Enhancing the performance of NTP

TEAM: NEXUS

Sneha Maraliga Jayaram (maraliga@usc.edu)
Aaditi Ganesh Pai (aaditigp@usc.edu)
Praval Panwar (ppanwar@usc.edu)
Arjun Sharma (arjuns@usc.edu)
Lavish Singal (lsingal@usc.edu)
Varsha Venugopal (varshave@usc.edu)

***Abstract*—This paper presents a technique for network time synchronization, which is an improvement over the legacy Network Time Protocol (NTP). NTP is a time synchronization protocol, used to distribute time information over large Internet systems and ensure synchronization among them. This paper presents solutions to two of the main drawbacks of NTP, kernel jitter and asymmetric delays. The solutions provided are kernel timestamping and employing node-to-node synchronization respectively. It describes strategies to make the current time synchronization more precise. Finally, this paper compares the offsets measured by our solution and NTP.**

Keywords— NTP, kernel timestamping, asymmetric delay, offset.

I. INTRODUCTION

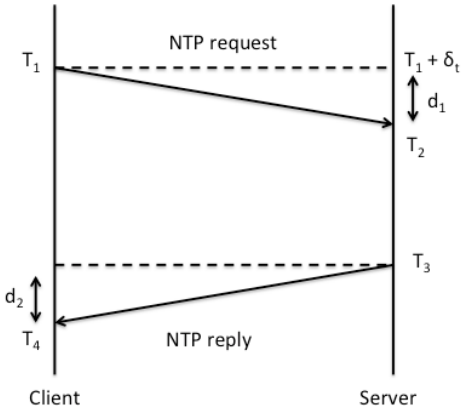
Network Time Protocol (NTP) is a widely used Time Synchronization protocol, which synchronizes time for systems in a distributed network to Coordinated Universal Time (UTC). It is used for networks of all types and sizes as the importance of precision and accuracy is increasing by the day. It uses a dynamic server discovery scheme to find the appropriate server to synchronize with. NTP has a few drawbacks associated with it, which are asymmetric delay and kernel jitter.

This paper presents solutions to these drawbacks, which in turn results in lower offset. Precise time coordination requires precise send and receives timestamps of messages exchanged between the server and the client. This paper presents a software only implementation, which instead of time stamping in the application layer, timestamps at the driver level, which is as close as possible to the hardware.

This paper is organized as follows: Section II is a brief introduction to NTPv4. Section III is a brief introduction to Linux clock system. Section IV is the implementation details. Section V presents the test results comparing the offsets deduced by our implementation and legacy NTP, Section VI includes the conclusion, Section VII includes the IEEE journals, papers and other manuals.

II. NTPv4

In NTPv4, every equipment revises its own time by sending a request to a server at a higher level, which acts as a time source, through a network. It determines the appropriate server to connect to, using an algorithm called as the Marzullo's algorithm. The basic working of NTP is shown below:



NTP client sends a NTP request message to the server for synchronization and records the timestamp of the sent packet as T_1. The server when it receives this packet records the receive timestamp as T_2, adds the timestamp to the packet, processes it and then sends it recording the send timestamp as T3 and adding this to the packet as well. This is called the NTP reply packet. This is received by the client which records the receive timestamp and calculates the offset according to the following formula:

$$\text{offset} = ((T_2 - T_1) + (T_3 - T_4)) / 2 \quad (1)$$

An important thing to note here is that NTP assumes the propagation delays between the server and the client to be the same, i.e., it assumes that the paths taken by the NTP request and the reply packets are the same hence causing deterioration of the precision by several milliseconds. Also, when jitter occurs, the measurement of time difference value becomes difficult because NTP does not take into account the difference caused by the one-way delay of requests and replies.

III. Brief Introduction To Linux Clock Systems

Any Linux system has two clocks, real time clock and kernel clock. The real time clock(RTC), which is also called the hardware clock is on even when the machine is off. Kernel clock, which is also called the software clock, is updated based on the hardware clock or by communicating with the timeserver when connected to the Internet. The time is continuously updated in accordance with the network timeserver if the system is connected to an Internet connection and hence, the RTC can be completely ignored.

When the system is rebooted, the kernel clock is reset according to the hardware clock and from then on, the two clocks runs independently and hence the synchronization with network timeserver is lost. The operating system is responsible to keep the clocks in sync, but the intervals at which the sync is done should be less, as it would affect the performance otherwise.

IV. Implementation

The two problems of NTP are handled in our implementation as two different parts. The first part deals with minimizing the kernel jitter. The second part handles the asymmetric delay problem.

A. Kernel Timestamping

The transmission of an NTP packet happens by the server capturing the time using a system call like *gettimeofday()* at the user level. However, this system call introduces a kernel jitter as the system call itself consumes some amount of time to traverse through the kernel and back. Hence our implementation employs a technique to minimize the kernel jitter caused by *gettimeofday()*. Minimizing the kernel jitter is done using the *ioctl* system call where the time the packet leaves the kernel level is captured. This capturing of time at the kernel level is done by using *ktime_get_real*() function and storing it in the socket buffer or SKB. The socket buffer, or "SKB", is the data structure that is mostly used in Linux networking code. Every packet sent or received is handled using this data structure. It records the time at which a packet is sent or received in the stamp variable of the type *struct timeval*.

The programs where the modification has been done along with the code snippet is as follows:

a) Modification for receiving

The program for capturing time while receiving a packet is e1000_main.c. The below snippet shows the function in e1000_main.c and the modification:

```
static bool e1000_clean_rx_irq(struct e1000_adapter
*adapter, struct e1000_rx_ring *rx_ring, int
*work_done, int work_to_do)
{
  ...
  process_skb:
  ...
        skb->tstamp = ktime_get_real();
        ...
}
```

b) Modification for transmission

The program for capturing time while sending a packet is dev.c. The below snippet shows the function in dev.c and the modification:

```
static int __dev_queue_xmit(struct sk_buff *skb, void
*accel_priv)
{
  ...
        skb->tstamp = ktime_get_real();
        ...
}
```

Following the modifications in the kernel, the timestamp recorded in the sk_buffer should be retrieved by the user-level program. Linux 2.6.30 or later versions provide a new API for the user space to get the network packets time stamps. Using the *ioctl* system call along with the socket options by using *setsockopt*(), the timestamps are recorded using SO_TIMESTAMP and SO_TIMESTAMPING. Different parameter settings for function setsockopt() can achieve different return results.

The socket options for receiving network packets timestamps are:

- SO_TIMESTAMP - Generates a timestamp for each incoming packet in system time. Reports the timestamp via *recvmsg*() in a control message as *struct timeval* (*usec* resolution).

- SO_TIMESTAMPNS – It works in a similar mannar as SO_TIMESTAMP, but reports the timestamp as *struct timespec* (*nsec* resolution).

- IP_MULTICAST_LOOP + SO_TIMESTAMP [NS]
  Only for multicast: approximate transmit timestamp obtained by reading the looped packet receive timestamp.

- SO_TIMESTAMPING
  Generates timestamps on reception, transmission or both. Supports multiple timestamp sources, including hardware. Supports generating timestamps for stream sockets.

Capturing the timestamp from the kernel using these APIs gives the time at the user level. These times are used to calculate the values required for the delay and the offset formulae.

The exchange of packets for calculating the offset closely follows the legacy NTP. The implementation includes creating a custom NTP packet, which consists of four timestamps, T_1, T_2, T_3 and T_4. T_1 is the time at which server sends the packet to the client. This time is captured at the kernel level using the *ioctl* system call mentioned previously. T_2 is the time at which the packet is received on the server side. For the packet arrival time to be as accurate as possible, it should be captured as soon as it is received by the server at one of the lower levels and hence is captured at the driver level using the *ioctl* system call. The client sends T_2 and T_3 to the server. T_3 is the time recorded at the user level using the *gettimeofday()* system call. The time at which the server receives the client's packet is recorded as T_4.

Using these time values - T_1, T_2, T_3 and T_4, the delay is calculated using the below formula:

$$Delay_{client\text{-}server} = (T_4 - T_1) - (T_3 - T_2) \quad (2)$$

After calculating the delay, the server sends the custom NTP packet to the client. The contents of this packet include the time recorded using *gettimeofday()* and the delay calculated earlier. Upon receiving the server's packet, client updates its clock using the *settimeofday()* system call. This results in the server and client clocks to be synchronized.

The below formula is used to calculate the exact time for *settimeofday()*:

$$mastertime - T_{3jitter} + RTT/2 + Recv_{jitter} \quad (3)$$

$Recv_{jitter}$ and $T_{3jitter}$ are calculated using the formulae below:

$$T_{3jitter} = (Delay_{client\text{-}server}) + (T_4 - T_1) + (T_2 - T_{3kernel}) \quad (4)$$

$$Recv_{jitter} = T_{user} - T_{kernel} \quad (5)$$

On the client side, the previously mentioned formula (4) is used to remove the kernel jitter which occurs as a result of using the *gettimeofday()* API. Similarly, to remove the kernel jitter between the times we received the custom NTP reply packet at kernel and user we use (5).

The offset is calculated every time there is an exchange of packet between the client and the server using these values and the below formula:

$$offset = \frac{((T_2 - T_1) + (T_3 - T_4))}{2}$$

B. Asymmetric delay

The drawback of Network Time Protocol is that the server assumes that the packet takes the same route from server to client as it originally took from the client to the server, which might not be true at all times. Since the aim is to keep the master and client in sync, our implementation considers the idea of node-to-node synchronization.

The idea is that the master server will synchronize all its neighboring nodes with stratum1. This chain will further synchronize its neighboring nodes using the same method acting as master server for the stratum2 nodes..

The topology that we implement the above idea is as below:

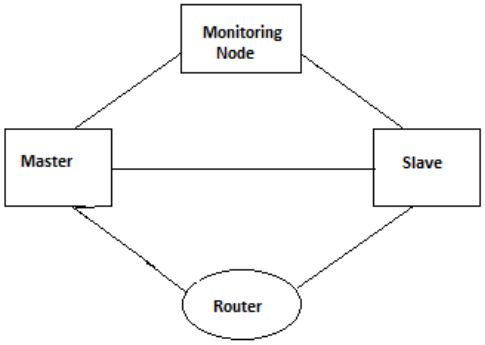


Fig 1. Topology used for implementation

In the above topology, the master and the router first exchange packets as described above and after calculating the networking delay and other parameters, the router uses the *settimeofday()* system call to adjust its clock accordingly. This results in the master and the router to sync within each other. The router then becomes the master to the slaves directly connected to the other end and the communication between router and slave takes place in a similar manner and they get synchronized. As a result, the master and the slave finally get synchronized.

V. Results and Validation

A. *Results*

The offset using the custom NTP was calculated in order to show the results for both the problems – node to node and considering asymmetric delay.

The offset calculation for node to node was done for different scenarios by varying the delay on the link and the delay values introduced were 0ms, 10ms, 50ms, 100ms, 200ms. The statistics for these five delay values are as shown below:

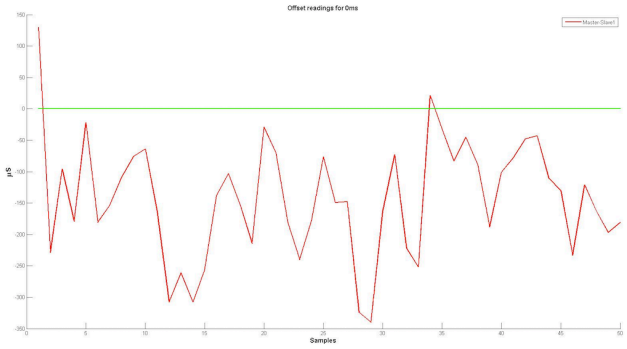


Fig. 2. Offset values between two nodes with delay of 0ms

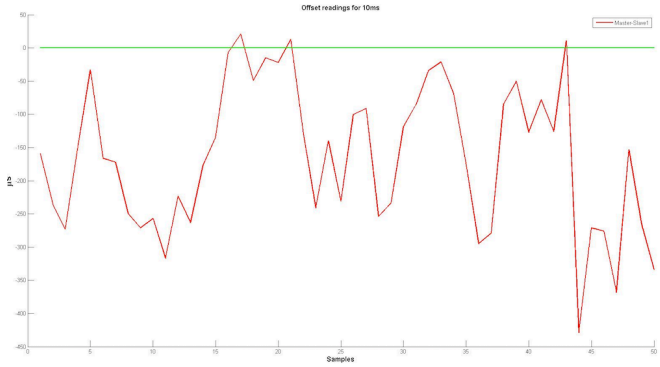


Fig. 3. Offset values between two nodes with delay of 10ms

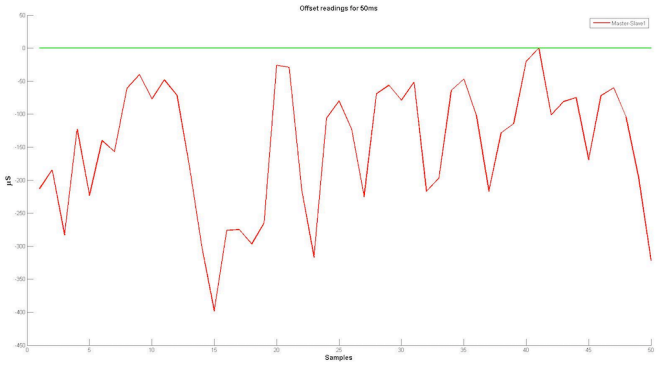


Fig. 4. Offset values between two nodes with delay of 50ms

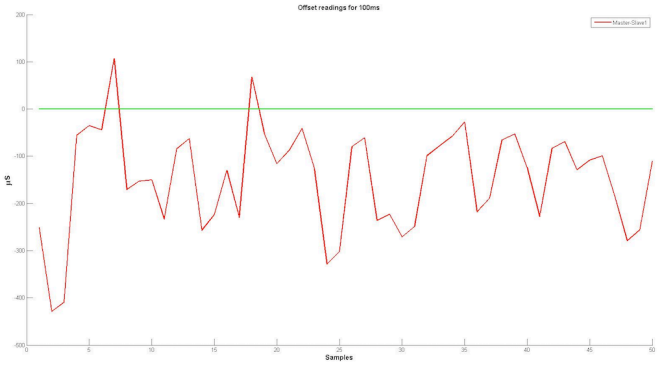


Fig. 5. Offset values between two nodes with delay of 100ms

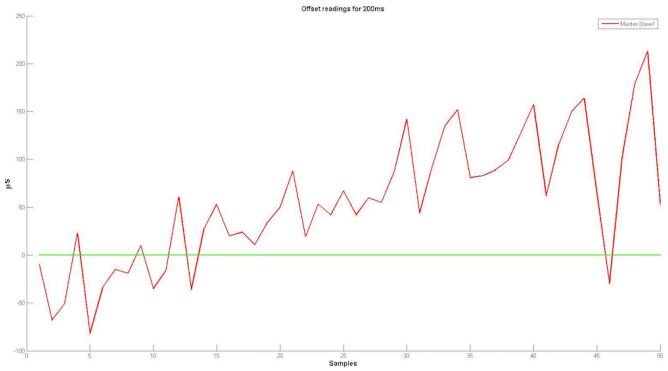


Fig. 6. Offset values between two nodes with delay of 200ms

The offset was also calculated for the topology shown above for the same delay values. This was to consider the asymmetric delay problem in NTP. The idea is to prove that the offset between two nodes with a direct link would be the same as the offset calculated by considering the router in between the two nodes, which is not the case in the standard implementation of NTP. This is because our implementation synchronizes every router in the route between the server and client and thus ensures end-to-end synchronization as a result of which the offset calculated would be the same as the offset calculated between two nodes.

The statistics for the offset values for the above shown topology are as below:

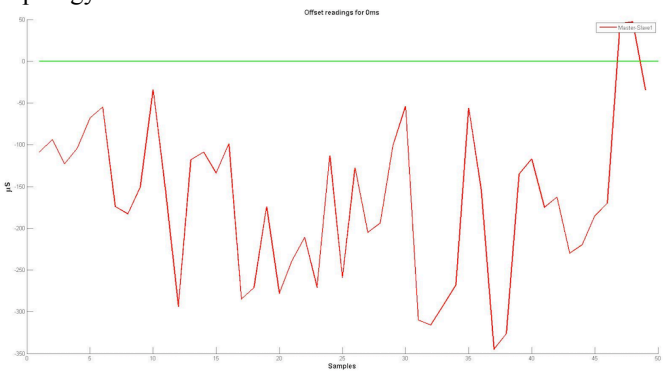


Fig. 7. Offset for topology with a delay of 0ms on each link

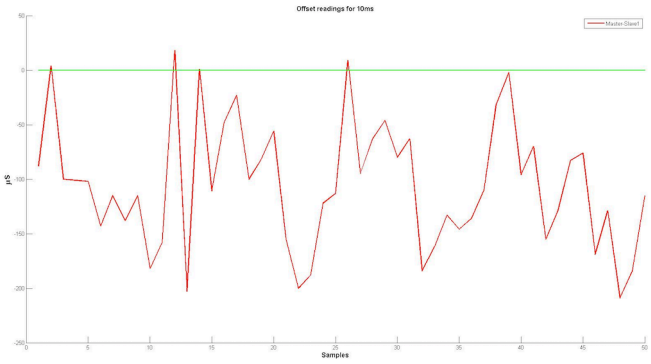


Fig. 8. Offset for topology with delay of 10ms on each link

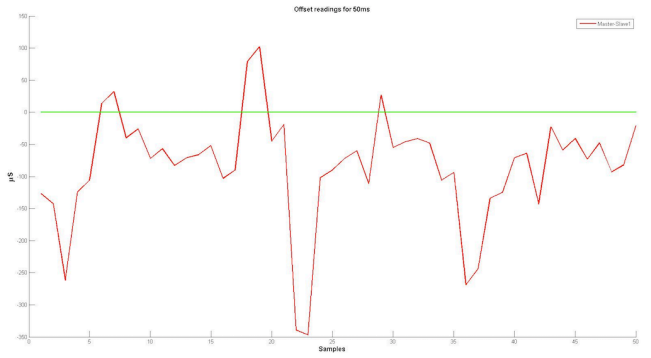


Fig. 9. Offset for topology with delay of 50ms on each link

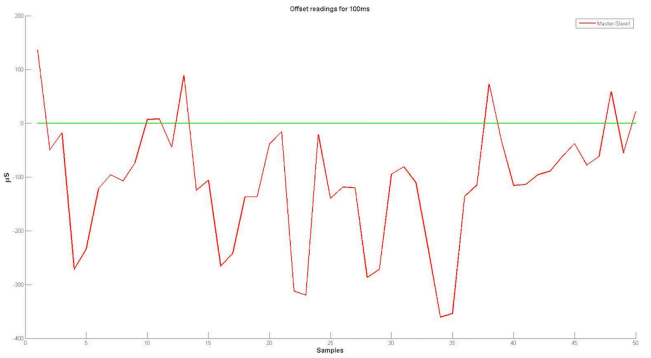


Fig. 10. Offset for topology with delay of 100ms on each link

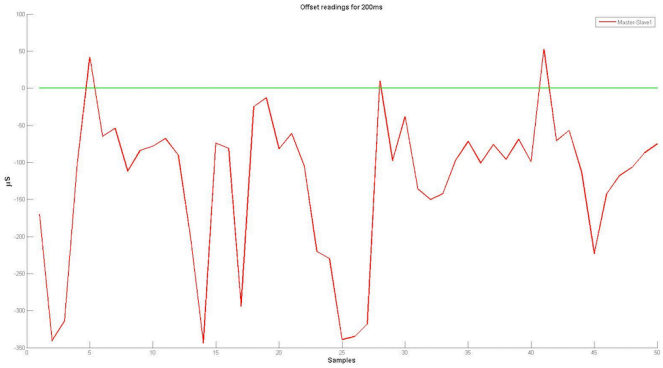


Fig. 11. Offset for topology with delay of 200ms on each link

B. *Validation*

Validation with standard NTP implementation is done by using a monitoring node, which is synchronized with a stratum 0 server. After the client and server are synchronized using the custom NTP implementation, the offset between the client and monitoring node is checked against the offset between the server and monitoring node. These two offsets that are calculated are almost the same, which proves that they are synchronized. Additionally, the offset calculated is compared with the offset recorded by legacy NTP and the result is that the offset calculated by custom NTP is lesser than the legacy NTP. The below graphs show that the offsets between the server and monitoring node and the offsets between the client and the monitoring node are overlapping. This means that the offsets are the same with respect to a different server.

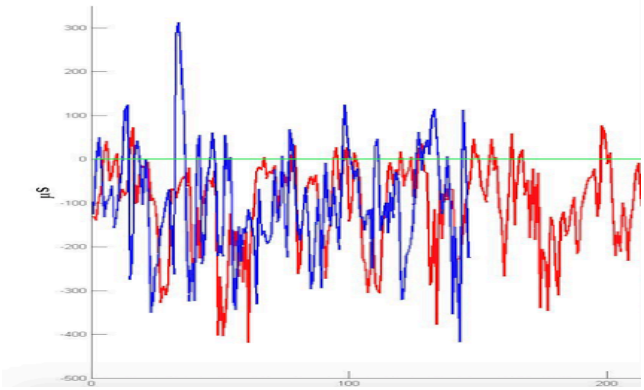


Fig. 12. Offset overlap for validation using a monitoring node

C. *Performance measurements*

The difference between the offsets (one between master to monitoring node and other between slave to monitoring node) were calculated and plotted as a graph against the legacy NTP.

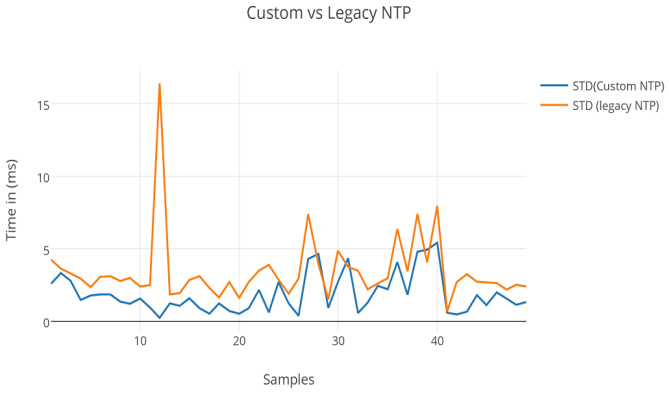


Fig. 11. Graph showing standard deviation between offsets calculated considering asymmetric delay

VI. Conclusion

As seen from the results and validation above, our custom Network Time Protocol improves the offset as compared to the legacy NTP and also gives better results by considering asymmetric delays which legacy NTP does not. This paper develops the custom NTP which is based on NTP to satisfy more and more high-efficiency and high-accuracy requirement in the network applications.

VII. References

[1] David L. Mills. "Network Time Protocol Version 4 Reference and Implementation Guide" NTP Working Group, Technical Report 06-6-1

[2] Sun-Mi Jun, Dong-Hui Yu, Young-Ho Kim, Soon-Yong Seong. "A Time Synchronization Method for NTP", IEEE 1999

[3] Kendall Cornell, Nick Barendt and Michael Branicky. "Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol"

[4] Yoshiaki Kitaguchi, Akihiko Machizawa, Masato Tsuru, Hiroyuki Fukuoka and Katsuya Hakozaki. "Research of advanced time synchronous system with network support", 2003 IEEE

[5] HE Peng, PENG Ruiqing, YUAN Wenxue, GUO Min, ZHAO Bin. "The Optimization Techniques for Time Synchronization based on NTP", 2009 IEEE

[6] Peter Orosz, Tamas Skopko. "Software-based Packet Capturng with High Precision Timestamping for Linux", IEEE 2010

[7] Robert Love. "Linux Kernel Development", 3rd edition

[8] Prof. Hans Weibel and Dominic Bechaz. "Implementation and Performance of Time Stamping Techniques"

[9] Zhi Li, Zhenlin Zhong, Wangchun Zhu, Binyi Qin. "A Hardware Time Stamping method for PTP Messages Based on Linux System"