Sincronização de semáforos baseada em NTP

https://github.com/AndBarata/DistributedSystems/tree/main/Projeto

1st André de Azevedo Barata
up201907705@up.pt

2nd Diogo Vilela
up201907804@up.pt

***Abstract*—A sincronização de relógios num sistema distribuído é de elevada importância, de modo a manter uma noção explicita do tempo ao longo do sistema. Uma grande parte das soluções existentes para o comportamento de semáforos baseia-se em soluções de tempo fixo, em que a mudança de cor ocorre em ciclos de tempo invariável, sem qualquer mecanismo de sincronização com semáforos que funcionam em perpendicular. Assim, este projeto visa explorar a sincronização de 2 semáforos, perpendiculares, numa interseção, cujo objetivo é de fazer a transição da cor de ambos com a maior precisão possível, garantindo a longevidade da sincronização. Para tal utilizou-se o protocolo NTP, para sincronizar os relógios, e desenvolveu-se um algoritmo de ajuste de rate. O sistema é constituído por 2 Raspberry Pi 4, que simulam o comportamento dos semáforos, um monitor, para observar os resultados, um ponto de acesso e um servidor NTP.**

Keywords: Sincronização de relógios, NTP, Raspberry Pi 4, semáforos.

I. Introdução

Este trabalho apresenta um caso prático de sincronização de relógios usando o Network Time Protocol (NTP). O problema apresentado consiste em sincronizar as transições de estado de 2 semáforos de uma interseção sem que haja trocas de informações entre os eles. Deste modo, todos os semáforos irão utilizar um relógio abstrato, escravizado de acordo com um servidor NTP, para inferir o seu estado (Verde, Vermelho).

Por exemplo, considerando a figura 1, quando o semáforo da via horizontal (1) está verde, o da via vertical (2) está vermelho. Ora, caso esta alteração de estado ocorra de t em t segundos, cada semáforo deve tomar essa decisão autonomamente de acordo com o seu relógio. Os dois relógios apresentam características diferentes, pelo que t segundos no semáforo 1 pode representar $t+\delta$ segundos no relógio 2, pelo que a transição no 2 seria tomada δ segundos mais tarde. Normalmente, δ é um valor muito pequeno, pelo que quando considerando uma só transição este valor é irrelevante. Contudo, esta discrepância acumula com o tempo e, sem a aplicação de mecanismos de sincronização, pode levar ao caso de os dois semáforos estarem verdes ao mesmo tempo. Uma correta sincronização não irá colocar δ a zero, mas sim mantê-lo com um valor baixo o suficiente para que seja irrelevante para o sistema e constante ao longo do tempo. Para tal, cada semáforo irá atualizar periodicamente o "rate", o "offset" e o "delay" dos seus relógios com um servidor NTP, garantindo que δ não acumula infinitamente.

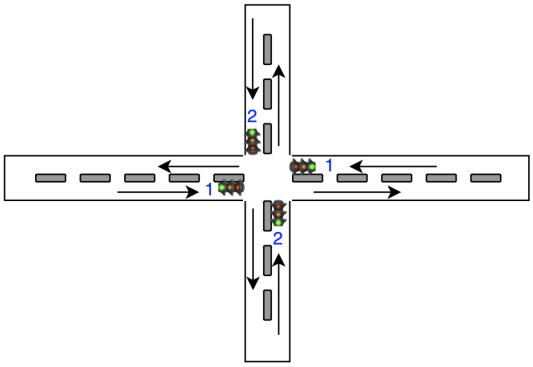


Fig. 1: Interseção de uma estrada

Atualmente, as soluções para a mudança de estado dos semáforos pode ser agrupada em 2 grupos distintos, um primeiro baseado em sistemas de tempo fixo e um segundo em sistemas baseados em sensores. Para o primeiro caso, a mudança de estado é feita através de ciclos de tempo fixo, com base no relógio de cada semáforo. Contudo, maior parte dos semáforos não têm qualquer tipo de sincronização, tendo, em vez disso uma manutenção periódica aos seus relógios. O segundo caso utiliza sensores para medir o fluxo de trânsito, ou controlar a velocidade. Com base nos valores medidos por estes sensores, uma decisão é posteriormente tomada pelo sistema.

II. Protocolo NTP

O Network Time Protocol é um protocolo de sincronização de relógios que sincroniza os relógios de um sistema distribuído através de redes com comutação de pacotes e de latência variável, com uma precisão na ordem dos milissegundos. Este protocolo foi, primordialmente, concebido para ter uma alta exatidão e fiabilidade [2].

Para uma típica operação do NTP, o cliente questiona um ou mais servidores NTP e recebe o tempo atualizado segundo o servidor. Após a receção dos dados do servidor, o cliente calcula o offset e o rate do seu relógio relativamente ao servidor e o delay da rede. Para tal, é necessário que ele saiba os timestamps da mensagem enviada por ele ao servidor e da consequente resposta. A figura 2 ilustra uma execução do protocolo. Os timestamps necessários para o cálculo dos parâmetros de sincronização são: tempo em que pedido é enviado pelo cliente, t0, tempo em que o pedido é recebido pelo servidor, t1, tempo em que a

resposta é enviada pelo servidor, t2 e tempo em a resposta é é recebida pelo cliente, t3.

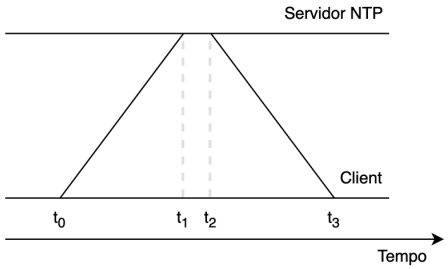


Fig. 2: Ordem de eventos do protocolo NTP

Com estes 4 tempos é possível inferir os seguinte parâmetros:

- Offset, θ: corresponde à diferença entre o relógio do cliente e do servidor, equação1a.
- Delay, δ: corresponde ao tempo que o pedido demora a chegar ao servidor, equação 1b.
- Rate, ρ: razão entre a velocidade de incrementação do relógio do servidor e do cliente, equação 1c

$$\theta = \frac{((t_1 - t_0) + (t_2 - t_3))}{2} \tag{1a}$$

$$\delta = (t_3 - t_0) - (t_2 - t_1) \tag{1b}$$

$$\text{rate} = \left| \frac{t_1' - t_1}{t_0' - t_0} \right| \tag{1c}$$

Na equação 1c, os valores t_0' e t_1' correspondem aos timestamps mais recentes, enquanto t1 e t0 correspondem aos timestamps da sincronização anterior. Deste modo, o rate apenas depende das duas últimas iterações entre cliente-servidor. Contudo, esta equação não tem em conta o delay da rede, sendo que este nunca é constante e influencia o valor do rate. Na figura 3 observa-se o efeito do delay da rede para o período NTP, ou seja o período entre t_1 e t_1' ($T_{NTP} = t_1' - t_1$). Idealmente, o delay é constante com o valor μ. Contudo, no caso em que um pedido sofra um delay menor que o esperado, d, e o seguinte um delay maior que o esperado, d', o valor de T_{NTP} será superior ao real, de acordo com a equação 2, levando a que o cliente julgue que precise uma variação de rate superior à real.

$$T_{NTP_{incorreto}} = T_{NTP} + d + d' \tag{2}$$

Deste modo, para contrariar este tipo de casos, o valor esperado do delay é atualizado a cada sincronização de acordo com a equação 3a. A cada sincronização, é calculado o erro introduzido pela variação do delay (equação 3b) e descontado no cálculo do rate, equação 3c.

$$\mu_k = \frac{N \times \mu_{k-1} + \delta_k}{N+1} \tag{3a}$$

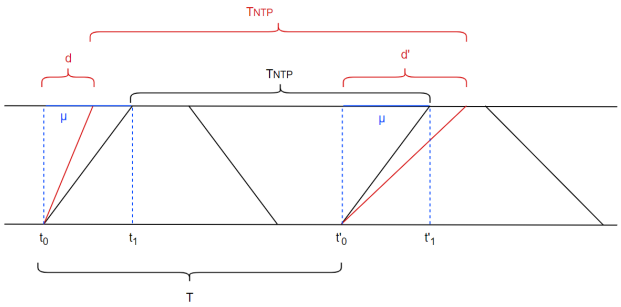


Fig. 3: Efeito da variação do delay no período NTP

$$\varepsilon = \mu - \delta + \mu - \delta' \tag{3b}$$

$$\rho = \left| \frac{t_1' - t_1 - \varepsilon}{t_0' - t_0} \right| \tag{3c}$$

Por fim, sempre que o cliente precisar de utilizar um tempo sincronizado, irá calcular o tempo que passou desde a última atualização de acordo com o seu relógio (não sincronizado), multiplicado pelo rate, somado com o offset e adicionado ao tempo da última sincronização.

$$\text{corrected\_time} = t_0' + \text{elapsed\_time} \cdot \text{rate} + \theta \tag{4}$$

III. Arquitetura do Sistema

Na interseção descrita na figura 1 existem 4 semáforos, pelo que, idealmente, cada um seria representado por uma raspberry Pi. Contudo, devido ao facto de apenas existirem duas raspberrypis disponíveis, os dois semáforos de cada sentido serão controlados por uma das duas raspberrys e terão o mesmo comportamento. Assim, quando a raspberry Pi A está no estado VERDE é permitido o transito horizontal e quando a raspberry Pi B está VERDE é permitido o trânsito vertical. As raspberrypis nunca apresentam o mesmo estado em simultâneo.

O sistema foi desenhado de acordo com a figura 4. Ambas as raspberrypis foram conectadas à rede Wi-Fi do portátil, que por sua vez comunica com o servidor NTP pool.ntp.org. Cada raspberry Pi representa um cliente que envia para o portátil os pedidos NTP de acordo com a secção II, que por sua vez são reencaminhados para o servidor. A resposta faz o caminho oposto. O estado da cada raspberry Pi é enviado para o portátil cada vez que existe uma transição. Este processo serve apenas para monitorização e está representado na figura como monitor. O monitor regista o tempo local da chegada da mensagem de mudança de estado de cada raspberry Pi e guarda o valor num ficheiro .txt. Por exemplo, A fica VERMELHO e B fica VERDE, o monitor mostra o tempo de cada transição (que deve ser muito semelhante).

O protocolo empregue para a comunicação é o "Transmission Control Protocol" - TCP. Se o servidor NTP não responder à solicitação, é acionado um "timeout", e o

sistema tenta restabelecer a ligação. Esta funcionalidade permite que a sincronização continue sem que o servidor esteja disponível, utilizando os último valores registados.

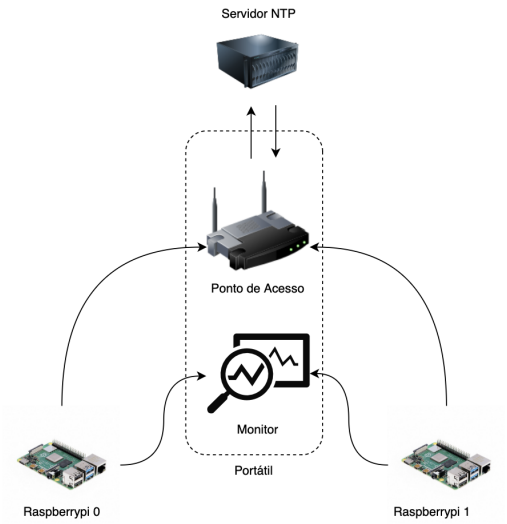


Fig. 4: Diagrama da arquitetura do sistema [1]

IV. Implementação

Durante esta secção irá ser descrito o algoritmo de sincronização e, posteriormente, a sua implementação em Python. Para além disso, é feita a seguinte distinção entre três relógios:

- **Relógio monotónico**: Relógio do hardware do computador. Nunca é corrigido e conta o tempo a partir do qual o sistema operativo foi iniciado.
- **Relógio NTP**: Relógio do servidor NTP e assumido como correto. Este relógio escraviza o sistema.
- **Relógio abstrato**: Relógio implementado em software em cima do sistema operativo. É o relógio monotónico corrigido em "rate" e "offset" de acordo com a secção II.

Posto isto, as raspberrypis têm dois relógios: monotónico e abstrato, sendo que este último é utilizado para tomar as decisões de mudança de estado. O monitor apenas utiliza o seu relógio monotónico para marcar os "timestamps" das mudanças de estado de A e B. O servidor NTP responde aos pedido com os "timestamps" do relógio NTP. A figura 5 ilustra um pedido de tempo ao relógio relógio abstrato. De referir que, o relógio abstrato é o relógio corrigido segundo os parâmetros obtido pelo último pedido NTP (offset, rate e delay), segundo as equações 1 e 4.

A. Algoritmo

O programa está dividido em dois processos distintos que correm em paralelo: correção dos parâmetros do relógio e atualização do Estado. Ambos os algoritmos estão expressos em pseudo-código nas figuras 6 e 7, respetivamente.

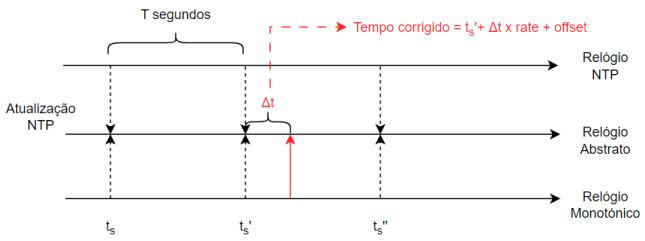


Fig. 5: Correção do tempo do relógio abstrato

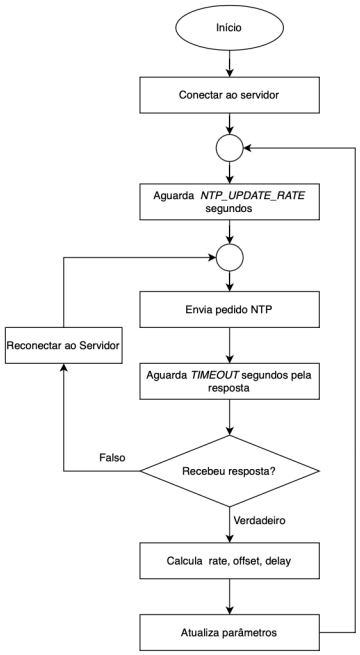


Fig. 6: Pseudo-código relativo à correção do relógio abstrato

B. Implementação em Python

O algoritmo foi implementado na linguagem Python. A sincronização NTP foi feita por uma thread periódica, sendo que a sua periodicidade é definida pelo utilizador na invocação do programa. Adicionalmente, o utilizador também pode definir o sentido a que o semáforo corresponde e o servidor NTP a que se pretende conectar. O programa consiste em 3 classes que descrevem o relógio abstrato, o cliente NTP e o monitor.

V. Resultados

Foram realizadas 6 experiências em 3 servidores diferentes. S1 corresponde ao pool.ntp.org, S2 a ntp0.ntp-servers.net e S3 ao *localhost*, que neste caso é o portátil, que faz de ponto de acesso. A experiências realizadas estão presentes na tabela I cujo os valores a vermelho foram considerados mais relevantes. As experiências realizadas para cada servidor foram as seguintes:

- **Correção rate**: Apenas o rate dos relógio é corrigido.

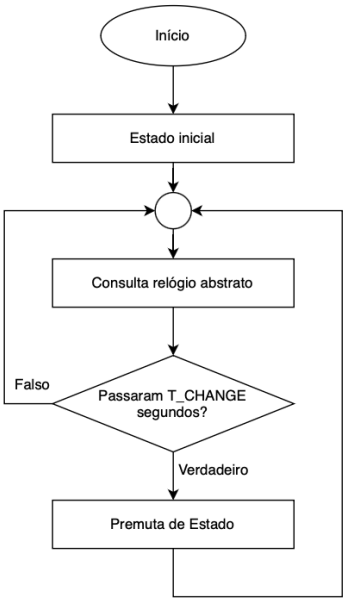


Fig. 7: Pseudo-código relativo à atualização do estado

	S1 T =5	S2 T=5	S3 T=5	S2 T=2	S2 T=15	
Correção rate .	0.9964	0.0285	-	-	-	Média slots
	10.8750	0.2650	-	-	-	Máx. slots
	1.3495	0.1255	-	-	-	Jitter
Correção rate + offset	0.0244	0.0252	0.0231	0.0521	0.0124	Média slots
	0.3590	0.8280	0.7340	0.0521	0.3910	Máx. slots
	1.0195	0.2104	0.7229	0.3229	0.2051	Jitter
Correção rate + offset + delay	0.0103	0.0118	0.0079	0.0568	0.0325	Média slots
	0.4380	0.1880	0.0310	0.8120	0.8900	Máx slots
	0.2037	0.2104	0.0264	0.3535	0.2052	Jitter
Sem correção .	0.0030					Média slots
	0.0620					Máx. slots

TABLE I: Experiências realizadas

- **Correção rate + offset**: Offset corrigido, rate corrigido de acordo com 1c
- **Correção rate + offset + delay**: Offset corrigido, rate corrigido de acordo com 3c
- **Sem correção**: Os relógio A e B não sofrem qualquer sincronização, para além de ajuste inicial do offset.
- **T=2, T=5, T=15**: Periodicidade da correção por parte do servidor NTP, em segundos.

As figuras 9a e 9b ilustram dois segmentos, um em que o delay não é considerado no cálculo do rate e outro em que é. A figura 10 demonstra a influência do delay na precisão dos slots. Define-se como slot o tempo que um dado semáforo fica num estado, assim, a precisão dos slots é a diferença de tempo entre a mudança de estado do semáforo A e do semáforo B, como indicado a vermelho na figura 8.

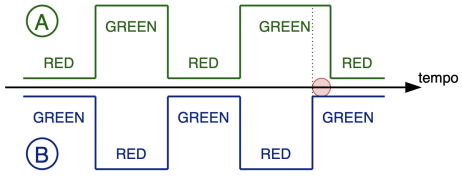
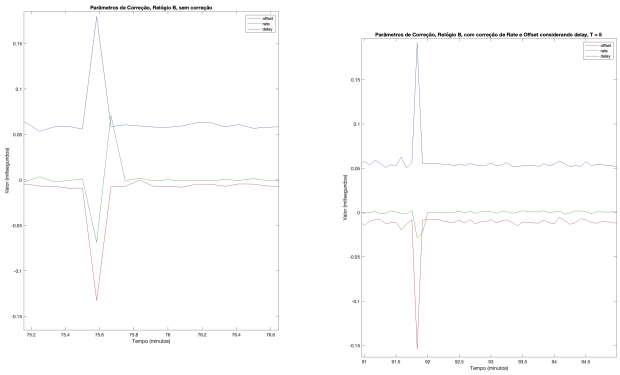


Fig. 8: Medição da precisão dos slots



(a) Rate sem considerar delay, sinal verde

(b) Rate considerando delay

Fig. 9: Influência da variação do delay no rate, sinal verde

Por fim, a duração de cada experiência está presente na tabela II, resultando num total de 34 horas e 15 minutos.

-	S1 T=5	S2 T=5	S3 T=5	S2 T=2	S2 T=15
Correção rate	30	90	-	-	-
Correção Rate + offset	15	150	90	120	160
Correção Rate + offset + delay	150	320	90	120	120
Sem correção	600	600	600	600	600

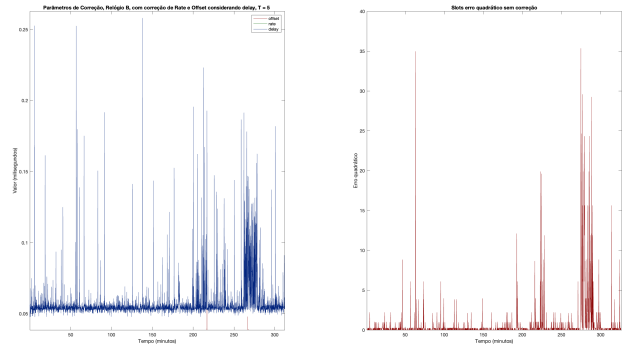
TABLE II: Duração de cada experiência, em minutos

VI. Conclusão

A. Análise dos resultados

De acordo com os resultados da tabela I é possível inferir o seguinte:

1) O servidor mais estável é o S3: devido aos valor baixos do jitter, algo esperado, visto que é um servidor local.
2) O experiência que apresentou melhores resultados foi quando não houve correção de relógio.
3) A experiências em que o jitter é mais baixo apresentam melhores resultados.
4) A variação do período de correção não tem grande influência nos resultados.



(a) Variação do erro quadrático com o tempo

(b) Variação do delay do servidor com o tempo

Fig. 10: Influência da variação delay no erro entre slots

Relativamente ao ponto 2), os relógios das raspberry Pis são bastante parecidos e apresentam drifts muito baixos (cerca de $15ppm$, segundo [4]). Contudo, um drift de $15ppm$ ao logo de uma hora representa um erro de $36ms$, e ao longo de uma semana de $6s$, o que já é critico para um sistema de semáforos. Ora, apesar de os resultados das experiências com sincronização serem piores a curto prazo, este são constantes com o tempo, ou seja, ao longo de uma semana o erro seria sensivelmente o mesmo nos casos com sincronização.

Os relógios sincronizados apresentam um desempenho inferior a curto prazo, devido ao o ajuste constante do rate, que causa variações que dependem das condições da rede. Por exemplo, na figura 10 em volta dos $80minutos$, as condições da rede variam muito o que resulta num aumento do erro quadráticos da precisão das slots. Este efeito é minimizado pela consideração do delay no cálculo do rate. A figura 9a apresenta um segmento em que houve uma variação do delay que não foi considerada, o que resulta numa proporcional variação do rate. Enquanto que na figura 9b esta variação foi considerada, resultando num rate mais estável. Estas figuras, revelam também a diferença entre as equações 1c e 3c, respetivamente.

Relativamente ao ponto 3), evidenciam-se que os resultados da experiência com correção de rate no servidor S1 e no servidor S3 com correção de delay, offset e rate. Na verdade, o servidor S1 é bastante instável: para além de apresentar um jitter elevado (1 segundo) tem também uma disponibilidade muito baixa. Ora, nesta experiência o servidor esteve indisponível durante vários minutos, sendo que nos instantes anteriores o delay variou bastante, comprometendo o rate. Assim os relógio utilizaram os últimos parâmetros disponíveis para a sua sincronização, que estavam incorretos, resultando em erros na precisão das slots de $10s$ em poucos minutos. Em contraste a este resultado, a experiência no servidor 3 com correção de offset, rate e delay teve um jitter de $26ms$ e um erro na precisão das slots de $8ms$, o baixo jitter conduziu a baixa variação de rate, que durante toda a experiência apenas variou 0.0088, o que é muito baixo comparando com a outra experiência nesse servidor que com um jitter de 0.7229 o rate variou 0.7590.

No que diz respeito ao ponto 3, como os relógios das raspberry Pi apresentam drifts muito baixos, as correções não precisam de ter um período muito baixo, pelo que os resultados são idênticos para vário períodos de sincronização. Esta conclusão é fundamentada pelo bom desempenho da experiência sem correção.

B. Conclusões finais

Os resultados deste projeto corroboram a influência do jitter na sincronização de relógio. Para além disso, permitem inferir sobre a melhor metodologia de sincronização. Apesar de os melhores resultados serem com correção de rate, delay e offset, os sistema distribuídos atuais não usam o offset como parâmetro de correção, visto que pode resultar em ajustes negativos o que leva a que o mesmo relógio apresente o mesmo timestamp em dois momentos consecutivos.

Para além disso, uma introdução do delay no cálculo do rate não é uma metodologia usual na sincronização. De facto, sistema atuais utilizam vários valores passados de offset e períodos de atualização segundo o servidor NTP para inferir o rate, enquanto que neste projeto apenas foi utilizado a amostra anterior do período segundo o servidor NTP.

Por fim, o resultado das sincronizações seria mais evidente caso a duração dos teste fosse maior. Deste modo, seriam necessário vários dias de teste para pelos menos duas metodologias: sem sincronização e com sincronização.

C. Contribuições

O grupo que realizou o projeto é constituído por dois elementos, sendo as contribuições as seguintes:

- André de Azevedo Barata - 50%
  - Implementação do cliente;
  - Métodos de sincronização;
  - Geração dos resultados;
  - Análise dos resultados;
- Diogo Vilela - 50%
  - Métodos de sincronização;
  - Montagem do sistema;
  - Realização das experiências;
  - Análise dos resultados;

REFERÊNCIAS

[1] Digi-Key Electronics, https://www.digikey.pt/pt/products/detail/raspberry-pi/SC0194-9/10258781
[2] D.L. Mills. Internet time synchronization: the network time protocol. IEEE Transactions on Communications, 39(10):1482–1493, 1991.
[3] Cisco Press, Time Error > Clocks, Time Error, and Noise, https://www.ciscopress.com/articles/article.asp?p=3128857&seqNum=2
[4] SwitchDoc Labs Blog, Benchmarks of Real Time Clocks; Raspberry Pi and Arduino, https://www.switchdoc.com/2016/06/benchmarks-real-time-clocks-raspberry-pi-arduino/
[5] GitHub do projeto Semaphore Clock Syncronization

```python
class NTPclient():
    def __init__(self, start_monotonic, start_datetime):
        # Connection parameters
        #self.host = 'pool.ntp.org' # case of NTP online server #ntp0.ntp-servers.net
        self.host = Servidor_NTP
        self.port = 123
        self.read_buffer = 1024
        self.address = (self.host, self.port)
        self.epoch = 2208988800
        self.client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

        self.start_monotonic = start_monotonic
        self.start_datetime = start_datetime

    def monotonicToDatetime(self, now, start = 0):
        variation = timedelta(seconds=(now-start))
        elapsed_datetime = self.start_datetime + variation

        return elapsed_datetime


    def getServerTime(self):

        '''
        This function communicates with an NTP server to get the server time. It sends a request to the server,
        receives the response, unpacks the timestamps from the response, and converts these timestamps to datetime objects.

        The function returns five datetime objects:
        - ref_time: Reference since the NTP clock started (Jan. 1st, 1900)
        - orig_time: The local time when the request was sent
        - rx_time: The time at the NTP server when the response was received.
        - tx_time: The time at the NTP server when the response was sent.
        - dest_time: The local time when the response was received.
        '''

        data = b'\x1b' + 47 * b'\0'
        orig_time = time.monotonic()

        self.client.sendto(data, self.address)
        self.client.settimeout(NTP_UPDATE_RATE/2)  # set a timeout of half the NTP update rate

        try:

            data, self.address = self.client.recvfrom(self.read_buffer)


            orig_int, orig_frac, ref_int, ref_frac, rx_int, rx_frac, tx_int, tx_frac = struct.unpack("!12I", data)[4:12]


            rx_int -= self.epoch
            tx_int -= self.epoch

            # Convert the fractional part to microseconds (1 second = 2**32 fractional units)
            ref_frac = ref_frac * 1e6 // 2**32
            rx_frac = rx_frac * 1e6 // 2**32
            tx_frac = tx_frac * 1e6 // 2**32


            t_timedelta = timedelta(microseconds=ref_frac)
            ref_time = datetime.fromtimestamp(ref_int) + t_timedelta

            t_timedelta = timedelta(microseconds=rx_frac)
            rx_time = datetime.fromtimestamp(rx_int) + t_timedelta

            t_timedelta = timedelta(microseconds=tx_frac)
            tx_time = datetime.fromtimestamp(tx_int) + t_timedelta

            orig_time = self.monotonicToDatetime(orig_time, self.start_monotonic)
            dest_time = self.monotonicToDatetime(time.monotonic(), self.start_monotonic)

            return ref_time, orig_time, rx_time, tx_time, dest_time


        except (socket.error, socket.timeout):
            # If a socket error occurs, print an error message and try again
            return 0
```

Apêndice 1 - Classe em python do cliente NTP

```python
class AbstractClock():
    def __init__(self, NTP_UPDATE_RATE):

        # Clock correction parameters
        self.delay = timedelta(0)
        self.last_delay = timedelta(0)
        self.last_offset = 0
        self.offset = 0
        self.rate = 1
        self.last_rate = 1
        self.mean_delay = 0
        self.n_corrections = 0

        file_path = "clockA_corrected_delay_2NTP.txt"
        self.file = open(file_path, "w")



        # Time parameters for rate
        self.start_datetime = datetime.now() # datetime of when the clock was started
        self.start_monotonic = time.monotonic() # monotonic time of when the clock was started

        # NTP client
        self.ntp_client = NTPclient(self.start_monotonic, self.start_datetime)
        self.last_ntp_timestamp = self.ntp_client.getServerTime()[2]
        self.ntp_timestamp = self.ntp_client.getServerTime()[3] # datetime of when the NTP server was polled



        # intrinsic clock attributes
        self.timestamp = time.monotonic()
        self.last_timestamp = self.timestamp

        self.correctClock()



    def correctClock(self):
        t_local = time.monotonic()
        ntp_time = self.ntp_client.getServerTime()
        if ntp_time:
            t0 = ntp_time[1]
            t1 = ntp_time[2]
            t2 = ntp_time[3]
            t3 = ntp_time[4]

            # Update the rate parameters
            self.last_ntp_timestamp = self.ntp_timestamp
            self.ntp_timestamp = t1
            self.last_timestamp = self.timestamp
            self.timestamp = t_local

            self.updateDelay(t0, t1, t2, t3)
            self.updateOffset(t0, t1, t2, t3)

            self.mean_delay = (self.mean_delay * self.n_corrections + self.delay.total_seconds())/(self.n_corrections + 1)
            self.n_corrections += 1
            self.updateRate(t0, t1, t2, t3)




        else:
            print("Error connecting to NTP server. Retrying...")

        #print("\n_NTP update_\n") # DEBUG



        print(f"offset:{self.offset.total_seconds():.5f}")
        print(f"rate:{self.rate}")
        print(f"delay:{self.delay}")
        print("\n")
        self.file.write(f"offset:{self.offset.total_seconds():.5f}\n")
        self.file.write(f"rate:{self.rate}\n")
        self.file.write(f"delay:{self.delay}\n")
        self.file.flush()  # Ensure immediate write to the file


    def periodicClockUpdate(self):
        while True:
            self.correctClock()
            time.sleep(NTP_UPDATE_RATE)


    def updateOffset(self, t0, t1, t2, t3):
        self.last_offset = self.offset
        self.offset = ( t1-t3 + t2-t0) / 2

    def updateRate(self, t0, t1, t2, t3):
        self.last_rate = self.rate
        delta_ntp = (self.ntp_timestamp - self.last_ntp_timestamp).total_seconds()
        delta_delay = (self.delay + self.last_delay).total_seconds()
        delta_local = (self.timestamp - self.last_timestamp)
        self.rate = abs( (delta_ntp - delta_delay + 2*self.mean_delay)/(delta_local))
        #self.rate = abs( ( + (self.last_delay-self.delay).total_seconds())/(self.timestamp - self.last_timestamp))

    def updateDelay(self, t0, t1, t2, t3):
        self.last_delay = self.delay
        self.delay = ((t3-t0) - (t2-t1))/2

    def getCorrectedTime(self):
        now = time.monotonic()
        elapsed_time = now - self.timestamp
        corrected_time = self.timestamp + elapsed_time *self.rate + self.offset.total_seconds()
        return self.ntp_client.monotonicToDatetime(corrected_time, self.start_monotonic)
```

Apêndice 2 - Classe em python do relógio abstrato