Algoritmo de Cristian

Guilherme Lage Albano, Jezreel Neres Dos Santos
19.1.8055, 14.2.8370

Sistemas de Informação - Universidade Federal de Ouro Preto.
Instituto de Ciências Exatas e Aplicadas
João Monlevade - MG - Brasil

guilherme.albano@aluno.ufop.edu.br, jezreel.santos@aluno.ufop.edu.br

Abstract. This paper aims to describe Cristian's algorithm, which is a solution for clock problems in distributed systems. It also presents the application in a real scenario, showing the code implementation and the execution, matching different computer clocks.

Resumo. Este artigo visa apresentar o algoritmo de Cristian, que é uma solução para o problema de relógios em sistemas distribuídos. Também é apresentado sua aplicação em um cenário real, mostrando a implementação e a execução, coincidindo os relógios de diferentes máquinas.

1. Introdução

1.1 Sincronização e comunicação

Em sistemas distribuídos é crucial possuir comunicação e sincronização entre processos de forma consistente e estes são pontos de maior atenção, pois diferentemente de sistemas centralizados, é necessário garantir sincronia entre os dispositivos e realizar comunicação eficiente, não é aconselhável manter todas as informações sobre o sistema em um único lugar, pois tal prática é de sistemas centralizados. Em sistemas distribuídos, a comunicação é mais complexa e é necessário a utilização de algoritmos específicos neste processo, porque a informação está espalhada entre diferentes máquinas, permitindo que processo tomem decisões baseadas somente nos dados que possuem, permitindo a divisão de tarefas e melhor poder de processamento, porém, a complexidade de implementação se torna uma desvantagem, este artigo apresentará um dos problemas encontrados na implementação de sistemas distribuídos, a sincronização de relógios, e então, será apresentado uma das soluções desenvolvidas e sua implementação.

1.2 Sincronização de relógios

A comunicação de processos necessita de que exista sincronização entre os envolvidos, é comum que, com o passar do tempo, os relógios das máquinas fiquem diferentes um dos outros, e isto é um problema em sistemas distribuídos já que irá afetar a forma da comunicação, deixando-a desorganizada e causando falhas, para evitar estes problemas, existem estratégias de sincronização de relógios, neste artigo, descreveremos o algoritmo de Cristian, porém, existe também o algoritmo de Berkeley.

1.3 Protocolo NTP

O Network Time Protocol (NTP) é responsável pelo gerenciamento de data e hora em uma rede, seu funcionamento é com base no protocolo UDP sendo capaz de atender uma grande quantidade de clientes sem perder a confiabilidade e segurança.

O protocolo NTP possui seu funcionamento baseado em “stratum” em diferentes níveis, estes são servidores de tempo e quanto menor o nível do stratum, maior será a precisão de seu relógio. Stratum nível 1 são conectados diretamente a fonte de tempo que é o nível 0, e a cada nível sucessor K, ele é sincronizado com o stratum K-1.

De acordo com o website do NTP, a hierarquia dos estratos pode ser definida:

Em teoria, quanto mais perto da raiz, ou seja, do estrato 0, maior a exatidão do tempo. O estrato ao qual o servidor pertence é a principal métrica utilizada pelo NTP para escolher dentre vários, qual o melhor servidor para fornecer o tempo.[1]

1.4 Algoritmo de Cristian

O funcionamento do algoritmo consiste em um servidor que irá consultar a hora exata pelo protocolo NTP, este servidor será o responsável por responder as requisições das outras máquinas dentro de um sistema distribuído, as máquinas irão enviar uma requisição para obter a hora correta e então atualizarão seus relógios de acordo com a resposta do servidor.

O processo de enviar requisições e receber as respostas do tempo podem demorar tempos variáveis, podendo haver grandes diferenças de tempo, principalmente se o servidor for sobrecarregado. Outra desvantagem é que se trata de uma solução centralizada, para minimizar a centralização, podem ser criados vários servidores UTC e as máquinas realizam solicitações por mensagens multicast.

2. Implementação

A implementação do algoritmo foi dividida em duas partes, servidor e cliente, e a linguagem da implementação é o python, o arquivo de código do servidor será nomeado como server.py e o cliente como client.py .

O código final pode ser encontrado em: https://github.com/iamAlbano/Cristian-Algorithm

2.1 Servidor

O algoritmo do servidor será responsável por obter o horário correto por meio do protocolo NTP e então responder as requisições com este horário, para esta ação, será usada a biblioteca datetime para importar o datetime e o timezone, e a biblioteca ntplib para realizar a requisição ao protocolo NTP, sua instalação pode ser realizada através do comando “pip install ntblib”.

2.1.1 Algoritmo e Protocolo NTP

O servidor possui um algoritmo para obter o horário via protocolo NTP

```
import ntplib

from datetime import datetime, timezone

def getNTPTime():

  ntp = ntplib.NTPClient()

  response = ntp.request('pool.ntp.org', version=3)

  return datetime.fromtimestamp(response.tx_time)
```

Figura 1: Implementação do algoritmo com horário por protocolo NTP

Cria-se um cliente NTP e então realiza um *request* recebendo o horário correto e o retorna.

2.1.2 Servidor

Implementado o algoritmo de obter horário pelo protocolo NTP, o próximo passo é criar o servidor utilizando sockets.

Cria-se um socket através de seu construtor e então se utiliza o bind com a porta do servidor, no caso está sendo utilizada a porta padrão 8000. Após configurado, se inicia o processo *listen*, para receber as requisições.

```
# Cria o socket
  s = socket.socket()
  # Define o endereço e a porta do servidor 8000
  s.bind(('', 8000))
  # Função para ouvir as conexões
  s.listen(5)
```

Figura 2: Criando e configurando o socket

Com o socket criado, o próximo passo é iniciar a recepção de requisições por um loop na qual o servidor irá receber as chamadas e responder com a hora obtida pelo protocolo NTP da função construída anteriormente.

```
# Loop infinito para aceitar conexões
  while True:
    # Aceita conexões com clientes
    connection, address = s.accept()
    print('Server conectado com ', address)
    # Envia a hora atual para o cliente
    conexao.send(str(getNTPTime()).encode())
    # Fecha a conexão com o cliente
    connection.close()
```

Figura 3: Laço para receber e responder requisições

Ao receber uma requisição, o servidor a aceita, armazenando informações do endereço da máquina solicitante e a conexão, em seguida, envia o horário através do resultado da função criada para obter o timestamp via NTP e fecha a conexão.

2.2 Cliente

Com o servidor funcionando, faz-se necessário a implementação do código do lado do cliente que irá realizar as requisições e obter o horário, a estratégia deve ser a mesma, utilizando o socket. Além do socket e datetime, será necessário importar o parser da biblioteca dateutil e o default_timer da biblioteca timeit e a biblioteca time.

```
import time

import socket

import datetime

from dateutil import parser

from timeit import default_timer as timer
```

Figura 4: Importações do código do cliente

2.2.1 Iniciando cliente

O primeiro passo para iniciar o cliente, é criar o código e configurá-lo, usando a mesma lógica do servidor, porém conectando, neste caso, ao servidor local e na mesma porta (8000).

```
# Algoritmo de Cristian - Cliente

def client():

  # Cria o socket

  s = socket.socket()

  # conecta com o servidor local na porta 8000

  s.connect(('127.0.0.1', 8000))
```

Figura 5: Criando socket do cliente

2.2.2 Recebendo horário do servidor

Uma das desvantagens citadas do Algoritmo de Cristian é o tempo de delay entre a requisição e a resposta, neste caso, iremos calcular esta diferença para ajustar ao relógio e realizar testes com os resultados. Para calcular este delay, primeiro, calcula-se o horário da realização da requisição criando um timer.

```
# recebe timestamp da requisição

  hora_requisicao = timer()
```

Figura 6: Calculando o horário da requisição

Logo após obter o horário atual, realizamos a requisição para o servidor, estes dois trechos de códigos devem ser sequências para não haver diferença de tempo entre elas e o horário calculado ser o mais preciso possível. Após o envio da requisição, realizamos outro timer, desta vez para obter o horário de recebimento da resposta. Outra informação importante a ser armazenada antes da atualização, é o horário atual do cliente, pois calcularemos a diferença entre o horário do cliente e o horário recebido do servidor, para então observar quantos segundos de diferença ocorreu entre o tempo de cada requisição.

```
# recebe a hora do servidor

  hora_ntp = parser.parse(s.recv(1024).decode())

# timestamp do recebimento da resposta

  hora_resposta = timer()

# timestamp do horário atual do cliente antes da atualização

  hora_atual = datetime.datetime.now()
```

Figura 7: Enviando requisição e armazenamento do horário de recebimento da resposta.

Perceba que é necessário utilizar o parser e o decode para conversão da resposta para um formato que fique melhor para trabalhar com os horários futuramente.

2.2.3 Calculando delay

Armazenada as informações de tempo, calcularemos o delay de tempo entre o envio da requisição e a resposta do servidor, basta subtrair o horário da resposta pelo horário da solicitação.

```
#delay é a diferença entre a hora da resposta e da requisição

  delay = hora_resposta - hora_requisicao

  print("Latência: " + str(delay) + " segundos")
```

Figura 8: Calculando o delay da requisição

2.2.4 Calculando horário do cliente

Após obter todos estes dados, calcularemos o horário do cliente, que será o horário respondido pelo servidor mais o delay, para realizar esta soma, utilizamos a função timedelta da biblioteca datetime[2]. O cálculo da hora do cliente é dado pela seguinte fórmula:

Hora do cliente = Hora do servidor + (segundos de delay/2)

```
# hora do cliente é a hora do servidor + o delay

  hora_cliente = hora_ntp + datetime.timedelta(seconds = delay/2)
```

Figura 9: Calculando o novo relógio do cliente

2.2.5 Diferença na sincronização

Ao passar do tempo, o relógio do cliente acaba por perder a sincronização com o servidor e por isso se faz necessário algumas consultas periódicas para atualizar o horário, vamos calcular a diferença entre o horário do cliente e o horário do servidor depois de cada requisição para saber o quão defasado estava o relógio do cliente. Para realizar isto, basta calcular a diferença do antigo horário do cliente com o horário recebido do servidor atualizado com o delay. Após isto, podemos fechar o socket pois não realizamos mais nenhuma ação. A função *abs( )* fará com que o valor seja sempre positivo.

```
# calcula a diferença na sincronização

  diferenca = abs(hora_atual - hora_cliente)

  s.close()
```

Figura 10: Calculando a defasagem do relógio do cliente

2.2.6 Executando chamadas periodicamente

Como dito anteriormente, com o passar do tempo, há uma defasagem no relógio do cliente e é necessário sincronizar novamente, para realizar esta ação, é necessário realizar outra requisição ao servidor. Para permitir que existam requisições regulares, utilizaremos a função *sleep* da biblioteca *time,* possibilitando que o código pare por um período de tempo e retornando a executar após isso, colocando a função dentro de um laço, podemos ter esta execução ativa enquanto o cliente estiver funcionando.

```
if __name__ == '__main__':

  minutos = 2 # Tempo em minutos para sincronizar novamente

  while True:

    client()

    # Espera 2 minutos para sincronizar novamente

    time.sleep(60*minutos)
```

Figura 11: Função principal que realiza as requisições no período de tempo definido

3. Testes

3.1 Introdução aos testes

Para iniciar os testes do algoritmo, executaremos localmente, a fim de verificar sua funcionalidade e então verificar possíveis melhorias. A execução local pode ser realizada abrindo o terminal no diretório do arquivo *.py*, o primeiro passo é executar o servidor pelo código “python server.py”.

Com o servidor executando, receberemos a mensagem “Servidor ativo para conexões” e então saberemos que ele está ativo e podemos realizar requisições. Com o servidor ativo, é possível realizar requisições iniciando o cliente com o comando “python client.py”

3.1.1 Resultados de execução local

O algoritmo foi executado por 30 vezes, neste tempo foi coletado os dados de delay e defasagem de hora em cada iteração, a fim de entender a média de tempo destes dados.

Os resultados obtidos foram 0.1346 segundos de média de delay das requisições e 0.5570 segundos de média de defasagem de relógio.

3.2 Testes em máquinas virtuais

O próximo teste executado será a comunicação entre máquinas virtuais que simulam máquinas distintas em um sistema e realizarão as requisições ao host. O software usado para criação das máquinas virtuais será o VirtualBox, e os sistemas operacionais das máquinas serão Ubuntu 22.04 LTS e Zorin 16.2. Os sistemas operacionais escolhidos já possuem o Python na versão 3 instalada de fábrica.

A configuração e criação da máquina virtual segue o padrão do VirtualBox, com a diferença que a rede das máquinas deve estar configurada em modo Bridge, possibilitando a comunicação com o Host que irá executar o servidor.

3.2.1 Configuração de rede

Para ser possível a comunicação com o servidor, será necessário alterar a conexão do socket, para isto, devemos alterar o localhost (127.0.0.1) na configuração do socket para o ip da máquina que executa o servidor.

```
# Cria o socket

 s = socket.socket()

 # conecta com o servidor local na porta 8000

 s.connect(('192.168.0.1', 8000))
```

Figura 11: Alterada a configuração do socket para conexão com o servidor

3.2.2 Testes na máquina Ubuntu

A primeira máquina virtual a ser testada será a com sistema operacional Ubuntu, executaremos o cliente e então deixaremos por 30 minutos realizando requisições e então, calculamos a média de delay e defasagem de horário.

Os resultados do teste mostram uma média de 0.1765 segundos de delay e 0.1839 segundos de defasagem de sincronização.

3.2.3 Testes na máquina Zorin

O segundo teste foi realizado em uma máquina virtual com sistema operacional Zorin, executando o código do cliente da mesma forma que na máquina virtual Ubuntu.

Os resultados foram semelhantes, mostrando uma média de delay de 0.1812 segundos e 0.1872 segundos de defasagem de sincronização.

Uma observação realizada durante a execução dos testes foi que a média de segundos de defasagem entre o horário do servidor e o horário do cliente tende a aumentar com o passar do tempo.

3.3 Atualizando relógio do sistema operacional

A próxima implementação será de atualização do relógio do sistema operacional de acordo com a data e hora retornada pelo algoritmo de Cristian, para realizar esta ação, será necessário uma nova função, responsável por receber o timestamp e então realizar os comandos de atualização no terminal do sistema.

3.3.1 Atualizando o relógio em sistema operacional Linux

Para realizar a atualização de hora nas máquinas virtuais criadas para representar os clientes, adicionaremos a seguinte função ao código do *client.py*.

```
def linux_set_time(timestamp):
  try:
    import subprocess
    import shlex
    time = ( timestamp.year, timestamp.month, timestamp.day,
      timestamp.hour,timestamp.minute,
      timestamp.second, timestamp.microsecond
      )
    time_string = datetime.datetime(*time).isoformat()
    subprocess.call(shlex.split("sudo timedatectl set-ntp false"))
    subprocess.call(shlex.split("sudo date -s '%s'" % time_string))
    subprocess.call(shlex.split("sudo hwclock -w"))
  except:
    print("Erro ao sincronizar o horário do sistema")
```

Figura 12: Função para atualizar relógio do sistema operacional

A função criada recebe o datetime e então o converte para uma tupla, a tupla então é utilizada para criar um novo datetime no formato adequado para realizar a mudança do relógio através da chamadas de subprocessos como usuário com permissionamento adequado.

Um ponto importante neste código é que o usuário que o executar deve possuir permissões de administrador para realizar a alteração, ao contrário, não será possível realizar a alteração.

3.3.2 Realizando chamada de alteração de relógio.

Com a função de mudança de relógio do sistema operacional criada, adicionaremos a condição de chamada deste algoritmo, como as máquinas virtuais são sistemas baseados em Linux, adicionaremos esta validação.

```
# hora do cliente é a hora do servidor + o delay

  hora_cliente = hora_ntp + datetime.timedelta(seconds = delay/2)

  # Atualizar horário do sistema operacional

  if sys.platform == 'linux2' or sys.platform == 'linux':

    linux_set_time(hora_cliente)
```

Figura 13: Realizando chamada da função de atualizar relógio

3.3.3 Atualizando relógio da máquina Ubuntu

Nesta etapa vamos realizar o teste do algoritmo de atualização do relógio do sistema operacional, a máquina teste será a com sistema Ubuntu, para testar, vamos desatualizar o relógio do sistema com uma data antiga.

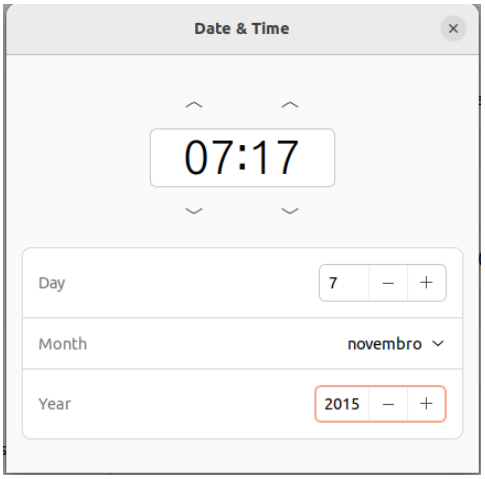

Figura 14: Alterando data e hora do relógio

A data de realização do teste é 18 de Março de 2023 às 11:57 e o horário da máquina virtual foi definido como 7 de Novembro de 2015 às 7:20.

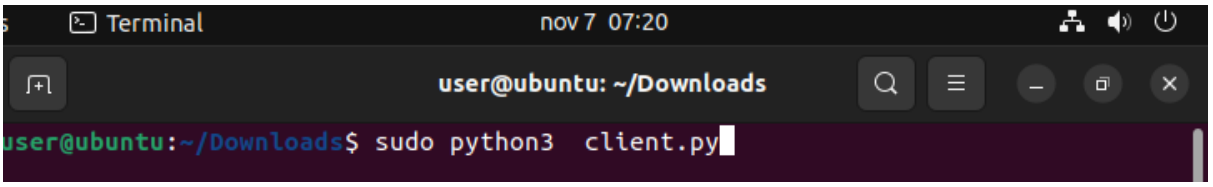

Figura 15: Relógio da máquina virtual com data e hora errados.

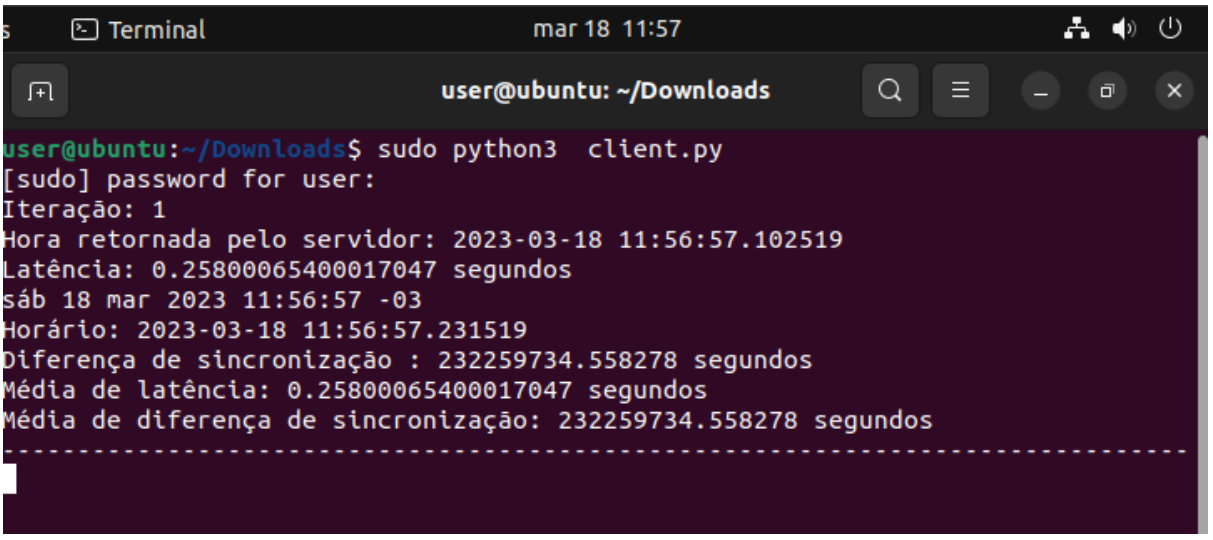

Figura 16: Relógio atualizado após realizar requisições ao servidor.

Neste caso, podemos verificar a funcionalidade do algoritmo de Cristian em um cenário mais próximo do real, simulamos um caso de sucesso de seu uso. Para a execução do cliente com permissão de alteração do relógio do sistema, foi dado as devidas permissões ao usuário e então inserida a sua senha , após isto, o algoritmo é executado com sucesso e atualiza o relógio periodicamente.

Enquanto o servidor estiver executando, o cliente é capaz de realizar requisições de atualização de relógio no intervalo que ele definir.

4. Conclusão

O algoritmo de Cristian pode ser uma solução viável para o problema de sincronização de relógios em sistemas distribuídos, mesmo com sua desvantagem de delay de requisição, que foi possível amenizar o impacto através do cálculo de tempo entre requisição e resposta. Uma das desvantagens no algoritmo implementado foi a centralização, pois existe um único servidor responsável por responder todas as requisições, esta abordagem pode ser eficaz em sistemas menores, porém possui impactos maiores em larga escala. A dependência do protocolo NTP acaba que por tornar o servidor menos instável, podendo falhar por erro externo e neste caso pode ser mais difícil manter a transparência do sistema, mas que foi contornada criando uma exceção no código python que retorna o horário local do servidor ao invés do horário NTP, evitando que o cliente fique sem resposta e a execução do servidor pare, causando uma possível queda.

5. Bibliografia

1. Algoritmo de Cristian e Berkeley. Acesso em 16/03/2023. Disponível em: https://prezi.com/4po8x0fvzp-m/algoritmo-de-cristian-e-berkeley/
2. Algoritmo de Cristian. Acesso em 16/03/2023. Disponível em: https://acervolima.com/algoritmo-de-cristian/

6. Referências

1. Arquitetura do NTP. Acesso em 16/03/2023. Disponível em: https://ntp.br/conteudo/ntp/.
2. Python | datetime.timedelta() function. Acesso em 16/03/2023. Disponível em: https://www.geeksforgeeks.org/python-datetime-timedelta-function/