# 物理时钟同步算法


> 作者：黎清兵
>
> 邮箱：3263109808@qq.com
>
> 学号：2120230678


# 1. 算法介绍

## 1.1 Berkeley 算法

**背景与用途**：
Berkeley 算法用于在局域网或集群环境中同步没有精准时钟的分布式节点。它适用于节点之间没有精确时间标准的情况，例如一组计算节点需要在内部保持一致的时间。

**工作流程**：

1. **时间采样**：选定一个节点作为协调者（master），该节点将询问其他节点（clients）的当前时间。
2. **计算平均时间**：
   - 各节点响应并返回它们的当前时间。
   - 协调者收集所有节点的时间数据，并计算平均时间偏差。
3. **时间调整**：
   - 协调者基于平均偏差计算出一个新的统一时间。
   - 协调者发送时间偏差调整信息给每个节点（而不是直接发送当前时间），各节点根据接收到的偏差来更新自己的时间。

**特点与应用**：

- 适合小型网络（如局域网、集群等），尤其是局域网内部一致性要求较高的分布式系统。
- 由于直接使用时间偏差进行同步，不依赖任何外部精确时钟，因此适合完全隔离的内网环境。

---

## 1.2 Cristian 算法

**背景与用途**：
Cristian 算法主要用于客户端-服务器模式下的时钟同步。它通常适用于一个或多个客户端需要与一个精确时间源（如时间服务器）同步的场景。

**工作流程**：

1. **请求时间**：客户端向时间服务器发送时间请求。
2. **时间响应**：
   - 服务器收到请求后立即返回服务器的当前时间。
3. **时间校准**：
   - 客户端在收到响应后，根据请求和响应的往返时间（RTT）估算网络延迟，并根据此延迟进行时间校准：

$$\text{时间校准} = \text{服务器时间} + \frac{\text{RTT}}{2}$$

   - 这里假设网络的传输延迟是对称的，即发送和接收的时间相等。

**特点与应用**：

- Cristian 算法适用于单一服务器与多个客户端的环境，尤其是网络延迟较小且对称的系统。
- 对于互联网等不对称的网络，其精度受网络延迟的波动影响较大。

---

## 1.3 NTP 算法

**背景与用途**：
NTP（Network Time Protocol）是目前广泛使用的时钟同步协议，支持跨广域网和局域网的大规模分布式系统。NTP 提供较高的精度，并支持多个层级的时间源结构。

**工作流程**：

1. **时间请求**：
   - 客户端向服务器发送时间同步请求，记录请求发送的时间（t1）。
2. **服务器响应**：
   - 服务器接收请求，并记录接收到请求的时间（t2）。
   - 服务器在响应中返回三个时间戳：t1（请求发送时间）、t2（服务器接收请求的时间）和 t3（服务器响应发送的时间）。
3. 时间计算：
   - 客户端接收到响应，记录到达时间为 t4。
   - 客户端根据以下公式计算时间偏移（offset）和往返时间（RTT）：

$$\text{offset} = \frac{(t2 - t1) + (t3 - t4)}{2}$$
$$\text{RTT} = (t4 - t1) - (t3 - t2)$$

   - 客户端使用计算得到的时间偏移调整自己的时钟。

**特点与应用**：

- NTP 可以实现毫秒级的时间同步精度，并适应互联网环境中的延迟波动。
- 支持分层次的服务器结构（即所谓的 NTP 层级模型），最高层级为一级服务器，直接与 GPS 等精准时间源同步，其他服务器通过级联关系同步到上一级服务器。
- 广泛应用于需要高时间精度和长时间稳定性的系统中，如金融、网络管理系统、分布式数据库等。

---

## 1.4 总结

- **Berkeley 算法**：适合小型封闭网络，没有精准时钟源时可用，通过计算平均时间进行同步。
- **Cristian 算法**：适用于单个时间源和多个客户端的同步，网络传输时间对称的情况下效果较好。
- **NTP 算法**：广泛应用的标准，适合互联网等大规模分布式系统，支持层级同步和复杂网络环境。

# 2. 实验实现

## 2.1 广域网模拟

#### 广域网模拟原理和实现

在 Linux 中，结合网络命名空间（namespace）和流量控制（`tc`）可以有效模拟广域网（WAN）环境。`namespace` 用于创建隔离的网络环境，而 `tc`（traffic control）工具则用于设置网络延迟、丢包率、带宽限制等，从而模拟出广域网的网络特性。以下是简要步骤：

1. **创建网络命名空间**：使用 `ip netns add` 命令为每个节点创建独立的命名空间。例如：

```
ip netns add node1
ip netns add node2
```

2. **创建虚拟网络接口对（veth pair）**：`veth pair` 是一对虚拟网络接口，用于连接不同命名空间，实现相互通信。将 `veth pair` 的一端连接到主机或桥接网络，另一端放入命名空间。例如：

```
ip link add veth-node1 type veth peer name veth-host1
ip link set veth-node1 netns node1
```

3. **设置 IP 地址**：为 `veth` 接口分配 IP 地址，以便各命名空间之间可以通信。例如：

```
ip netns exec node1 ip addr add 10.0.0.1/24 dev veth-node1
ip netns exec node1 ip link set veth-node1 up
ip addr add 10.0.0.2/24 dev veth-host1
ip link set veth-host1 up
```

4. **设置路由规则**：在命名空间中配置必要的路由规则，以保证所有节点可以通过适当的网络路径通信。
5. **使用 `tc` 进行网络模拟**：在每个 `veth` 接口上使用 `tc` 添加网络延迟、丢包率、带宽限制等。例如：

```
ip netns exec node1 tc qdisc add dev veth-node1 root netem delay 50ms loss
1% rate 1mbit
```

   这里，`netem` 模块用于设置网络延迟（50ms）、丢包率（1%）和带宽限制（1 Mbps）。
6. **验证配置**：可以使用 `ping` 或 `iperf3` 等工具验证节点之间的网络连接是否符合设定的广域网特性。命令示例如下：

```
ip netns exec node1 ping 10.0.0.2
```

#### make_env.py

上述流程比较繁琐，因此我写了一个 python 脚本来自动化进行虚拟集群的构建。在配置文件中描述集群的节点以及延迟和丢包等指标，然后运行脚本就可以得到一个全连通的虚拟集群。除了上述的`网络命名空间`、`veth pair` 之外，为了实现简单的全连接，`make_env.py` 里面还创建了一个虚拟的网桥来进行节点的联通。

`make_env.py` 在项目的 `README.md` 里面有详细介绍。

## 2.2 算法实现

#### 算法抽象

各算法的处理实现虽然不一样，但是都可以抽象为 server 端和 client 端的方法。因此，可以使用一个抽象类来定义处理接口：

```
class ClockSyncAlgorithm(ABC):

    @abstractmethod
    def server_process(self, name, sock: socket.socket, num_client: int = 3) ->
None:
        """
```

```
        Server processing logic for the algorithm.
        :param name: node name
        :param sock: UDP socket bound to server address
        :return: None
        """
        pass

    @abstractmethod
    def client_process(
        self, name, sock: socket.socket, server_ip: str, server_port: int
    ) -> dict:
        """
        Client processing logic for the algorithm.
        :param name: node name
        :param sock: UDP socket for sending requests
        :param server_ip: Server IP address
        :param server_port: Server port
        :return: dict for t1, t2, t3, t4, offset, rtt
        """
        pass
```

各个算法在派生类中实现处理逻辑。

**算法实现**

在实现之前，有几个关键点需要注意：

1. **准确度的计算**
   - 对于 `Berkeley` 算法，由于其同步结果是所有节点时间的平均值，可以根据系统时间计算准确度。
   - 对于 `Cristian` 和 `NTP` 算法，节点的时间以 `master` 时间为基准。因此可以通过计算 `client` 和 `master` 之间的时间差来评估同步的准确度。

   但是，**在真实的分布式系统中，我们无法直接知道 `master` 的"绝对准确时间"**（否则也不需要同步了）。
   不过在单机模拟的环境下，我们可以**通过系统时间来获取 `master` 的准确时间**，从而计算出 `client` 与 `master` 的时间差，以评估算法的准确性。
2. **时钟偏差的模拟**

   另一个问题是，我们不会真正去修改系统的时间。那么如何模拟出每个 `client` 的时间变化呢？我们可以引入一个变量 `cumulative_offset` 来表示每个 `client` 的累计时间偏差。
   - 每次获取 `client` 的当前时间时，使用 `系统时间 + cumulative_offset` 来模拟当前的 `client` 时间。
   - `master` 的时间直接使用系统时间，而 `client` 的时间则是系统时间加上 `cumulative_offset`，同步算法会通过调整 `cumulative_offset` 来模拟校准效果。

# 3. 实验

我进行了三个设置下的实验。实验设置如下：

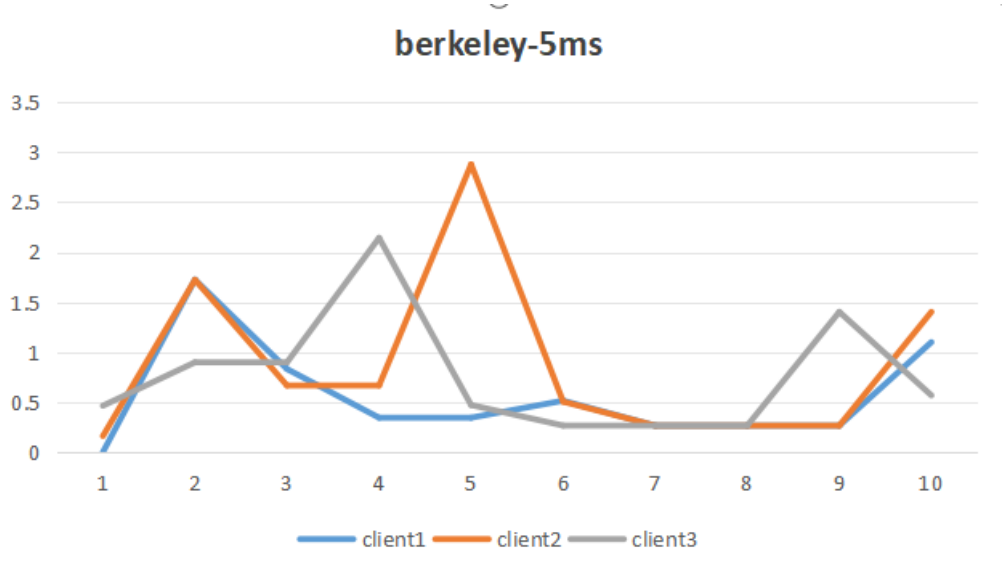
- 四个节点
- 网络延迟分别在 5ms、50ms、200ms
- 丢包率为 5%

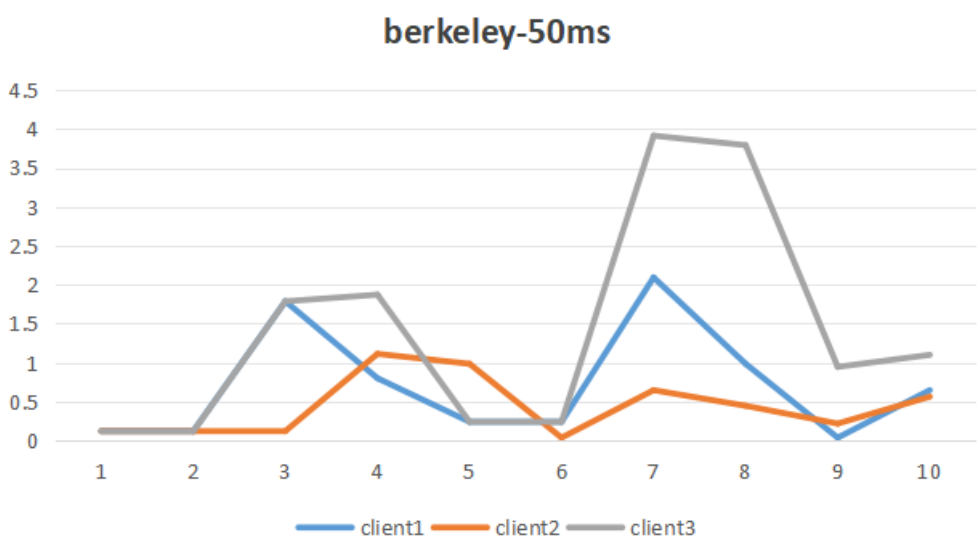
每种算法都计算了各 client 时间和系统时间的偏差。

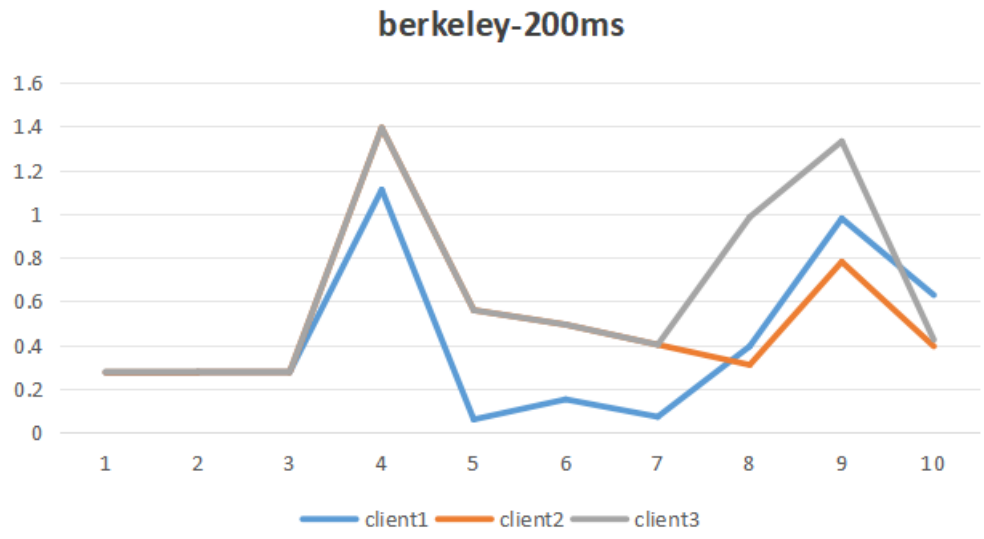
## 3.1 实验结果

- 折线图横坐标为不同的迭代，纵坐标为节点时间和系统时间的偏差
- 由于模拟集群的往返时延是对称的，算法在不同时延下并没有表现出明显差异
- 从图表可以看出，`Berkeley` 算法的准确度是低于 `cristian` 和 `NTP` 的
- `cristian` 和 `NTP` 并没有明显差异

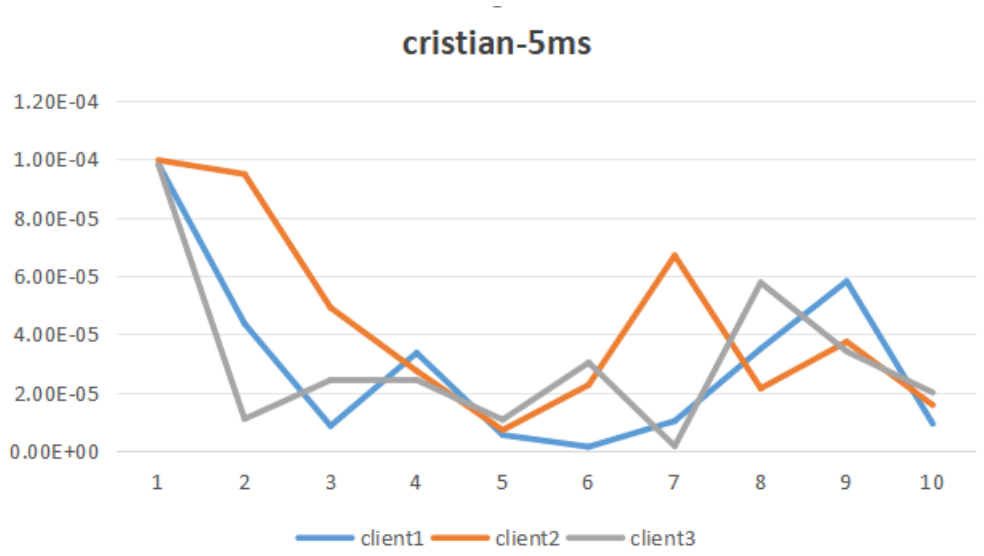
**Berkeley 算法**

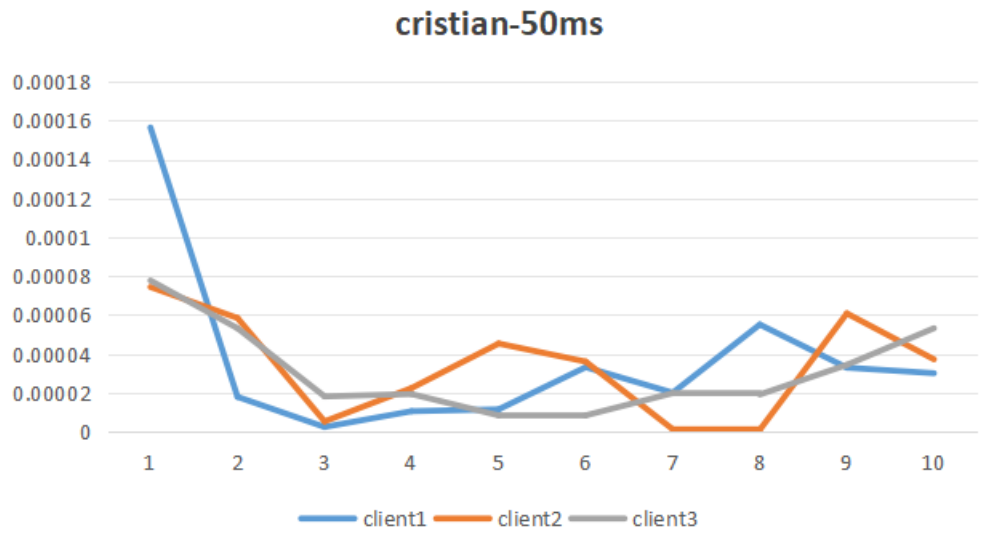


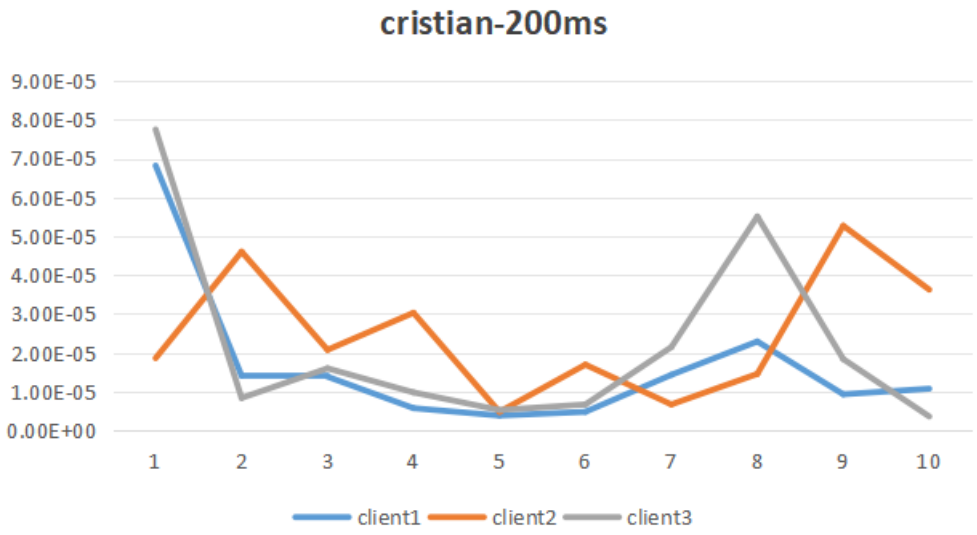




berkeley-200ms
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
client1
client2
client3

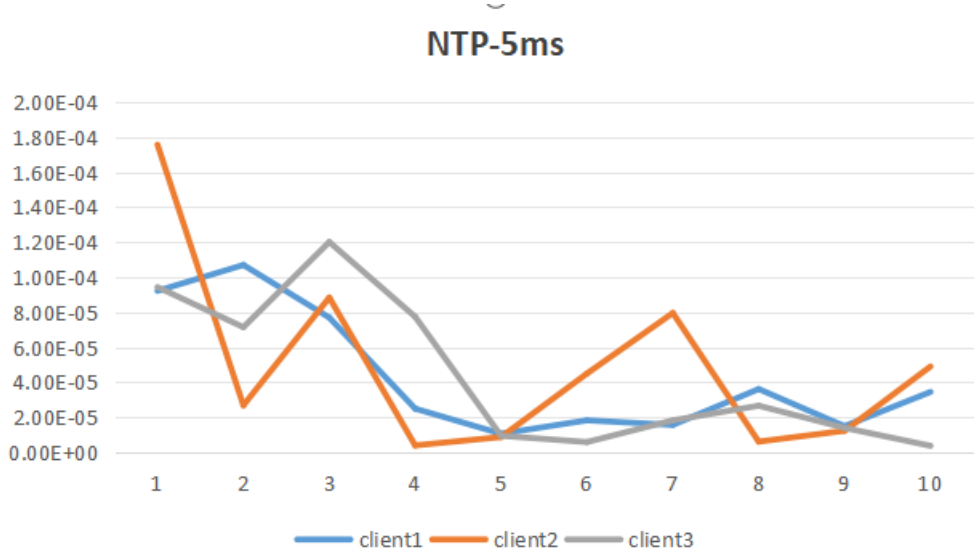
**Cristian 算法**

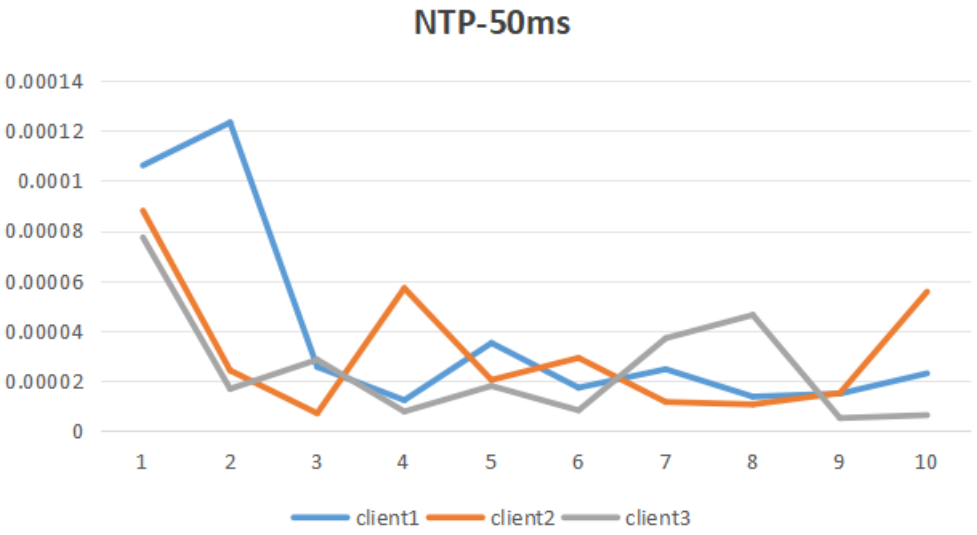

cristian-5ms
1.20E-04
1.00E-04
8.00E-05
6.00E-05
4.00E-05
2.00E-05
0.00E+00
1
2
3
4
5
6
7
8
9
10
client1
client2
client3

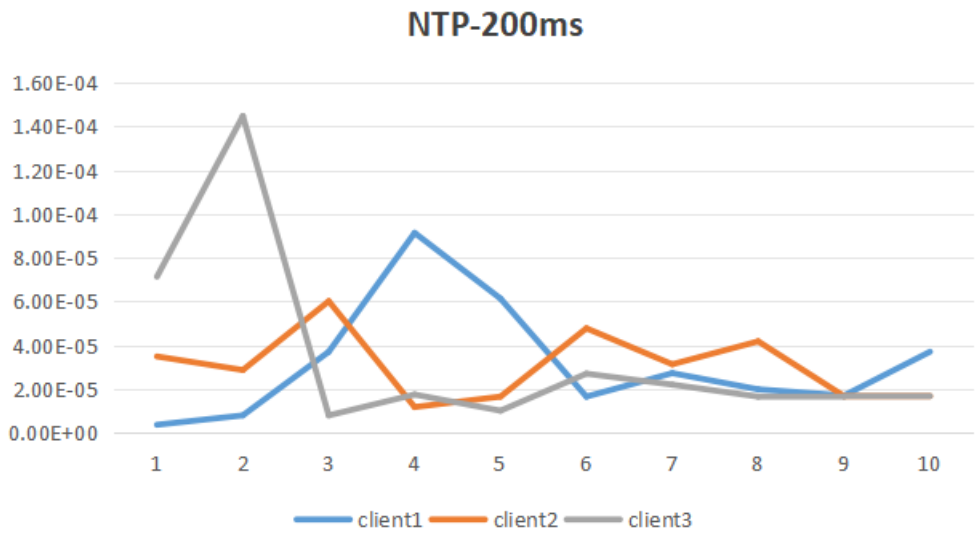

cristian-50ms
0.00018
0.00016
0.00014
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
1
2
3
4
5
6
7
8
9
10
client1
client2
client3


cristian-200ms
9.00E-05
8.00E-05
7.00E-05
6.00E-05
5.00E-05
4.00E-05
3.00E-05
2.00E-05
1.00E-05
0.00E+00
1
2
3
4
5
6
7
8
9
10
client1
client2
client3

**NTP 算法**


NTP-5ms
2.00E-04
1.80E-04
1.60E-04
1.40E-04
1.20E-04
1.00E-04
8.00E-05
6.00E-05
4.00E-05
2.00E-05
0.00E+00
1
2
3
4
5
6
7
8
9
10
client1
client2
client3


NTP-50ms
0.00014
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
1
2
3
4
5
6
7
8
9
10
client1
client2
client3


NTP-200ms
1.60E-04
1.40E-04
1.20E-04
1.00E-04
8.00E-05
6.00E-05
4.00E-05
2.00E-05
0.00E+00
1
2
3
4
5
6
7
8
9
10
client1
client2
client3