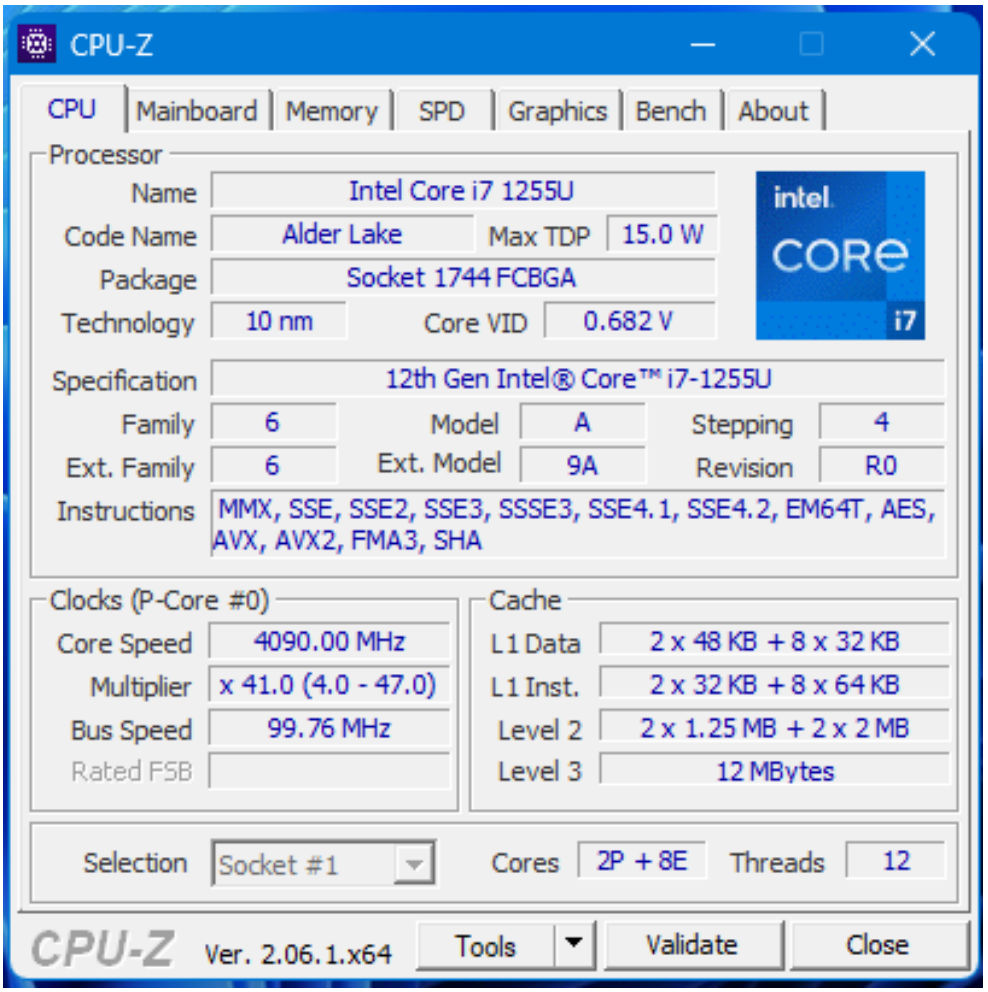

Figura 4: Especificações do sistema segundo o software *CPU-Z*.

4. METODOLOGIA

4.1 Entendimento do Protocolo NTP:

Antes que o processo de desenvolvimento da aplicação fosse iniciado, foi preciso estudar e entender as especificações do protocolo NTP definidas na RFC 5905 (MILLS, *et al*. 2010). Logo de cara percebeu-se que, devido à complexidade de uma implementação completa do protocolo NTP, decidiu-se omitir algumas partes não essenciais para o funcionamento básico do protocolo. Em relação ao cabeçalho, decidiu-se manter o campo Leap Indicator como “00”, a não ser que o servidor receba a requisição de um cliente com um valor diferente. Isso foi feito pois,para implementá-lo corretamente, aparentemente seria necessário consultar constantemente a tabela de leap seconds da IERS (International Earth Rotation and Reference Systems Service).

Os campos de extensão do cabeçalho estão disponíveis para serem utilizados. Porém não foi feita nenhuma função que cria pacotes com o uso deles pois, dentre as especificações do trabalho, nenhuma aparentou precisar do uso desses campos, já que as funções de criptografia fazem uso apenas dos campos de MAC (Key ID, e Message Digest).

Em relação às questões de autenticação e validação definidas pela RFC, decidiu-se implementá-las por meio da função de hashing HMAC-SHA256. Decidiu-se criar uma implementação que permite o uso de diversas chaves compartilhadas, sendo cada uma associada à um Key ID diferente. Porém, o processo de remoção ou adição de novas chaves é manual,

precisando ser feito dentro do próprio código.

Também não foi implementado um mecanismo de Slew (onde a troca do horário do relógio é feita gradativamente, e em pequenas partes). Além disso, também decidiu-se não implementar todos os processos e heurísticas utilizados no cálculo do Jitter e subsequente filtragem de pacotes considerados não confiáveis. Isso foi feito devido à grande complexidade desse processo, e devido ao fato de ele não ser necessariamente essencial para o funcionamento básico de uma implementação do protocolo.

Além disso, decidiu-se fazer uma implementação básica da parte de recebimento de pacotes "*Kiss of Death*". Porém, a parte do servidor não envia *Kisses of Death* para clientes que fazem requisições problemáticas, pois a definição de como e quando isso deveria ser feito não pareceu clara após a leitura da RFC. Por outro lado, o processo de polling foi implementado quase que por completo, pelo menos no contexto de requisições no modo client.

Fora esses ajustes, procurou-se implementar um sistema que fosse compatível com qualquer outra implementação do protocolo NTP. Inicialmente, a ideia era criar dois programas diferentes: um que age como cliente, e outro que age como servidor. Porém, a RFC explicitou que uma implementação do NTP tem que necessariamente poder agir como ambos. Logo, ambas as implementações foram condensadas num único programa.

O processo de implementação se deu em 5 etapas:

1. Implementação dos tipos de dados, aonde foram criadas as funções para converter dados do tipo double para NTP Timestamp e NTP Short, e vice-versa. Para isso, usou-se a biblioteca *math* e a biblioteca *time*.
2. Implementação da função responsável por receber diversas variáveis e montar um cabeçalho NTP válido. Essa função recebe como parâmetro todos os diversos elementos do cabeçalho, depois os converte para um formato binário por meio da biblioteca *struct*, concatena-os, e retorna o pacote completo.
3. Implementação da função de client, que recebe do usuário as diversas variáveis necessárias (como IP e porta do servidor de consulta), calcula o tempo atual, e utiliza a biblioteca *socket* para enviar os pacotes a qualquer servidor NTP válido. Também vale mencionar que, caso o usuário deseje, o sistema irá automaticamente atualizar o horário da máquina com base na resposta do servidor. Essa função está disponível para máquinas com os sistemas operacionais *Windows* e *Linux*.
4. Implementação da função de server, que também recebe alguns parâmetros do usuário e, com ajuda da biblioteca *socket*, fica aguardando o recebimento de requisições indefinidamente. Ao receber uma requisição, é chamada uma função que processa o pacote recebido, e cria um pacote novo com base no relógio do sistema.
5. Por fim, após a implementação ter sido testada e validada, foram adicionadas diversas funções de criptografia ao sistema. O cliente agora possui a possibilidade de enviar pacotes com um *key id* e uma chave definidas pelo próprio usuário antes do início do processo de polling. Já a interface do servidor irá apenas pedir se o servidor irá validar os pacotes ou não, e a validação já é feita automaticamente com base nos *key id's* e chaves armazenadas pelo algoritmo. Se houver alguma inconsistência entre ambos, o pacote é ignorado. Também foi alterada a função de criação de cabeçalhos: caso ela receba uma *chave* e um *key id*, também irá calcular o message digest por cima dos valores do cabeçalho (sem incluir o *key id* em si, conforme explicado pela RFC). Tudo isso foi feito com a ajuda da biblioteca *hashlib*.

Após cada etapa do processo de implementação, foram feitos inúmeros testes com servidores oficiais, com o objetivo de garantir que o sistema estivesse funcionando corretamente. Também vale mencionar que, para algumas funcionalidades mais complexas, foi necessário procurar ajuda de programadores mais experientes em fóruns online como o *StackOverflow*.

5. RESULTADOS E DISCUSSÃO

5.1 Implementação dos Tipos de Dados:

A implementação dos diferentes tipos de dados ocorreu de forma regular, sem grandes problemas. No final das contas, os algoritmos implementados não passam de funções que fazem a conversão de ponto fixo para ponto flutuante, e vice-versa. As funções que convertem segundos para algum tipo de dado NTP sempre têm um retorno no formato binário, enquanto que as funções que convertem de um tipo NTP para double sempre tratam a variável de entrada como uma tupla. Dois exemplos são explicitados na figura 5.

```python
def to_NTPtimestamp(tempo):
    # Converte um valor de tempo (em segundos) em ponto flutuante para o formato NTP Timestamp
    # Primeiros 32 bits - inteiro       Ultimos 32 bits - fracionário

    inteiro = int(tempo)
    fracionario = int((tempo - inteiro) * (2**32))

    return struct.pack("!II", inteiro, fracionario)


def timestamp_to_double(timestamp):
    # Converte um timestamp NTP de 64 bits (seconds, fraction) para um valor de ponto flutuante (double).
    seconds, fraction = timestamp
    return seconds + (fraction / 2**32) # Converte fração para segundos e soma
```

Figura 5: Duas implementações de conversão NTP <-> double.

5.2 Implementação do Gerador de Cabeçalhos:

A função responsável por gerar os cabeçalhos do pacote NTP passou por várias iterações. Inicialmente, ela não recebia nenhum parâmetro: tudo era calculado dentro da própria função. Essa primeira iteração foi feita para ser utilizada apenas pelo cliente. Porém, após a decisão de fazer o sistema poder agir tanto como cliente quanto como servidor, a função precisou ser refatorada, para se tornar mais genérica. Agora, ela recebe todos os campos do cabeçalho como um parâmetro diferente, conforme mostrado na Figura 7.

Todos esses parâmetros são convertidos para o formato binário (exceto os timestamps, que já estão no formato correto), e depois são concatenados numa única variável, que então é retornada como um pacote NTP válido. A figura 6 mostra um exemplo do processo. Caso algum dos parâmetros de extension field, key id, ou message digest forem enviados como “0” ou “None”, a função não irá incluí-los no pacote. Esse processo está explicitado na figura 8.

```python
packet = struct.pack(
    "! B B B b",
    (LI << 6) | (VN << 3) | mode,   # LI, VN, mode  (1 byte)
    stratum,                        # stratum       (1 byte)
    poll,                           # poll          (1 byte)
    precision                       # precision     (1 byte)
) + rootdelay + rootdisp            # 4 bytes e 4 bytes
```

Figura 6: Processo de conversão dos parâmetros para binário.

```
def packet_builder(
    # Recebe todas as variáveis necessárias para a criação de um pacote NTP, e retorna um pacote binário

        LI,                         # Leap Indicator,       2 bits
        VN,                         # Version Number        3 bits
        mode,                       # mode                  3 bits      3 - client, 4 - server.
        stratum,                    # stratum               8 bits      não usado pelo client
        poll,                       # poll exponent         8 bits      frequência de envio de mensag
        precision,                  # precision exponent    8 bits
        rootdelay,                  # root delay            32 bits, NTP short
        rootdisp,                   # root dispersion       32 bits, NTP short
        refid,                      # reference ID          32 bits
        reftime,                    # reference timestamp   64 bits, NTP timestamp
        org,                        # origin timestamp      64 bits, NTP timestamp
        rec,                        # receive timestamp     64 bits, NTP timestamp
        xmt,                        # transmit timestamp    64 bits, NTP timestamp
        exf1,                       # Extension Field 1     variável
        exf2,                       # Extension Field 2     variável
        keyid,                      # key ID                32 bits
        chave                       # message digest        128 bits MD5 hash
    ):
```

Figura 7: Parâmetros da função de criação de pacotes.

```
# Adiciona Extension Field 2, se existir
if exf2:
    exf2_padded = exf2 + b"\x00" * ((4 - len(exf2) % 4) % 4)
    packet += exf2_padded

if keyid and chave:
    # Se existirem, calcula o digest (hash HMAC-SHA256) sobre
    digest = calcular_hmac_client(packet, chave)
    packet += struct.pack("!I", keyid)  # Converte o keyid pa
    packet += digest                    # Concatena o digest
```

Figura 8: Processamento dos campos opcionais.

5.3 Implementação das Funções de Cliente:

A implementação do cliente aconteceu sem grandes problemas, exceto pela refatoração que foi explicada no tópico anterior. Logo, será explicado o funcionamento básico do modo de associação 3 (client). Quando iniciado, o sistema irá perguntar qual o modo de associação que o usuário deseja usar (as opções são 3 - Cliente, e 4 - Servidor). Depois, é requerido ao usuário o endereço de IP para conectar o socket e, caso o campo não for preenchido, o sistema seleciona aleatoriamente entre um dos servidores oficiais do NTP.br. A partir de agora, a execução do código ocorrerá em uma de duas funções diferentes: o modo_client(), ou o modo_server(). Aqui será explicado o modo_client(), que processa o envio de pacotes pelo modo de associação 3.

O modo_client() inicialmente irá pedir ao usuário se ele deseja utilizar criptografia, qual a chave e key id que serão utilizados, e qual a frequência de polling desejada pelo usuário. Caso o polling for ativado, as requisições ocorrerão num loop While(True). Antes de fazer uma requisição, o cliente cria um socket, um pacote NTP novo, e calcula o instante de envio do pacote (Figura 9). O pacote é então enviado com ajuda da biblioteca *socket* e, quando a resposta chega, o tempo de chegada é armazenado. O pacote recebido é processado, os valores de T2 e T3 são extraídos, para que o offset (theta) e o round-trip delay (delta) sejam por fim calculados, como especificado na RFC 5905 (Figura 10) (MILLS, *et al*. 2010). Por fim, caso o usuário tenha especificado, o relógio do SO é atualizado, e o loop do polling continua (ou não).

```
# Criação de um socket UDP
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Gera o pacote NTP e envia para o servidor
pacote = packet_builder( 00, VERSION, 3, 0, poll, calcPrecision(),
to_NTPtimestamp(0), None, None, keyid, chave)

T1 = time.time()                          # Tempo de envio do pacote
```

Figura 9: Funcionamento básico do client.

```
# Offset de B relativo a A:     theta = T(B) - T(A) = 1/2 * [(T2-T1) + (T3-T4)]
theta = (1/2) * ((recv_timestamp - T1) + (xmt_timestamp - T4))
# Round trip delay entre A, B, e A: delta = T(ABA) = (T4-T1) - (T3-T2)
delta = (T4 - T1) - (xmt_timestamp - recv_timestamp)
```

Figura 10: Cálculo do theta e do beta.

O modo client foi então testado com diferentes servidores NTP oficiais. Os pacotes NTP foram criados, enviados ao servidor, e recebidos normalmente. Logo, a conexão foi um sucesso. O resultado de um teste feito com o servidor *200.160.0.8:123* pode ser visualizado na figura 11. Após o recebimento do pacote, o relógio da máquina foi atualizada. Um acesso rápido ao website *ntp.br* mostrou que o relógio estava de fato sincronizado corretamente com a rede (Figura 12).

```
======--==--=== Pacote NTP ===-==--======
Stratum: 2
Modo: 4 (servidor)
Root Delay: 1590
Root Dispersion: 0.0024261474609375
Reference Timestamp: (3950625031, 995428206)
Originate Timestamp: (0, 0)
Receive Timestamp:  1741636292.8395405
Transmit Timestamp: 1741636292.8395586
Refid: È º
========--==-======--=======--==-=======

O Offset é de           0.747492790222168 segundos
O Round-Trip Delay é de 0.039687156677246094 segundos

segunda-feira, 10 de março de 2025 16:51:32


Horário do sistema ajustado!
```

Figura 11: Conexão NTP feita com sucesso.

Figura 12: Relógio corretamente sincronizado com o servidor *ntp.br*.

5.4 Implementação das Funções de Servidor:

A implementação do modo de associação 4 (server) também aconteceu sem maiores problemas. Na verdade, o resultado foi similar ao modo client. Ao receber o IP e a porta do servidor, o sistema irá perguntar se o usuário quer visualizar todos os pacotes recebidos, e se ele deseja autenticar os mesmos com uso de uma das chaves compartilhadas armazenadas. O servidor ficará infinitamente esperando por requisições e, caso receber uma, o pacote será autenticado (ou não) e interpretado. Depois disso, será enviado um novo pacote de resposta com o resto dos campos corretamente preenchidos.

Todos os pacotes recebidos são processados pela função interpretador_pacote_server(), que desempacota os binários de todos os campos do header da requisição (Figura 13). Essa função retorna um pacote novo, capaz de atualizar o relógio do cliente que fez a requisição. Esse pacote novo é então enviado de volta ao cliente por meio da biblioteca *socket*.

```
li_vn_mode, stratum, poll, precision = struct.unpack("! B B B b", pacote[:4])
LI = (li_vn_mode >> 6) & 0b11  # Os 2 primeiros bits representam o LI
mode = li_vn_mode & 0b111  # Os 3 bits seguintes representam o mode

ref_id = pacote[12:16].decode('latin1')
root_delay_pacote       = ntpshort_to_double(struct.unpack("!HH", pacote[4:8]))
root_dispersion_pacote  = ntpshort_to_double(struct.unpack("!HH", pacote[8:12]))
ref_timestamp_pacote    = timestamp_to_double(struct.unpack("!II", pacote[16:24]))
orig_timestamp_pacote   = timestamp_to_double(struct.unpack("!II", pacote[24:32]))
recv_timestamp_pacote   = timestamp_to_double(struct.unpack("!II", pacote[32:40]))
xmt_timestamp_pacote    = timestamp_to_double(struct.unpack("!II", pacote[40:48]))
```

Figura 13: Separação dos elementos do header da requisição.

Por fim, foi feito um teste conectando o cliente implementado com o servidor. A comutação de pacotes ocorreu sem erros, então o servidor foi um sucesso. O output do cliente pode ser visto na esquerda da figura 14, enquanto o do server está na direita. O offset existente pode ser explicado pela latência no processamento dos pacotes, já que a linguagem utilizada (*python*) é uma linguagem interpretada, que pode ter uma velocidade de processamento um pouco baixa quando utilizada por alguém sem muito domínio na área.

```
======-==-=== Pacote NTP ===-==-======
Stratum: 2
Modo: 4
Root Delay: 0.0
Root Dispersion: 0.0
Reference Timestamp: -2208988800.0
Originate Timestamp: 1741715202.1370144
Receive Timestamp:  1741715202.1390104
Transmit Timestamp: 1741715202.1410093
Refid:
=======-==-=======-=======-==-=======

O Offset é de           0.0009981393814086914 segundos
O Round-Trip Delay é de 0.0019958019256591797 segundos

...
```

```
=====-==-=== Pacote NTP ===-==-======
Pacote recebido de ('127.0.0.1', 55682):
Stratum: 0
Modo: 3
Root Delay: 0.0
Root Dispersion: 0.0
Reference Timestamp: 0.0
Originate Timestamp: 3950704002.1370144
Receive Timestamp:   0.0
Transmit Timestamp: 0.0
Refid:
=======-==-=======-=======-==-=======

Resposta enviada para ('127.0.0.1', 55682):
```

Figura 14: Funcionamento da requisição enviada ao servidor (à direita), e resposta recebida (à esquerda).

5.5 Implementação do Sistema de Autenticação:

Por fim, com o cliente e o servidor funcionais, foram implementadas as funções de criptografia. Nos tópicos anteriores já foram mostradas as situações em que a criptografia é

utilizada no corpo do código, então aqui só será explicado os cálculos utilizados e um exemplo do sistema funcionando na prática.

Para a implementação da criptografia, foi utilizada a biblioteca *hashlib*, com uma função pronta para o cálculo do HMAC-SHA256. Antes da execução dessas funções, o sistema define um dicionário (variável global) responsável por armazenar todas as chaves com os seus key id's relacionados. Esse dicionário só é acessado pelo servidor (Figura 15).

```
# Dicionário de chaves pré-compartilhadas,
CHAVES_DICT = {
    1: b"eu_sou_uma_chave",             #
    2: b"chave_chavosa_chaveada",       #
    42: b"super_chave_chavosa",         #
    51: b"chave_chavente_chaveante",    #
    55: b"eu_nao_sou_uma_chave"
}
```

Figura 15: Dicionário que armazena as chaves pré-compartilhadas.

Existem duas funções de cálculo do HMAC: uma usada pelo cliente, e outra usada pelo servidor. A função usada pelo cliente (calcular_hmac_client()) recebe como parâmetro um pacote e uma chave, e o hash é calculado com base nos dois. Já a função do servidor (calcular_hmac()) recebe como parâmetro um pacote e um key id, e só irá calcular o hash caso o key id fornecido pelo cliente seja válido (Figura 16).

```
def calcular_hmac(packet, keyid):
    # Função que recebe um keyid, e calcula o HMAC de um pacote com

    chave = CHAVES_DICT.get(keyid)  # Servidor pega o keyid forneci

    if not chave:
        print("Keyid inválido!")
        return

    # Depois, o servidor calcula o hmac com base na sua chave equiv
    hmac_packet = hmac.new(chave, packet, hashlib.sha256).digest()
    return hmac_packet              # E retorna o hash calculado


def calcular_hmac_client(packet, chave):
    # Função aonde um cliente envia uma chave e um pacote, e recebe
    return hmac.new(chave, packet, hashlib.sha256).digest()
```

Figura 16: Funções de cálculo de hash por HMAC-SHA256.

Quando um cliente envia um pacote, e quiser usar autenticação, ele envia um key id e uma chave para ser utilizada pela função de criação de pacotes para o cálculo do message digest. Já o servidor, ao receber um pacote com autenticação, ele separa o key id e o digest da requisição, calcula um digest próprio com base no key_id fornecido pelo cliente, e compara o digest do cliente com o seu próprio digest calculado. Caso forem iguais, o processamento do pacote ocorrerá normalmente. A função responsável por isso está explicitada na Figura 17.

```python
def validar_hmac(packet):
    # Pega um pacote recebido pelo server, e valida ele

    if len(packet) < 48 + 4 + 32:
        return False, 0, 0    # Pacote muito curto para ter autenticação

    # Separa o Key ID e o HMAC, para poder usá-los depois
    keyid = struct.unpack("!I", packet[-36:-32])[0]
    hmac_recebido = packet[-32:]

    # Agora o servidor vai recalcular o hmac usando apenas o keyid, e a chave que ele tem armaz
    packet_sem_hmac_keyid = packet[:-36]

    try:
        hmac_recalculado = calcular_hmac(packet_sem_hmac_keyid, keyid)  # Tenta calcular o hmac
    except ValueError:
        return False, 0, 0    # Significa que o keyid não existia/inválido

    # Por fim compara os dois, pra ver se o cliente enviou um pacote válido
    # Vale notar que não adianta o cliente só enviar uma chave válida. Ele precisa enviar o seu
    return hmac.compare_digest(hmac_recebido, hmac_recalculado), keyid, CHAVES_DICT.get(keyid)
```

Figura 17: Validação do hash por parte do servidor.

Por fim, foram feitos diversos testes entre o cliente e o servidor implementados, com a criptografia ativada, com o fim de identificar se o sistema está funcionando corretamente. Quando o cliente envia uma chave válida, mas com o key id errado, o servidor rejeita a requisição (Figura 18). Quando o cliente envia um pacote sem nenhum key id, o cliente também rejeita a requisição (Figura 19). Já quando o cliente envia um pacote com um key id e uma chave válida, a requisição é processada normalmente (Figura 20).

```
======--==--=== Pacote NTP ===--==-======
Timeout! Nenhuma resposta do servidor.

Escolha um modo de associação: [3 - client] [4 - servidor
]
 - Antes de iniciar um servidor, recomenda-se sincronizar
 com um servidor oficial pelo modo client.
R: _
```

```
Pacote falso recebido!

Pacote falso recebido!
```

Figura 18: Funcionamento da requisição enviada ao servidor (à direita), e resposta recebida (à esquerda).

```
======--==--=== Pacote NTP ===--==-======
Timeout! Nenhuma resposta do servidor.

Escolha um modo de associação: [3 - client] [4 - servidor
]
 - Antes de iniciar um servidor, recomenda-se sincronizar
 com um servidor oficial pelo modo client.
R:
```

```
r: s

Servidor iniciado em 127.0.0.1:124, aguardando requisições...
Keyid inválido!

Pacote falso recebido!
```

Figura 19: Funcionamento da requisição enviada ao servidor (à direita), e resposta recebida (à esquerda).

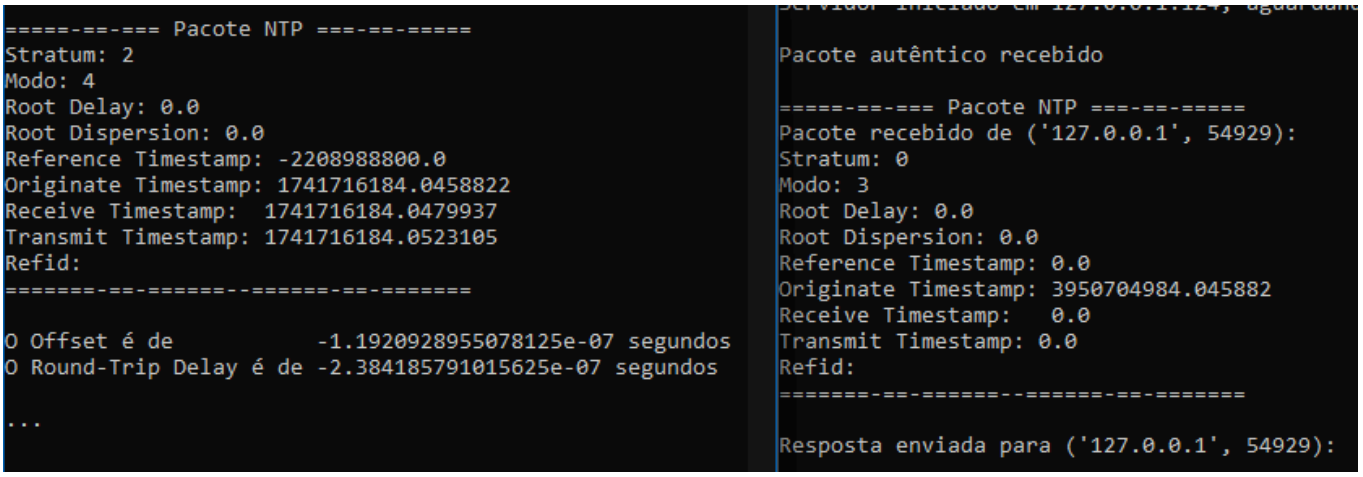

Figura 20: Funcionamento da requisição enviada ao servidor (à direita), e resposta recebida (à esquerda).

6. CONCLUSÕES

A implementação NTP criada no presente trabalho conseguiu ser compatível com todos os servidores NTP oficiais testados, e também foi capaz de responder às requisições de outros clientes NTP sem problemas. Como esperado, a linguagem de programação *python* também se mostrou extremamente conveniente para o processo, já que a sua sintaxe simples e sua abundância de bibliotecas fizeram da escrita do código um processo rápido e eficiente.

Além disso, praticamente todos os requisitos do trabalho conseguiram ser cumpridos, com exceção do requisito 1, que foi completado em partes. Segundo ele, a implementação deveria seguir a especificação da RFC 5905 (MILLS *et al*, 2010). Porém, devido à complexidade do que essa RFC considera como "uma implementação completa do RTP", não foi possível cumprir o requisito totalmente. Algumas partes, como o mecanismo de Slew e a filtragem de pacotes, precisaram ser deixadas de fora. Porém, essas partes estão longes de serem essenciais para o funcionamento de uma implementação do protocolo.

Apesar de tudo, também vale citar que a implementação do sistema de autenticação com HMAC-SHA256 de fato conseguiu garantir a segurança e integridade das comunicações, reforçando a confiabilidade do protocolo. Portanto, conclui-se que o trabalho conseguiu atingir os seus objetivos principais com sucesso, proporcionando uma base sólida para futuras melhorias e expansões, caso necessário.

BIBLIOGRAFIA

CAMPISTA, Miguel Elias M. et al. Interconexão de Redes na Internet do Futuro: Desafios e Soluções. **Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC**, v. 2010, p. 47-101, 2010.

CHANDRA, Deka Ganesh; KATHING, Margaret; KUMAR, Das Prashanta. A comparative study on IPv4 and IPv6. In: **2013 International Conference on Communication Systems and Network Technologies**. IEEE, 2013. p. 286-289.

CORONA, Adrián Estrada. *et al*. **Protocolos TCP/IP de internet**. 2004.

DINIZ, Pedro Henrique; JUNIOR, Nilton Alves. Ferramenta IPERF: geração e medição de Tráfego TCP e UDP. **Notas Técnicas**, v. 4, n. 2, 2014.

DOURADO, Ícaro Cavalcante; COSTA, Daniel G. **Utilizando UDP-Lite em Comunicações Multimídia em Tempo Real**. 2008

LEINER, Barry M. et al. **A brief history of the Internet**. ACM SIGCOMM computer communication review, v. 39, n. 5, p. 22-31, 2009.

MILLS, David L. **Network Time Protocol (NTP)**. RFC 958 . Internet Engineering Task Force (IETF), set. 1985. Disponível em: https://tools.ietf.org/html/rfc958. Acesso em: 9 de março de 2025.

MILLS, David L *et al*. **Network Time Protocol Version 4: Protocol and Algorithms Specification**. RFC 5905. Internet Engineering Task Force (IETF), june 2010. Disponível em: <https://datatracker.ietf.org/doc/html/rfc5905>. Acesso em 1 de março de 2025

MKACHER, Faten; BESTEL, Xavier; DUDA, Andrzej. Secure time synchronization protocol. In: **2018 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)**. IEEE, 2018. p. 1-6.

MONTEIRO, Luís. **A internet como meio de comunicação: possibilidades e limitações**. In: Congresso Brasileiro de Comunicação. sn, 2001.

NETO, Jahyr Gonçalves. **Desenvolvimento de uma Plataforma Multimídia Utilizando a Linguagem Python**. 2007. Tese de Doutorado. Universidade Estadual de Campinas.

OLIVEIRA, Ronielton Rezende. **Criptografia simétrica e assimétrica-os principais algoritmos de cifragem**. Segurança Digital [Revista online], v. 31, p. 11-15, 2012.

SCARSELLI, Rafael B.; SOARES, Leonardo F.; MORAES, Igor M. Evaluating Cryptographic Algorithms in IEC 61850 Networks. In: **2019 10th International Conference on Networks of the Future (NoF)**. IEEE, 2019. p. 9-16.

SOARES, Felipe Augusto Moreira; DANTAS, Wesley Moreira. **Protocolo de Tempo para Redes (NTP): utilizando o sistema de posicionamento global como referência de tempo confiável**. FATECS, Brasília. 2022.

TANENBAUM, Andrew S. **Redes de Computadores**, 7ª Edição, Editora Campus, Rio de Janeiro – RJ, 2003.

UTIME, Sérgio Akira. **Relógio autojustável por meio do protocolo NTP**. Trabalho de Conclusão de Curso. Universidade Tecnológica Federal do Paraná. 2021