Best Practices in Modern Timing Infrastructure

Matt Sherer
FSMLabs

In an ideal world, accurate time would be everywhere. Applications anywhere can request the current time, get an answer with zero overhead and zero error in the response.

We don't live in that world. Time sources can and do fail. They simply stop serving time, or begin broadcasting bad data. Substandard protocol implementations introduce noise in the signal. Network links go down. Attacks against infrastructure can compromise timing signal integrity. Even in an otherwise well operated production environment, minor temperature variations introduce detectable noise in the distributed time.

Put simply: It's easy to get unverified and inaccurate time to a large number of network clients.

Getting reliable, verifiable, and accurate time everywhere isn't a huge task. There are pitfalls, and sometimes those are difficult to see, but they are well known. TimeKeeper is built to avoid as many as possible out of the box. The goal of this document is to identify what you should be able to expect out of your timing infrastructure, and how to get there with a minimum of effort.

First we'll define some common terms, and from there we can discuss the best ways to serve time, the best ways to receive time. Then, we'll cover what may be the most important step - proper validation techniques.

Laying the Groundwork

Before going into the specifics of how to best distribute time on your network, it's worth framing the problem and some common terms, and describe some common concepts.

Client and server can have different meanings in different contexts. A client of the network timing infrastructure can be viewed as a server by those using the machine to do work. In

Keys to a Reliable Timing Infrastructure

Following the guidelines in this whitepaper will result in a time infrastructure that is:

Resilient

Failures in time sources will not affect client time sync.

Verifiable

Reported client sync accuracy is provably correct.

Servers collect and validate client sync performance.

Accurate

Consumers of time get the best possible accuracy with a minimum of technology.

Accuracy better than 1 microsecond is not only achievable, but commonly expected.

Low Overhead

Fast access to time makes an accurate time signal actionable.

this document, a server is a system providing time on the network, and a client is a consumer of that time. Clients are downstream from servers, servers are upstream from clients.

A GPS clock is a time source derived from the GPS signal, and can be found in a variety of forms. Some have high cost backups internally, such as a rubidium oscillator, but a GPS clock can be as simple as a 'hockey puck' antenna with a serial port output. They deliver time in various forms, which TimeKeeper can consume and distribute on the network.

Similarly, a CDMA clock synchronizes time against a cellular network signal. While less accurate than a GPS input, a CDMA clock can function where the GPS signal cannot reach.

Once a reference signal is obtained, whether it's from GPS or a CDMA input, products like TimeKeeper distribute that time accurately on the network over the PTP or NTP protocols. Given one or more GPS signals, TimeKeeper can synchronize timing clients to well within 1 microsecond of the reference GPS signal, whether the client is synchronizing over PTP or NTP. Given a CDMA signal, TimeKeeper can synchronize clients generally to within 10 microseconds of true GPS time. The quality and degree of this synchronization is what you'll see described as sync accuracy.

In this document, we'll be looking at distribution of time to clients over general purpose networks using PTP (the IEEE 1588 specification) or NTP. PTP version 2 is primarily considered – PTP version 1 is supported, but isn't very common on modern networks. Hardware-specific time distribution, like IRIG, is not something we'll consider here – adding in specialized hardware tends to be space and cost prohibitive in large deployments. With today's server densities, there are no means of putting additional hardware in every machine, from either a cost or space perspective.

However, a PPS, or Pulse Per Second, is worth mentioning here. TimeKeeper can use a PPS to accurately receive or transmit time as a client or server, but again, as machine densities increase, it's difficult to deliver a PPS to every client in a space-sensitive datacenter. A PPS is a useful tool, though, to validate that a client's reported time matches the server. Later sections of this document will cover this in detail.

'Multisource' is a term that will come back again and again. Because of all of the potentials for failure, it's important to ensure clients have multiple paths to get their time. This can mean multiple network paths, and/or multiple protocols. TimeKeeper can track multiple timing sources over multiple networks, regardless of protocol, and this is what we mean when we mention 'multisource.'

Distributing High Quality Time

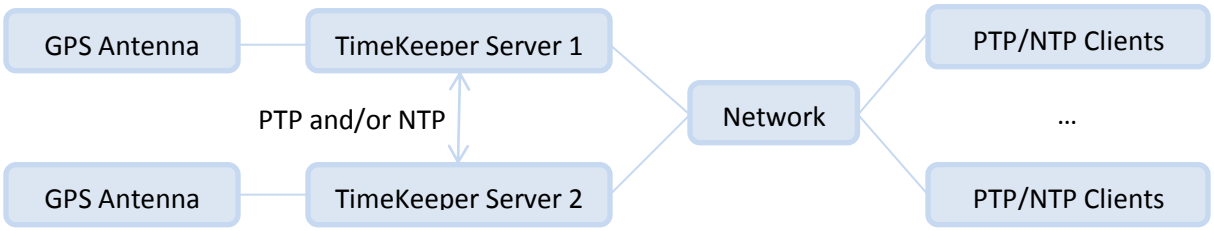
Before clients can get accurate time, proper precautions must be taken on the server side to ensure the timing data on the network is accurate and reliable. So the first step is to construct a server with the ability to take in accurate inputs and be ready to distribute that time reliably and accurately.

Select Time Sources

Providers of time are nearly always clients of a sort also – even if they are receiving time from an upstream source like the GPS constellation. An exception is when the server has no reference input but instead delivers its own time as authoritative. Here, we'll describe a server that provides GPS-derived time as a best practice, although it is fairly common for a number of machines to need to be synchronized to each other but not aligned with an external time source like GPS. TimeKeeper can support either scenario.[1]

Depending on the location, GPS may or may not be an option. It is the preferred time source in most environments if a signal is receivable in the datacenter or access to a roof mounted antenna is an option. The GPS signal can be brought in via a Symmetricom or Spectracom card mounted in the server, or via other receiver cards. Even a very simple device like a Garmin antenna with a serial output can be used to get the timing signal to the server.

Having at least two reference inputs to the server is ideal. This way, the GPS input and another signal can be continuously compared, and if one fails, the other will provide uninterrupted service to any clients. If two GPS inputs are not an option, it is still worth having a secondary time source to fall back to if the GPS input is lost. Below is an example diagram of what this configuration might look like, where the 'Server' is the multisource server we're discussing. The secondary GPS antenna and TimeKeeper server, if not available, could be substituted with an existing PTP grandmaster:



If a direct GPS signal is not an option, CDMA can provide a reasonably accurate time source. If neither GPS or CDMA are available, the server can consume existing NTP and PTP on the network, and deliver that to downstream clients. This is a stratum server (in NTP terminology) or boundary clock (in PTP terminology), and can be used as a means to deliver a reliable time signal from deeper in the network than a single top level server.

It is worth noting here that TimeKeeper delivers time better than many existing PTP and NTP servers. So, even if you already have these protocols on your network, having TimeKeeper clean up the signal and rebroadcast the same protocol can improve client sync accuracy. As a test of existing infrastructure, FSMLabs encourages users to consume PTP and NTP from their current servers and any new grandmasters under test, at the same

[1] To have TimeKeeper use itself as an authoritative time source, configure a primary source that looks like this:
SOURCE0() { PPSDEV=self; }
With this defined, TimeKeeper will consider its own time as authoritative.

time. Comparing the PTP and NTP output from the same system at the same can show two very different concepts of time.

If your intended time source is a remote NTP (or PTP) server, deploying a local NTP/PTP server that relays that onto the local network as NTP and/or PTP is usually a good solution. In this configuration, TimeKeeper can model the noisy link to remote server, remove the noise, and provide a clean local signal for your clients to consume.[2]

We'll cover the specifics of how to validate your source inputs below.

Select Distribution Medium and Capabilities

Bringing in a perfect GPS signal to a server won't do any good if it can't be delivered to clients in a relevant way. Two primary factors guide how the server will distribute time, regardless of which protocols are used.

The first factor is medium - how will the server provide time to clients? We mentioned earlier that hardwired time delivery to all clients is generally not an option, although delivering a PPS to clients can be viable in some environments. This leaves a variety of network connection types. Make sure that the server has the option of matching your network fabric, whether it is 1/10/40Gb Ethernet, Infiniband, or anything else.[3] A server won't provide time to clients if it can't connect and keep up with the network.

TimeKeeper can make use of any hardware available to Linux, so if you need to deliver time over a 10G optical connection to your network, simply bind the server to that network interface. This avoids the need to build special network paths between the common network and the timing server. As the network changes over time, the server's connection to the network can change as well – TimeKeeper is flexible in this regard.

The second factor is timestamping capabilities, which are important both on the server and the client. Here, on the server, timestamps are used to tag received and sent data. NTP and PTP both use those timestamps, although in different ways. By default, software-derived timestamps will be captured by TimeKeeper automatically. These software timestamps will be augmented with hardware timestamps (again, automatically) if the hardware and driver support the feature and provide valid timestamp information. This removes some uncertainty from the timing protocol data that is put on the wire, resulting in a better sync.

As above, TimeKeeper will automatically make use of and drive the hardware timestamping features if they are found to be available. FSMLabs can provide guidance in selecting the best-fit timestamp aware hardware for specific network configurations.

[2] TimeKeeper will track and relay public NTP data for local clients, but this is not recommended except for FINRA/OATS compliance. Public NTP servers are commonly diverge from GPS time by many milliseconds.

[3] If the output of the server can't match the rest of the network, a downgraded network path to get to the server may be an option. If the network is 10Gb but the server is limited to 1Gb connectivity, a special extra hop may be added for clients to get to the server. However, this is not recommended and TimeKeeper does not add this restriction on deployments.

Select Time Distribution Protocols

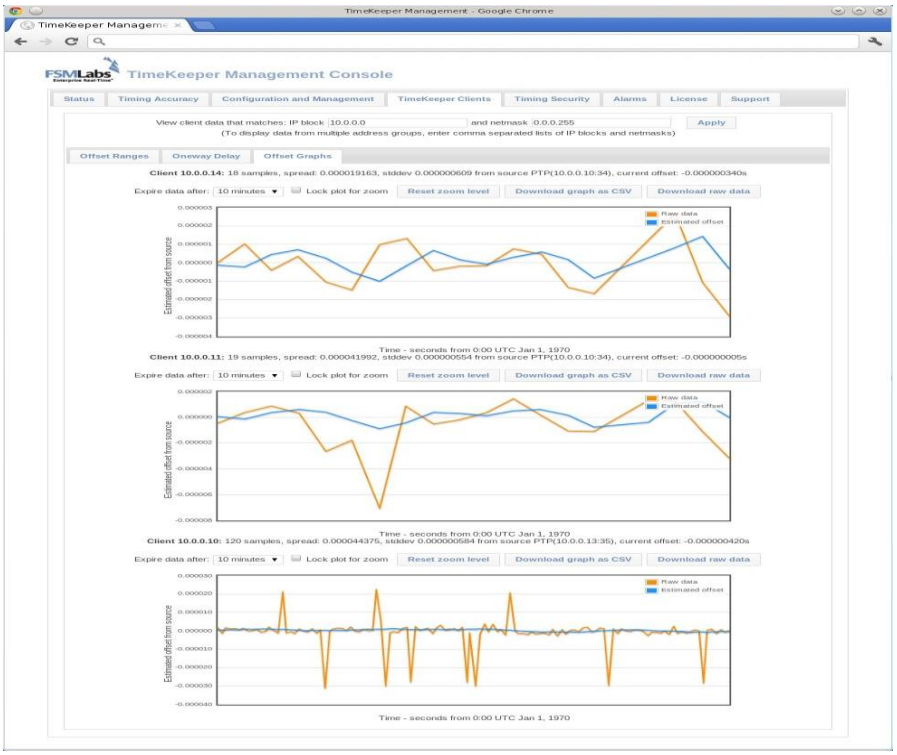
Once a server is selected, the reference signals selected, and network connectivity chosen, the next step is to determine how to deliver time to clients. NTP or PTP? PTP is a popular choice now, but not always for the right reasons. Strangely, it's not generally known that NTP can achieve the same quality sync as PTP in many situations, and sometimes can outperform PTP.[4]

The real story is that there is nothing wrong with the NTP protocol, but many NTP implementations are low quality and will deliver a low quality timing signal on the network. Even with a high quality client NTP server will wobble if it's tracking a noisy upstream NTP source as it wobbles. With low quality NTP clients, a noisy server isn't noticeable because the client is also noisy. NTP, implemented well on both client and server, can deliver sub-microsecond-level accuracy to network clients, just like PTP. It also does very well over long distances.

PTP can make better use of modern hardware timestamping capabilities due to the nature of the protocol, but in controlled networks behaves similarly to a high quality and modern NTP implementation. For longer distance links, PTP offers a unicast mode. For the highest accuracy sync, though, multicast PTP is commonly the best path compared to unicast.

PTP with TimeKeeper does offer improvements over NTP by providing the additional ability to query clients and report back how well clients are syncing to the server. With this feature, it's easy to see how all of the network clients are doing from a single location. However, if NTP is preferred as a primary source, TimeKeeper clients can report back their NTP sync accuracy using the PTP protocol. Pictured below is an example of how TimeKeeper can quantify client behavior live, as it happens – it can also visualize changes in PDV (Packet Delay Variation – in essence, the variability in the round trip time seen between the client and the server) across different network segments, among other things:

[4] This topic is also covered with more detail here: http://www.fsmlabs.com/blog/choosing-between-ptp-and-ntp



In the end, both NTP and PTP can do very well on a local network. For longer distances, both protocols can operate in unicast mode and provide a good sync. Which one should you use? The answer in many cases is: Both! A server providing GPS time over both NTP and PTP allows clients to track either protocol, or both simultaneously. Allowing clients to primarily track PTP but fall back to NTP offers protection against PTP protocol failures, issues with the protocol itself, multicast and port issues, and many other possible problems. Similarly, with two servers, having clients track PTP from one server and NTP from another provides protection against network/wire and protocol issues and server failures at the same time.

Here is one way to configure TimeKeeper as a server to provide both protocols - two PTP servers on two domains, bound to different network interfaces, and NTP serving across all interfaces:

```
SERVENTP=1
SERVEPTP0() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=0; IFACE=eth0; }
SERVEPTP1() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=1; IFACE=eth1; }
```

Deploy your Servers

Once the above is configured, deploy the servers on your network. One is a minimum of course - but placing servers at different locations on your network for redundancy increases resiliency. The redundant servers can broadcast on the same PTP domain or each use their

own specified domain, as long as the clients know what to listen for. In the case of NTP, the clients just need to know the IP/DNS name of each server.

Let's go back to the above diagram where we have two servers, each with an authoritative GPS input. Each server will take in GPS time and the other's PTP/NTP data. Both will provide PTP and NTP to clients, and assume each one has two network interfaces of some type.

The first server's configuration looks like this:

```
# server1
SOURCE0() { PPSDEV=symmetricom; }
SOURCE1() { PTPDOMAIN=2; PTPCLIENTVERSION=2; }
SOURCE2() { NTPSERVER=server2; }
SERVENTP=1
SERVEPTP0() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=1; IFACE=eth0; }
SERVEPTP1() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=1; IFACE=eth1; }
```

The server tracks the GPS input first, and should that fail, it will track PTP on domain 2 (to be provided by the second server). Should both fail, it will also attempt to track NTP from server2. It also serves NTP, and explicitly serves PTP over two interfaces on PTP domain 1.

The second server's configuration looks similar:

```
# server2
SOURCE0() { PPSDEV=spectracom; }
SOURCE1() { PTPDOMAIN=1; PTPCLIENTVERSION=2; }
SOURCE2() { NTPSERVER=server1; }
SERVENTP=1
SERVEPTP0() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=2; IFACE=eth0; }
SERVEPTP1() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=2; IFACE=eth1; }
```

Here, the server tracks a GPS input that comes in via a Spectracom card. If that fails, it will fall back to PTP from server 1, and worst case, track NTP from server1. Like server1, server2 provides PTP to any clients via two PTP instances, each one bound to a specific interface. It also delivers NTP to any client that requests it. The Ethernet devices could be bonded if desired – in that case, the interface that gets used is going to depend more on the bonding configuration specified in Linux.

In any of the failure scenarios, clients tracking these servers via NTP or PTP will not suffer any interruptions in service. We'll discuss client configurations later, but they are nearly identical to these servers.

Deploying Boundary Clocks

With a large network, delivering time from a single source can be difficult:

- Too many hops may prevent multicast PTP data from reaching clients

- Many networks with varying traffic patterns will reduce quality in the remote client sync
- There simply is too much load for a single server

NTP provides the concept of a stratum server, while the PTP specification defines a boundary clock. There are differences between the two, but both act as a relay point for time on the network. Rather than having all clients go back to a single source, downstream clients get time from this intermediate clock, and this clock gets its time from the upstream source. This reduces load on the central server and reduces traffic on the common networks.

An added benefit here is that the boundary clock can filter out network jitter just on the path between it and the upstream server, whether that server is authoritative or another boundary clock. The alternative is to make the client filter out all of the noise on all of the networks between it and the server. With multiple networks, each with their own varying load patterns, it becomes difficult for the client to accurately filter the true timing signal. A boundary clock and stratum server can serve a stable time reference downstream by limiting the amount of jitter upstream.

Boundary clocks 'push time out' throughout the network, rather than centralizing, and each of these boundary clocks can identify secondary and tertiary inputs in case their link to the upstream clock is lost. If this happens, downstream clients continue to receive data, whether it is PTP or NTP, without any interruption in service.

Leverage Existing Network Infrastructure

With TimeKeeper's ability to run on Arista switches, many boundary clocks can be installed throughout the network. Every Arista switch on the network can act as a timing server to its clients, with redundancy built in so that clients never go without good timing data.

Configuration of TimeKeeper on Arista switches is handled the same way as on any other system. Sources are defined in order of preference to bring in time to the switch, and PTP/NTP is configured to deliver time to downstream clients. As with other servers, defining multiple inputs ensures that clients suffer no interruption in service if one of the upstream time sources disappear.

If existing hardware cannot act as a boundary clock, addition of new machines throughout the network is an option, although not preferred. TimeKeeper can also run on other switches – if you have specific concerns about your infrastructure, contact sales@fsmlabs.com for more details.

Common Pitfalls

Providing multiple inputs to the server is very important. Sources do fail, and it's important to have backups, just like you would have a backup power supply on a server. With

TimeKeeper, identifying a backup is as simple as adding another configuration line pointing to another server. This ensures that downstream clients never see an interruption in service.

Relying on one network path can also be a problem. The PTP server configuration above showed how to serve PTP on two separate interfaces, so that if one link goes down, the other is still functioning for users, whether they are end clients or boundary clocks. Not specifying the interface will cause any PTP/NTP traffic to use the system's default network interface. This will function but offers no protection if that default network path is lost. As mentioned, Ethernet bonding can augment any timing failover scenarios if desired.

Lack of reporting is a concern - if a time source goes offline and a backup is used, it's important that someone is notified. TimeKeeper allows the use of SNMP traps or email notifications to alert operators of any issues.

Network hardware selection is up to the user, and depends mostly on what non-timing requirements specify. Simple but deterministic switches can be preferable to feature heavy hardware. As with anything else, testing and validating is important to quantify network behavior.

FSMLabs can provide guidance required in selecting a server and supporting hardware for just about any environment.

Ideal Configurations

We've already seen some configuration scenarios above detailing how to prepare servers in a failover configuration. Let's look at a couple variations of the above in a little more detail.

First, here is a configuration with two GPS inputs to a server, one via a Symmetricom PCI card and one via a Spectracom device, with both cards configured to track the GPS input:

```
SOURCE0() { PPSDEV=symmetricom; }
SOURCE1() { PPSDEV=spectracom; }
```

With the above, TimeKeeper will track the first source as primary, and model the second to be ready in case the Symmetricom card reports loss of sync. With that, let's add NTP serving to the configuration:

```
SERVENTP=1
```

If there are multiple primary servers with GPS inputs, a good failsafe step, as we saw above, is to set up each server to act as a client to the other on a secondary or tertiary source. This way, each server is ready to cover for the other.

Assuming the server has 4 network interfaces that are intended to deliver time to clients, adding redundant PTP (serving the same domain on multiple interfaces) is a matter of adding these lines:

```
SERVEPTP0() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=0; IFACE=eth0; }
SERVEPTP1() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=0; IFACE=eth1; }
SERVEPTP2() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=0; IFACE=eth2; }
SERVEPTP3() { PTPSERVERVERSION=2; PTPSERVERDOMAIN=0; IFACE=eth3; }
```

Configuring a boundary clock is no different than the above scenario. A TimeKeeper boundary clock is no different than a primary server, except instead of getting time from a GPS input and backup sources and serving PTP and/or NTP, it gets time from upstream PTP and/or NTP sources and uses that signal to serve PTP and/or NTP.

Notifying operators of any issues, like the loss of a source, requires the identification of any SNMP trap receivers. This requires a line like:

```
SNMPTRAPHOST=10.0.0.110,10.0.2.110
```

Additionally, the SNMP OID can be named if desired. If email is preferred, SNMPTRAPHOST can be disabled in favor of the EMAILNOTIFICATION option, which lets the user specify a comma separated list of email addresses.

Receiving High Quality Time

Once the server is constructed, receiving accurate time from multiple sources, distributing time on the network, it's up to the clients to make good use of that timing data. Here we'll discuss how to make that happen.

Select Medium and Capabilities

As on the server, the client needs to be able to use the right type of network connection without the need for specialized hardware. Anything that is supported by Linux and matches your other needs will do in general. Redundancy in network connections is ideal if possible. There is no hardware lock-in on the client side – as the network requirements change, the network interface should be free to change as well.

Similar to the server, timestamping capabilities should be considered if the best possible sync accuracy is desired. Hardware timestamping will offer a more accurate sync if available – without the feature, software-based timestamps will be automatically used. FSMLabs can provide guidance through the hardware timestamping device selection process.

Protocol Configuration

Selecting protocols on the server side makes client selection a simple process, but there are factors to consider, mostly around choosing multiple sources. Assuming the dual server configuration described earlier, the client configuration looks like:

```
SOURCE0() { PTPCLIENTVERSION=2; PTPDOMAIN=1; IFACE=eth0; }
SOURCE1() { PTPCLIENTVERSION=2; PTPDOMAIN=2; IFACE=eth1; }
SOURCE2() { NTPSERVER=server1; }
SOURCE3() { NTPSERVER=nist1-ny.ustiming.org; NTPSYNCRATE=0.1; }
```

Timekeeper selects a default NTP sync rate of one request every 1.1 seconds, but this can be increased or decreased based on need. A sync rate of 0.1 is one request every 1/10 of a second, or one request every 10 seconds. (Similarly, the PTP sync rate on the server can be increased or decreased.)

With the above configuration, the client makes use of multiple links back to the server, assuming eth0 and eth1 follow different paths (network segmentation is beyond the scope of this description). There are 4 clock models being tracked, and if the PTP server isn't available on one interface, the backup path will be used. If the PTP server goes offline entirely, this client will fall back to the first server's NTP server, and beyond that, a NIST NTP server at a slow sync rate (one request every 10 seconds) until the PTP servers and/or local NTP server comes back online.

With multiple sources, not only is there protection against network failures and clock failures, there's also an audit log demonstrating client sync quality against a public time source over time.

Common Pitfalls

When using PTP as a client, it's tempting to rely just on the BMC (Best Master Clock) algorithm, which, given multiple PTP servers on the network, allows the client to choose the best fit. If a server goes offline or changes its self-reported accuracy[5], the client uses the BMC algorithm to select a new one.

The BMC algorithm is helpful, but insufficient. It leaves the problem of multiple network paths to be implicitly handled behind the scenes somewhere, which may or may not be the best path when disaster strikes. The specific behavior depends on the state of the network at the time of the failure, and may cause unintended and cascading load problems even in what starts out as a partial outage.

Relying solely on the BMC compromises the client's ability to track both multicast and unicast PTP sources, and does not allow failover to other time sources, like an NTP server. Using the BMC may work out fine, but if there is a plan for redundancy and failover, it's better to make the backup plan explicit where possible. This explicit failover is what TimeKeeper's multisource feature is designed to provide. Specific PTP grandmaster failover selection can be identified, or the failover can switch protocols to account for multicast networking issues or lost network links.

TimeKeeper makes use of the BMC algorithm to select the best server for a domain, but it's very important to be able to track PTP servers on other domains and NTP servers that

[5] This also assumes that clocks are correct in their self-determined accuracy determination.

may live anywhere. Explicit naming of which interface will provide each source leaves no ambiguity about what will happen if a link is lost. Above all, if an upstream source fails, the client should be able to go elsewhere for time.

Trust but Verify

At this point, you've got at least one server tracking an authoritative time signal, and a number of clients tracking that server via at least one protocol, preferably two, ideally over multiple network paths.

Now what? It's likely that the clients are reporting a decent sync, and obviously the server is operational if the clients are getting data. If you're the trusting type, you're done. We recommend against that strategy.

The sync accuracy numbers you are seeing, whether they are from TimeKeeper or any other product, are subject to the 'printf() delusion'. Software tracking time pulled from the network, even with hardware timestamping network cards, may do its very best to achieve a nanosecond-accurate sync, believe it has done so, and report success. The problem is that there's no verification done against a reference signal. There's no way to prove that the software hasn't unknowingly calculated in a bias, or the oscillator on the hardware timestamping card isn't wandering off.

Verification is the most important step in constructing a trustworthy timing infrastructure, but it’s this step that is commonly skipped, once clients are seen to be tracking a server. There are a number of ways to validate whether the timing is correct, and we’ll walk through a couple of methods below.

Easy First Steps

A quick (but to be clear, not exhaustive) validation involves just looking for biases between servers, networks, and protocols. Assuming there are two servers, each serving both NTP and PTP, create a client to track all 4 sources. In TimeKeeper, this can be achieved with the following configuration, assuming you have localserver1 serving NTP and PTP domain 1, and localserver2 serving NTP and PTP domain 2:

```
SOURCE0() { PTPCLIENTVERSION=2; PTPDOMAIN=1; }
SOURCE1() { PTPCLIENTVERSION=2; PTPDOMAIN=2; }
SOURCE2() { NTPSERVER=localserver1; }
SOURCE3() { NTPSERVER=localserver2; }
```

Be sure to name network interfaces with the IFACE keyword if it applies to the client’s configuration. Start TimeKeeper so it begins to model all four time sources.

This will create several data files on the TimeKeeper client, including one file for each source. TimeKeeper will sync to and track SOURCE0 first. Once the clock is synced, the TimeKeeper data file and plotting tools will show a stable smoothed offset and a less stable

raw offset, representing the local time offset from SOURCE0. A line from that file might look like this:

1341409532.584137941 0.000000478 -0.000002117 29022179087873 …

At the time in column one (recorded in UTC seconds/nanoseconds since the start of the epoch), TimeKeeper had a smoothed offset of 478 nanoseconds (column 2) from of the server's GPS-derived time. Timekeeper is also modeling the clock sources for sources 1-3. By looking at the offset data at the same time for the other sources, it's easy to see if there are any biases in the other sources. It may be that the two PTP sources agree, but the settled offset for the NTP sources disagrees by microseconds or milliseconds (sometimes, the difference can even be multiple seconds). This is a quick way to identify problematic NTP servers. Compare the PTP and NTP signals of some of the common grandmasters on the market, and you can see large differences in reported time, and that's when the signal comes from the same source!

It may be that the sources from the first server agree with each other, and the sources from the second server agree with each other, but both are offset from each other by, 20 microseconds. This may indicate some network asymmetry in the link between the client and the second server or in the link between the client and the first server. As we mentioned above, this still in software, and we have not compared the client's time against a reference signal from each server to identify the source of the bias.

The timing data can be viewed with TimeKeeper's management GUI, pictured below. This tool allows you to look at the sync quality of multiple time sources at the same time and see how each behaves, so that you can identify any offsets. Here there are two local sources, each derived from the GPS signal, but showing a bias between the two. TimeKeeper's tools allow you to identify these kinds of problems with a minimum of effort:

Of course, it may be that all four sources agree - but even in this case, we still haven't verified against a reference signal. All four sources may be biased in the same way by the same amount. Let's look at comparing against a reference signal next.

Verifying with a Reference PPS (external to the client)

One of the best tests is to have the client output a PPS at the top of the second, and compare that to a PPS from the server. Any offset between the two indicates a bias between the client and the server, which may be caused by network asymmetries, biases in the protocol implementations, operating system or driver issues, or other factors.

Before determining where the problem is, let's look at how to compare these signals. TimeKeeper ships with a simple tool for generating a PPS as needed - it resides in the /opt/timekeeper installation with the rest of the product. It's called 'ppsgen', and it is a normal Linux kernel module that emits a PPS at the top of each second.

The tool defaults to using the DTR/RTS pins on the machine's serial port for the PPS. This is a simple way to trigger I/O without specialized hardware, to a reasonable degree of accuracy – about 1.2 microseconds. If a serial port isn't present, or a more accurate I/O interface is available, source to ppsgen is provided so that the PPS can be directed at any other output available on the system.

Once the module is loaded, it will output a PPS on the serial line (or anything else) that can be routed back to either the server for comparison, or to an oscilloscope.

Compare this offset over time as the server and client vary in load, and as the network traffic patterns vary through the day. This will ensure that the client sync quality doesn't vary as the network gets busy, or when the client heats up due to added processor load, or any other factor.

If the signal matches up, great - you have proof that the client's tracking of the server time is perfectly accurate over time, and you're done.

If the signal does not match up, there may be a bias from the network, or from within the client OS's software and hardware stack. The solution may be to update your OS, or drivers, or Ethernet hardware. You may just need some boundary clocks to push time farther out into your network. FSMLabs can help diagnose and assist in identifying the best steps based on the collected data - even to the point of pointing out where there may be a temperature issue on the client or the server.

Verifying with a Reference PPS or GPS (on the client)

If a PPS or GPS input can be delivered to the client, the received network sync can be compared directly to a reference signal, right on the client. An example would be a GPS signal delivered to the client via a Spectracom card. The TimeKeeper configuration might look like this:

```
SOURCE0() { PTPCLIENTVERSION=2; PTPDOMAIN=1; }
SOURCE1() { PPSDEV=spectracom; }
```

Here, the client will track PTP on domain 1, but also has a secondary source, a Spectracom card, which has a GPS signal delivered to it. Checking the PTP client signal to GPS is as simple as letting the PTP client sync up, then looking at the TimeKeeper-provided offset recorded for the GPS signal at the same time. Here's an example. In the PTP log file (timekeeper_0.data), there might be a line that looks like:

```
1337318267.945134570 0.000000582 -0.000000042 …
```

Looking at the Spectracom log, there are these two lines that provide bounding points for what the GPS sync looked like at that point:

```
1337318267.000000703 -0.000000293 -0.000001912 …
1337318268.000000708 -0.000000293 -0.000001947 ...
```

Again, TimeKeeper provides plotting tools and management interfaces to visualize the two datasets, but this is an easy way to cross check. From this data, we can tell that the system time was tracking the PTP source with approximately 500 nanosecond accuracy. At that

same point in time, the system time was also in agreement with the Spectracom GPS input to within about 300 nanoseconds. This means that the PTP client is performing well.

If there is a disagreement in the two sources, the bias can be quantified and investigated as a software, hardware, or network issue. TimeKeeper provides the tools needed to narrow in on the sources of any bias.

Checking for Improved System Call Overhead

Once the client sync is validated, it is also worth checking that the system has reasonable overhead for common timing calls, as application timestamping requires fast access to OS timing services. A highly synced clock is one component in the system, but sub-microsecond sync only helps so much if any OS calls may sometimes take several microseconds to complete. If there is a lot of variability in OS call overhead, when the time is returned, there's already some ambiguity in the time data.

TimeKeeper provides optimized timing calls and tools to validate their overhead. As with the ppsgen tool above, the 'timetests' applications are also provided in /opt/timekeeper with the rest of the product. Several tools are provided to validate overhead via static binaries, dynamically linked binaries, and 32 bit applications. These tools can be run both with and without TimeKeeper running to validate any improvements in performance.

Conclusion

This is not meant to be a exhaustive explanation of all best practices, but it is meant to cover the most common issues that FSMLabs' TimeKeeper customers run into in selecting and verifying their timing infrastructure. If you've still got questions, contact us at support@fsmlabs.com.