SCTE CABLE-TEC EXPO 22
SEPTEMBER 19-22 • PHILADELPHIA, PA

2022 Fall Technical Forum
SCTE • CABLELABS • NCTA

Managing the Data Firehose to Put Out Network Fires

A Technical Paper prepared for SCTE by

Jingjie Zhu

Senior Engineer
CableLabs

j.zhu@cablelabs.com

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

List of Tables

Title **Page Number**

1. Introduction

In today's network, data plays a very important role in helping the operators gain more visibility into their networks to make their networks more reliable and more performant. For example, for Cable Modems (CMs), by calculating robust DOCSIS 3.1® profiles using Profile Management Application (PMA), impairments are mitigated, and the OFDM/OFDMA channel's performance is maximized. The profiles calculated by PMA can also be used to target low performing Cable Modems.

For other remote devices such as the Remote PHY Device (RPD) and Remote MAC Device (RMD), YANG modeling language is used in developing their north-bound data models and interfaces, which allows Google Remote Procedure Call (gRPC) Network Management Interface (gNMI) Streaming Telemetry to be easily implemented for advanced real-time device monitoring, largely reducing the probability and duration of service disruption events.

The complexity of data collection itself is also increasing as new devices support more sophisticated measurement functions. For instance, comparing to DOCSIS 3.0 data collection, collecting data from DOCSIS 3.1 CMs requires following much more complex procedures, increasing the amount of resources that the data collector uses and the number of states the data collector needs to track during the data collection process. Therefore, five years ago, CableLabs developed the first Common Collection Framework (CCF) [1],[4] as the initial work of diving into DOCSIS 3.1 data and shared its source code with the industry. The goal was to provide a reference implementation to converge the south-bound data collection interfaces, which are the interfaces for working with devices such as CMs and CMTSs, automate the data collection procedures and the handling of states, and provide standard interfaces to applications on the north-bound of the data collector. Its original goals have been achieved as it has provided operators and vendors a well-documented reference implementation and has served as the go-to data collector in many small-scale trials such as lab trials and limited field trials [2].

However, in the past field trials, CCF's performance wasn't impressive. When collecting data from around 8,000 DOCSIS 3.1 CMs, CCF spent more than 1 hour to complete the tasks even when it's multi-processed and was occupying 100 percent of the CPU resources on the data collection server. While acceptable for trials and development sandboxes, and could be scaled, it was not as scalable as we wanted.

Considering the scale of the whole network where some operators may have many millions of CMs deployed, and the increase in data collection frequency and the number of measurements, a better reference design of CCF is much desired.

In this paper, we use CCF as an example to identify and analyze potential performance bottlenecks and other pain points that could exist in today’s network data collectors. And we share the experience of building the Next-Generation CCF (ng-CCF) to tackle each pain point and overcome scalability challenges. We hope the experience we share could provide references to others who are looking for such a data collection tool, on their way of building their own, or seeking for ways to improve the performance of their existing data collection tools. And we would like to share the new software, ng-CCF, with the cable industry as a reference design.

2. Background

2.1. The Complexity of DOCSIS 3.1 Data Collection

Since DOCSIS 3.1 technology was developed, multiple advanced measurements have been added to the CMs to collect and upload comprehensive physical layer metrics. Such data include:

1. CM downstream orthogonal frequency-division multiplexing (OFDM) symbol capture
2. CM downstream OFDM channel estimation coefficients
3. CM downstream OFDM constellation display
4. CM downstream OFDM receive modulation error ratio (RxMER) per subcarrier
5. CM downstream histogram
6. CM upstream orthogonal frequency-division multiple access (OFDMA) pre-equalization
7. CM upstream OFDMA pre-equalization last update
8. CM downstream OFDM forward error correction (FEC) stats
9. CM downstream spectrum analysis (full-band capture)

These test and query results provide rich information of the physical layer of the access network and are the fundamental requirements of advanced applications such as PMA. They require the data collector to perform multiple sequential simple network management protocol (SNMP) set steps on CMs for each test, and integration with trivial file transfer protocol (TFTP) servers for reading data uploaded by the devices.

On the CMTS side, DOCSIS 3.1 PNM results often require information that are challenging to gather or configure manually, such as the interface index numbers of OFDMA channels. These interface index numbers are often used as unique channel identifiers but are usually a reference number pre-determined by the CMTS and are offered through SNMP only, which suggests that an ideal data collector should automatically collect and prepare such intermediate information prior to data collection tasks that have dependencies on it.

The above aspects make data collection of these advanced PNM measurements significantly more sophisticated than data collection of traditional, common metrics where the collector usually queries the devices with one SNMP get or SNMP walk step for each data type and does not need to manage states.

As of today, this problem is solved as there are existing data collectors that are capable of handling the complexities, such as the first generation of CCF. But it has an impact that it encourages the data collectors to be highly concurrent for simplicity and scalability and to be microservice-like applications for the ease of scaling horizontally.

2.2. CCF Architecture

To allow flexible deployment, CCF version 2 was designed to consist of two microservices at a high-level: the Rest Agent (RA) and the Workflow Controller (WC). For the actual data collection tasks, modular “drivers” are introduced as plugin-like programs in CCF to provide extensibility and support rapid development. The RA provide hypertext transfer protocol (HTTP) APIs for the applications to use, and the WC handles the lifecycles and states of each individual data collection “driver” in parallel; the communication between RA and WC are done through

RESTful API calls. With this architecture, CCF can be scaled horizontally by applying multiplications of RA, WC, and coupled RA and WC instances.

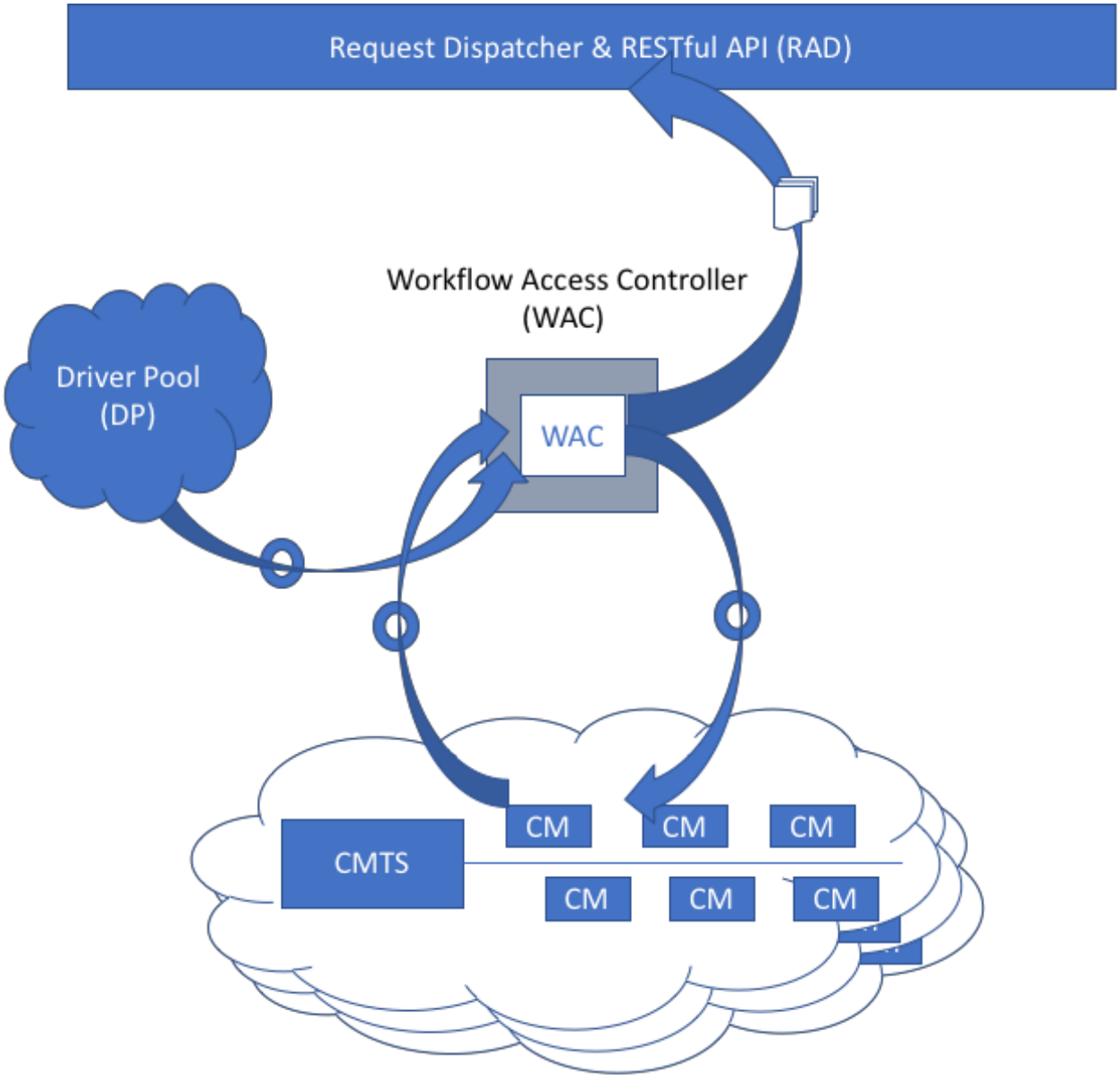

Figure 1 – CCF Architecture

CCF also integrates with external resources such as a local/remote TFTP server for gathering CM and CMTS uploaded PNM files, and an in-memory or filesystem datastore for storing data collection states and results. This architecture has been proven to provide benefits as applications can be easily built upon CCF's abstraction layer and its common APIs, while CCF scales the underlying interfacing activities with the network elements and provides protection to the devices' resources as the duplicated data collection is avoided and the same data is reused by different applications.

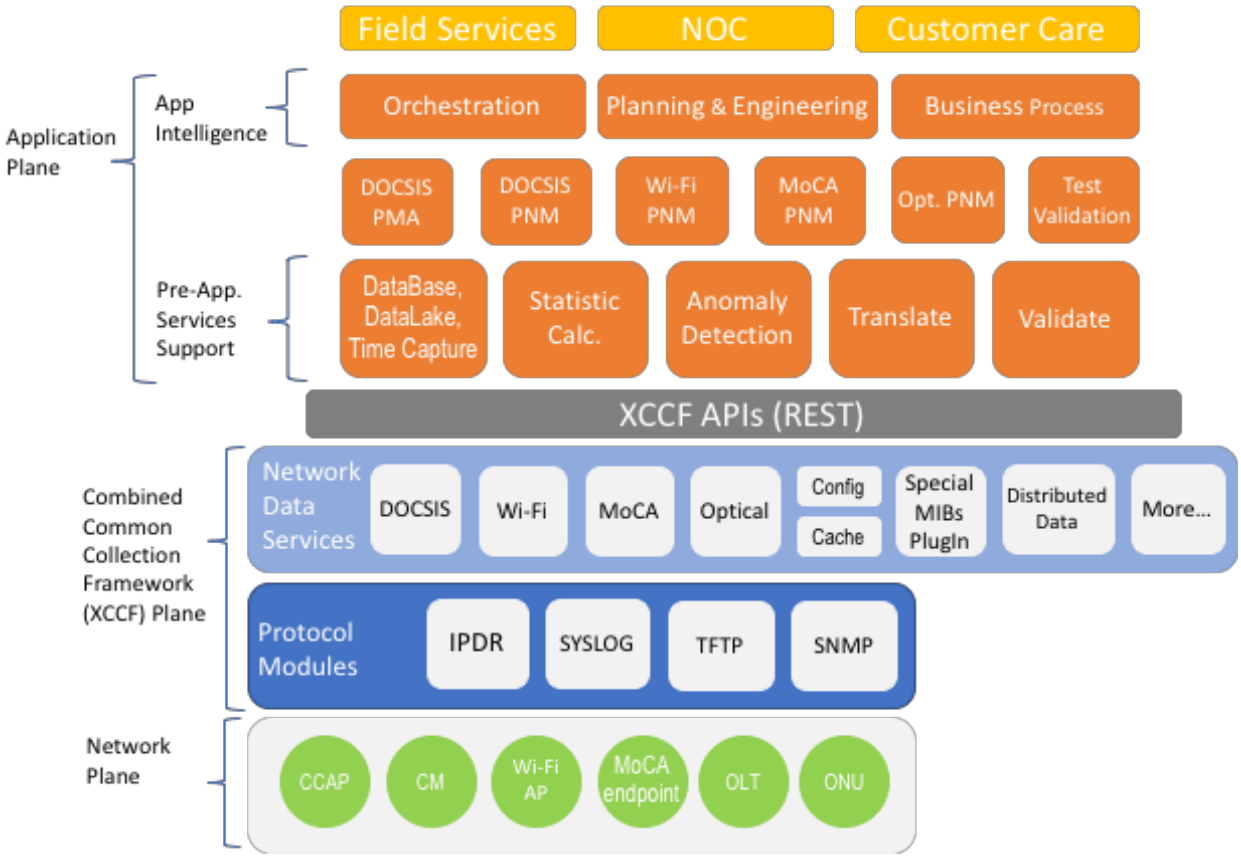

Figure 2 – CCF's role in today's data-driven system

Since the CCF is not intended to be coupled directly with a database, it's often deployed with a data collection scheduler to manage data collection cycles asynchronously, retrieve and decode the data collected by the CCF, and store the decoded data to a data service for applications to use. By doing this, the CCF only stores temporary data from the network and becomes straightforward to manage and maintain.

The drivers of the CCF perform individual data collection tasks that are designated for different devices and different measurements. The driver layer provides a simplified framework for users to easily add or modify data collection processes because the complexity of handling worker states, multiprocessing, and storing the data etc. are handled by the upper layers within the CCF. The drivers are also easy to develop and test as they can be tested individually from the command line.

2.3. CCF Technology Stack

The first two major versions of the CCF were developed in Python3. Python allowed us to quickly develop a working prototype and demonstrate the architecture and functionalities of the CCF. The HTTP APIs of the CCF were implemented with the Flask[9] Python library which provides a simple way to define and develop RESTful APIs. For communications between the

CCF's microservices, namely the RA and WC, the CCF uses Python3's requests [10] library to perform HTTP API calls.

For data storage, the CCF uses an in-memory cache to store temporary data as the default option. As an alternative, the CCF can use the Linux filesystem as a persistent data storage. This approach makes it easy to configure for a quick setup, but also provides persistent data storage options to the users.

For SNMP functionalities, the CCF uses a library python3-netsnmp which provides Python3 bindings for the NET-SNMP C library. Because of this, the SNMP dependencies are not portable and have to be pre-compiled for the system or compiled on the running system.

Last but not least, because DOCSIS 3.1 PNM measurements require the devices to upload the encoded measurement results to a TFTP server, the CCF integrates with an external TFTP server through the filesystem. This requirement allows the CCF to flexibly integrate with any TFTP server by pointing to their upload file directories. However, this method may have potential performance costs as the uploaded files are searched and identified using their filenames on the filesystem.

2.4. Parallelization

Because the CCF was designed to handle many data collection tasks in parallel for efficient data collection, multithreading/multiprocessing/concurrency is required in CCF's implementation. To simplify the driver layer, each driver runs single-threaded tasks, and multithreading/concurrency is handled by the WC.

Because Python has a Global Interpreter Lock (GIL) and it prevents the threads in Python from being "real" threads that use computing resources from multiple CPU cores, multiprocessing is needed for true parallelization that can use multiple cores from modern CPUs. However, in Python, this comes with significant CPU and memory overhead which prevents the CCF from efficiently handling a large number of concurrent data collection tasks using limited machine resources. The Python CCF implementation worked around this by starting a limited number of subprocesses and concurrently handling tasks assigned to each individual subprocess in event loops. This approach significantly helped improve the CCF's resource efficiency when it comes to parallelization. However, based on the trial results, this approach couldn't scale as optimally as desired, so higher efficiency is needed for handling a very large amount of concurrent data collection tasks.

3. Analysis of Pain Points and Challenges

During the past 5 years of using the CCF, helping operators and vendors configure and use the CCF, and building applications around the CCF at CableLabs, we've identified several pain points and challenges that are worth sharing and may be helpful for others to identify similar issues in their data collectors. These pain points and challenges are discussed in the following sections.

3.1. Estimated Data Collection Performance

The first challenge we identified is the CCF's ability to handle data collection for a large number of network devices. To understand the performance challenges at the high level, we can start with estimating the resource requirements of the data collector with the data collection requirements. An example of today's data collection requirements is:

1. the number of devices is around 10 million (DOCSIS 3.1 CMs)
2. a 6 hour data collection interval is required
3. 1 or 2 PNM data metrics are collected

However, in the foreseeable future, the following requirements may be asked given the increasing demand for field data, and the increasing number of deployed CMs, internet of things (IoT) devices, and more.

1. the number of devices is around 50 million
2. the desired data collection interval is 1 hour
3. multiple PNM data metrics are collected

With these future requirements, one can estimate that the data collection system will be required to collect data from around 56,000 devices every second on average. Based on the CCF's performance we observed in the previous field trials where 1 CCF instance spent more than 1 hour to collect Receive Modulation Error Ratio (RxMER) per subcarrier data from around 8,000 CMs, it infers that around 25,000 CCF instances will be required to handle this target workload to complete the data collection tasks within the required interval. This estimated number is overwhelmingly large, and it's asking for the computing power of a data center for a straightforward task of collecting data from network devices in parallel.

Although the CCF is considered a reference design, its performance is not ideal. Even if we consider a potential 10 to 100 times performance improvement for CCF, it will still ask for hundreds if not thousands of servers or virtual machines (VMs) on the cloud to be dedicated to data collection tasks. As the data collection demands continue to ramp up, it could only become more challenging for data collectors such as the CCF to catch up. Not only the resource consumption and cost of such data collection applications is significant, but it also increases the workload and complexity of managing such a large number of severs or VMs, not to mention databases.

3.2. API Performance

Another important aspect for microservice performance analysis is the application programming interface (API) performance. The CCF APIs are the interface for north-bound applications, these RESTful APIs are often called frequently during the data collection sessions as the north-bound applications may continuously check the data collection status of each individual measurement or each batch of measurements. This often results in the APIs being called thousands of times if not more, during the data collection sessions.

The baseline of how many API calls a performant microservice should handle varies by the context and requirements of the application. However, from the API performance testing results, the CCF could only handle up to 300 requests per second, which is significantly less than an ideal number.

The limited performance of the CCF's representational state transfer (REST) APIs can cause the north-bound applications to hang on measurement status checking requests. Low performance APIs could also cause high CPU or input/output (I/O) usage by the application. In CCF, the CPU usage spiked to 100% during the API performance test, which indicates that the CCF's HTTP APIs have a low CPU resource efficiency. When a large number of data collection tasks is being run, and when the north-bound application checks the measurement status frequently, the inefficient APIs can introduce issues to the entire data collection system by competing with other processes in the data collection application on CPU resources.

3.3. Data Store Performance

The CCF comes with an internal data store for saving and managing states, configurations, restricted amounts of collected data, and any other intermediate information. Usually, the CCF is configured to work with the following 2 types of data stores:

- filesystem
- in-memory

Because most web services are I/O bound, it's important to understand the CCF's data stores' performance and identify potential disadvantages of them.

Both data stores keep track of historical data and are implemented with in-memory file indexes. The use of in-memory file indexes makes data operations, such as insertion, deletion, and searching to be efficient. However, accumulating historical data increases the data operation costs over time. When working with hundreds of thousands of data entries, the performance impact is significant. This could further affect the CCF API's performance because slow data operations could shift the CCF APIs from CPU bound to I/O bound, further reducing the number of APIs the CCF can support during data collection sessions. We've observed this causing performance issues on long-running CCF instances in the lab setups and field trial setups.

Because the CCF's historical data storage has not showed value in multiple practical lab and field trials as the CCF has never been used as a primary data storage service itself, it is suggested that the fundamental design of the CCF's data store should be changed to offer improved data store performance. Removing historical data entries could be one of the improvements. In addition, compared to modern databases and caches, such as postgresql and redis, it doesn't provide benefits to implement the CCF's own data store while not taking advantages of the well-adopted data stores, especially when the performance differences are large. Replacing the CCF's data store with well-adopted, open-source databases or caches could further improve the CCF's data I/O and storage performance.

3.4. Configuration

Another pain point we've identified is that the configuration of the CCF requires knowledge that relates to Linux, software engineering, and networking. The prerequisite knowledge has blocked many users from setting up fresh CCF instances quickly without having to come to the developers of the CCF with questions. This situation also renders the CCF's setup documentations difficult to understand for users who are new to this field.

In addition, a typical set of the CCF's configuration files contain around 60 lines of configurations in JavaScript object notation (JSON) format, which further introduces work for the users to build a fresh CCF setup. Each of the decoupled microservices of the CCF (RA and WC) requires a separate configuration file, which adds difficulties and challenges to configuration and debugging for the users.

Considering the purpose and overall complexity of the CCF, the configuration and setup of it has been a major pain point since we shared the CCF with the industry and should be significantly simplified.

3.5. Deployment Challenges

The deployment challenges of the CCF are the collective outcome of the issues and pain points discussed in the previous sections. The performance of the CCF determines the size of the infrastructure that hosts CCF for large scale data collection. With CCF's performance being non-ideal, it's estimated to require a significant amount of computing resources for field data collection from millions of devices, which could add cost, maintenance work, and overall complexities to the data collection system.

In addition, the dependencies of the CCF are not compiled with the CCF's source code nor statically linked, making it complex to manage all of the CCF's dependencies in an internet-less deployment environment. Also, as the CCF is developed in Python and it depends on the NET-SNMP C library, the host of the CCF is required to run a complete Linux operating system which introduces overhead in CPU and memory usage for running the operating system (OS) and OS processes. When containerizing the CCF, this could result in large CCF images and heavy-weight containers.

Finally, the sophisticated CCF configuration process not only makes its instances difficult to set up, but also makes it harder to locate issues for the users to debug.

4. Resouce Usage and Efficiency of Data Collection

As discussed in the previous sections, a high-performance, scalable, and easy to configure and deploy data collector is highly desirable to be the foundation of future network maintenance innovations and cost savings as the data collection demands continue to grow. The initial version of the CCF did not meet the requirements based on the pain points discussed in the previous sections. To build a data collector that could meet these requirements, we start from analyzing

the resource usage of common network data collection tasks, and then identify potential solutions. This analysis is discussed in the following sections.

4.1. Computation

First, understanding what computation tasks the data collector is responsible for during the data collection session is an important step to estimate how efficient the data collector could become.

During a network data collection session, the majority of computations happen remotely on the devices. For example, downstream RxMER per subcarrier data is measured, encoded, and uploaded by each CM. During the data collection session, the data collector is only responsible for facilitating and managing multithreaded tasks, sending requests to the devices, and waiting for the measurements to complete. An individual data collection task should use a negligible amount of CPU resource for most of the time during its lifecycle, which makes it promising that the data collector could become highly CPU efficient and could handle a very large number of data collection tasks at the same time if the concurrency is done efficiently.

Figure 3 shows the roughly measured procedure runtime of performing the DOCSIS 3.1 RxMER per subcarrier measurement and the DOCSIS 3.1 Spectrum Capture measurement.

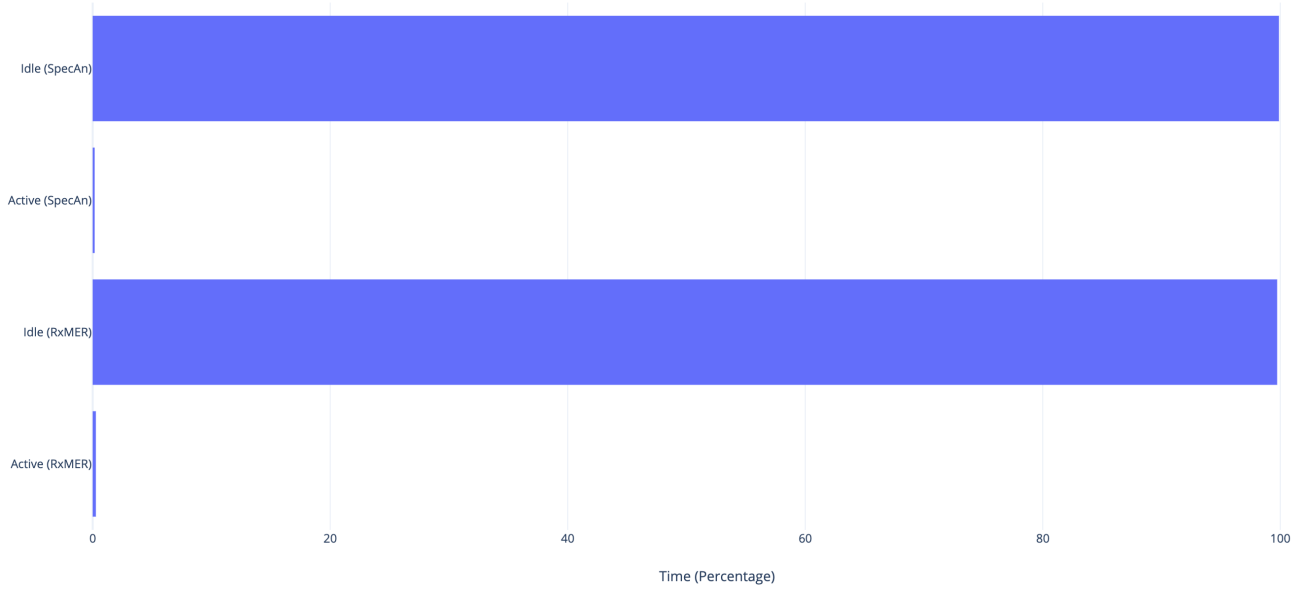

Figure 3 – PNM procedure runtime measurement

We tracked how much time the measurement procedure actively spent for triggering the data collections and how much time it spent on idling and waiting for the collections to complete. In the results shown in Figure 3, the measurement procedure only spent negligible amount of time triggering the measurements compared to its idle time. During its idle time, the computing resources should be made available for other processes.

4.2. Networking

When collecting data from network devices, the data collector uses the server's network connection to query for data or trigger asynchronous measurements on the remote devices. For example, this procedure could involve sending user datagram protocol (UDP) packets (for SNMP) to CMs and CMTSs. To provide a reference data point, the measured number of UDP (SNMP) packets sent to the Cable Modem during a downstream OFDM RxMER measurement session is shown in Table 1.

Table 1 – Downstream OFDM RxMER SNMP network load

Type	Number of Packets	Total Size	Direction
SNMP GET request	9	828 bytes	downstream
SNMP GET response	12	12,369 bytes	upstream
SNMP SET request	2	360 bytes	downstream

The SNMP GET responses transmitted the most amount of data because the data collector periodically checked the measurement status on the device by walking its measurement status MIBs. The entire measurement spent roughly 5 seconds to complete, which can add up quickly.

For RxMER measurement, the data collector performs SNMP SET on 6 MIBs; however, these SNMP SETs can be put into a single SNMP packet to reduce network loads. In the above measurement, the SNMP SETs were completed by sending 2 SNMP packets to the Cable Modem, resulting in 360 bytes of network usage.

If a data collection task uses longer time to complete, the number of SNMP GET request/response packets will increase as the data collector waits for longer durations while continuously sending SNMP GET requests to check the measurement status.

Based on the measured network load of downstream RxMER data collection from one CM, we can estimate the total network load for collecting downstream RxMER data for 1 million CMs at the same time, as shown in Table 2.

Table 2 – Network load estimations for concurrent data collection of OFDM RxMER from 1 million CMs (duration: 5 seconds)

Type	Traffic Rate	Direction
SNMP downstream network usage	1.9 Gbps	downstream
SNMP upstream network usage	19.79 Gbps	upstream

Both downstream and upstream traffic usage are high if the concurrent data collection for 1 million CMs is initiated from a single server.

In addition, packet per second (PPS) is another important statistic to estimate for network load as the servers and routers tend to be PPS bound instead of network throughput bound during large scale data collection. Table 3 shows the estimation of PPS loads.

Table 3 – PPS estimations for concurrent data collection of OFDM RxMER from 1 million CMs (duration: 5 seconds)

Type	PPS
SNMP downstream	2,200,000
SNMP upstream	2,400,000

The PPS numbers are also in the high range especially when we consider that most of the servers or VMs would be challenged to handle millions of packets per second. However, because the network usage is highly dependent on the protocols the data collector uses and the data collection procedures the devices implement, the data collector's network usage is primarily related to the deployment and scheduling scenarios. The data collector software may have limited room for improvement around network usage. Therefore, the above estimations are informative for the large-scale deployment of the data collector.

After the OFDM RxMER data collection from 1 million CMs completes, the CMs upload their measurement results to their designated TFTP servers. Assuming that the PNM files have an average size of 4 KB, 1 million CMs will upload roughly 4 GB of data to the TFTP server once the measurements complete. This would add additional network load to the system and transmit a considerably large number of packets per second through the network upstream as TFTP by default uses a packet size of 512 bytes. This calculation also leads to the analysis of data storage in the next section.

4.3. Data Collector's Storage

Data storage, or temporary data storage is another resource we should consider while building a large-scale data collection system. Assuming that a single instance of the data collector is responsible for data collection from 1 million devices, and each individual measurement data has an average size of 4 KB. This assumption results in 4 GB of storage usage per round and per measurement type during data collection.

For a high-performance data collector, an ideal design is to use dedicated long-term data stores for the collected data, and only cache the latest data collection and internal states within the data collector's temporary storage. This way, the performance of the data collector is less limited by I/O speeds as it would require less storage for data and could take advantage of in-memory caches for high-speed data access.

Ideal implementations of such caches could take advantage of open-source libraries such as BigCache for Golang, or take advantage of well-maintained and widely adopted caches such as Redis. Both are performant options and provide protections to the memory consumption through

configuring memory usage limits and aging off old data entries. And based on the above estimation of temporary data storage requirements, the data collector's cache can be fully implemented in memory while keeping the memory consumption within a reasonable range.

4.4. Data Collector's APIs

The data collector's APIs are responsible for the interactions between the data collector and northbound applications, schedulers, or the users etc. The APIs' performance is primarily determined by three factors:

- CPU processing speed
- Data I/O speed
- Network speed

The APIs' performance could be CPU bound if the API calls instantiate processing loads on the data collector which use significant amount of CPU resources. When the API calls require the data collector to communicate with the data store(s) very often, the performance of APIs could be I/O bound. And finally, when the data exchanges between the API callers and the data collector introduce significant network loads, the APIs' performance could be bound by the throughput of the network interface. The throughput of the network interface could be a limitation that's outside of the scope of the data collector's design considerations. However, minimizing the data payload sizes for API calls would be recommended. In addition, although the data collector could use high-speed in-memory caches as data stores, it's always recommended to reduce the number of direct data store hits from the API calls.

Ideally, for scalability, the data collector's APIs should not introduce high CPU loads and should focus on providing efficient connections to the data collector's data storage.

4.5. Parallelization

Highly optimized implementation of parallelization could drastically reduce the amount of resource the data collector uses for large-scale data collection tasks in deployment. In contrast, an inefficient implementation of parallelization could introduce a significant amount of memory overhead and CPU overhead. For example, the initial version of the CCF implements parallelization in Python using multiprocessing and event loops, which has the following shortcomings.

- High memory overhead introduced by multiprocessing in Python
- High CPU overhead introduced by increased Internal Procedure Calls (IPCs)
- Python as a programming language is not ideal in performance and efficiency

Fortunately, these shortcomings have already been addressed in modern programming languages such as Golang. With the Golang source code being compiled to native code and due to the goroutines and channels, highly efficient parallelization could be achieved for the data collector.

5. The Next-Generation CCF (ng-CCF)

Based on the analysis around resource usage and efficiency in the previous section, we now have clear objectives to develop the Next-Generation CCF (ng-CCF) [3] to resolve the pain points we've identified in the initial version of the CCF. The main objectives of the ng-CCF are to have

- High performance and efficient concurrency,
- High API performance,
- High data I/O performance,
- Efficient resource usage,
- Simplified installation and configurations,
- Simplified dependency management, and
- Implementation of all CCF's data collection functions and APIs for compatibility.

In addition to the main objectives, we also identify features that could be useful additions for ng-CCF:

- Having a built-in TFTP server
- Supporting the integration with remote/external TFTP servers
- Having a built-in data store
- Supporting the integration with external data stores
- Having a built-in SNMP client
- Supporting integration with gNMI targets/clients
- Cross-platform
- Horizontal and vertical scalability
- Small executables

Based on these objectives, we designed and developed the ng-CCF which has significant advantages and improvements compared to the initial versions of the CCF. The design, implementation, and performance analysis are discussed in the following sections.

5.1. Technology Stack

Referring to the analysis around resource usage and efficiency in the previous section, we decided to start the development of the Next-Generation CCF from completely rewriting the software in Golang as many objectives would be impossible to achieve if we build the ng-CCF based on the source code of the CCF which is in Python. This decision has a trade-off that the entire source code of the data collector needs to be rewritten, but it allows us to take advantage of Golang's ability to handle concurrency in a highly efficient way. This choice also allows the source of the ng-CCF to be compiled into a single statically linked executable which provides benefits to installation, configuration, and deployment of the data collector.

To implement the HTTP APIs for the ng-CCF, we chose to use Golang's Fiber [6] library instead of the built-in HTTP package because the Fiber library is built on top of Golang's Fasthttp [11]

package which provides superior API performance as shown in the benchmark results in Figure 4.

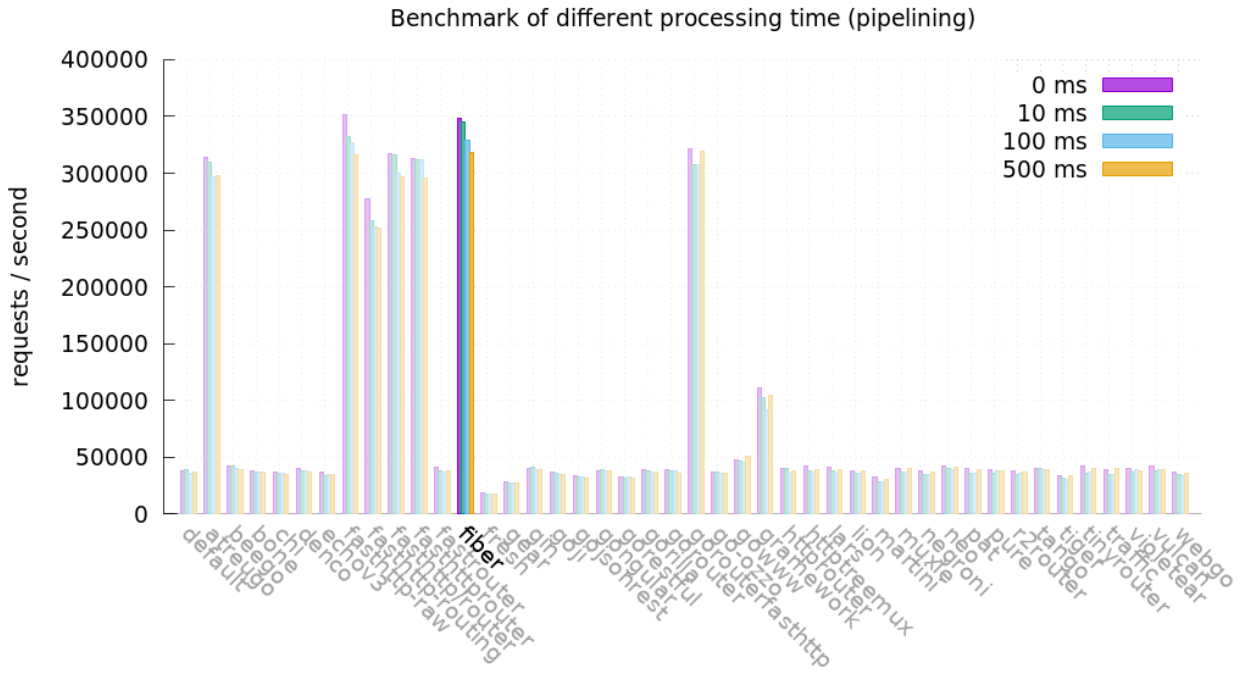

Figure 4 – gofiber's API benchmark (requests per second)
https://docs.gofiber.io/extra/benchmarks

For data storage, we chose to define an interface for the ng-CCF's data stores and implemented the interface with a built-in cache built on top of Golang's BigCache library and a Redis client to communicate with the external Redis cache as 2 supported options. This interface can be conveniently implemented for the ng-CCF to integrate with any other types of data stores such as MongoDB or PostgreSQL. The built-in cache allows the users to start the ng-CCF without relying on external services, whereas the Redis cache allows multiple ng-CCF instances to share the same remote cache and make the cache accessible to be backed up, duplicated, or persisted. The Redis cache option is particularly useful in deployment because it largely reduces each ng-CCF instance's memory usage and allows the users to host and manage the caches on dedicated servers. The built-in cache's speed is bound by the time complexity of the in-memory data structure and the memory speed. And the Redis cache's speed is bound by the speed of Redis, memory speed, Redis API call's speed, and potentially network speed if the cache is remote.

For SNMP, we decided to not rely on OS dependencies such as NET-SNMP and chose an SNMP library gosnmp [7] which is fully implemented in Golang to make the executable portable. This SNMP library supports all SNMP functionalities the data collector needs, such as GET, SET, WALK, BULKWALK, and SNMPv3. Using this well-integrated library also allows the data collector to provide very detailed debugging messages for SNMP down to a per packet payload

level, which could be helpful for debugging the system when the devices don't respond as expected.

We also decided to give the ng-CCF a built-in TFTP server so that it becomes a completely self-contained software solution when there's need for a quick setup and trial. In the initial CCF, the TFTP server is an OS dependency and is integrated with the CCF through the Linux filesystem. In the ng-CCF, the TFTP server is implemented with Golang's tftp [8] library which allows the ng-CCF to process all uploaded files into memory without relying on system calls and using the slower hard drive.

In addition to the built-in TFTP server, the ng-CCF has TFTP, SFTP, and HTTP clients to handle different types of integrations with external TFTP, SFTP, and HTTP servers. For example, the ng-CCF can integrate with remote or external TFTP servers using its TFTP client. This ability could be particularly useful if there are already TFTP servers in deployment. Some CMTSs may implement an SFTP server for the applications to download PNM measurements instead of uploading the measurement results to a TFTP server. The SFTP client in the ng-CCF allows it to automatically switch between the TFTP client and the SFTP client based on the detected CMTS types. For integration with external HTTP services such as Prometheus, the ng-CCF driver can leverage its built-in HTTP client.

Finally, for concurrency, although Golang's goroutines are used in many submodules in the ng-CCF, we decided to employ Golang's "ant" package to build the primary data collection task pool to automatically manage task lifecycles and potentially achieve higher performance compared to using unlimited goroutines.

5.2. Architecture

The architecture of the ng-CCF is largely simplified compared to the initial version of the CCF. In the ng-CCF, there's no longer separate microservices that introduce communication overhead. To reduce the amount of computation, the architecture tries to reduce the amount of CPU processing and data store access when possible. The architecture drawing is shown in Figure 5.

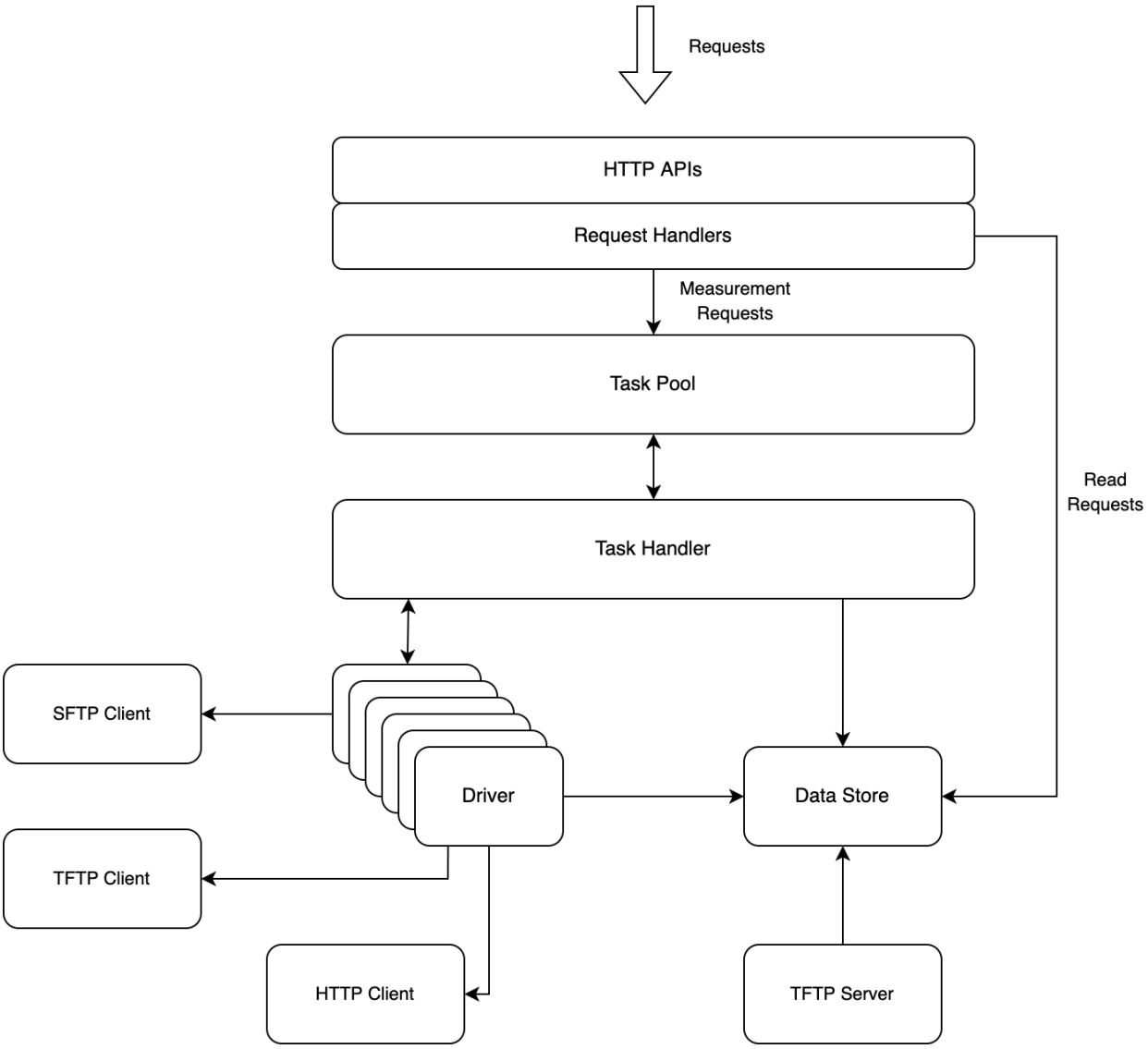

Figure 5 – ng-CCF's architecture

The ng-CCF's data store keeps the latest temporary data and serves as the state storage for data collection tasks and the storage for the TFTP server. The data store operations are thread-safe which makes the data store ideal for the goroutines to pass states and data blobs in addition to using Golang channels for local messaging between goroutines.

When a read request comes to the ng-CCF, the request handlers directly connect the request to the data store for the data request. Because the ng-CCF is designed to use in-memory caches, there's currently no need for an additional caching layer before the data store.

When a data collection and measurement request come to the ng-CCF, the request handlers create the description objects of the tasks according to the request and pass the task description objects to the task pool. The task pool then instantiates an individual, concurrent task handler for this specific task, and proceeds with executing the task and sub-tasks and managing their life

cycles. When the data collection tasks complete, the resulting data are collected from the drivers by the task handler and then stored into the data store along with the updates task states for future read requests.

In this architecture, the task pool handles the majority of the concurrency in the data collector, and the task handler manages sequential and parallel execution of individual sub-tasks and their life cycles, making the software easy to manage and maintain.

Concurrency can also be implemented within the drivers to boost data collection performance. For example, a driver can concurrently send SNMP GET and WALK requests to different MIBs for faster data collection. This is optional for the implementation of the ng-CCF drivers, and it is local to the drivers, which means it's isolated and modular and doesn't increase the overall complexity of ng-CCF's concurrency and internal messaging.

This architecture also makes it easy for testing each individual data collection drivers and testing the API handlers with mock drivers. The modular design makes the ng-CCF easy to maintain and update during long-term development and deployment.

5.3. Data Collection Functions

To make the ng-CCF compatible with existing applications that depend on the CCF and also improve its data collection capabilities, a wide variety of data collection drivers have been developed for the ng-CCF. The drivers provide functions to collect DOCSIS 3.1 specific data elements as well as DOCSIS 3.0 data elements. They also provide data collection functions from external services such as Prometheus, which can be used for integration with gNMI collectors. The current list of ng-CCF drivers is shown in Table 4.

Because the drivers are similar to plugins in the ng-CCF, new data collection functions can be conveniently added to the ng-CCF with concurrency, data storage, and API calls automatically handled.

Table 4 – ng-CCF data collection functions

Type	Description
CM OFDM downstream RxMER	A DOCSIS 3.1 specific data element that provides per subcarrier RxMER data of OFDM channels used by the CM
CM OFDM channel estimation coefficients	A DOCSIS 3.1 specific data element that provides per subcarrier channel estimation data of OFDM channels used by the CM
CM OFDM constellation diagram	A DOCSIS 3.1 specific data element that provides constellation diagram data of OFDM channels used by the CM
CM downstream histogram	A DOCSIS 3.1 specific data element that provides downstream power histogram data measured by the CM

Type	Description
CM OFDMA upstream pre-equalization	A DOCSIS 3.1 specific data element that provides the upstream OFDMA pre-equalization coefficients that the CM is using
CM OFDMA upstream pre-equalization last change	A DOCSIS 3.1 specific data element that provides the last adjustments to the OFDMA pre-equalization coefficients that the CM is using
CM downstream OFDM FEC summary	A DOCSIS 3.1 specific data element that provides a OFDM FEC summary for each individual modulation profile over a 10-minute or 24-hour time frame
CM upstream OFDMA RxMER	A DOCSIS 3.1 specific data element that provides per subcarrier RxMER data of OFDMA channels used by the CM
CM downstream spectrum capture	This measurement provides the full-band capture data of the CM's downstream spectrum
CM SC-QAM upstream pre-equalization	This measurement provides the upstream SC-QAM pre-equalization coefficients that the CM is using
CM capabilities	Collect and decode CM capability requests and responses per TLV 5 defined in the DOCSIS 4.0 MULPI specification
CM events	Collect CM device event logs, event times, and event IDs
CM OFDM channel topology	Discover the OFDM channel based logical topology
CM OFDMA channel topology	Discover the OFDMA channel based logical topology
Prometheus data	Collect any data from Prometheus APIs

5.4. Packaging

The source code of the ng-CCF is written in Golang compiled into a statically linked executable which contains all required dependencies, and no other software packaging process is required. This packaging allows the ng-CCF to run inside of a "scratch" docker container which is an empty container that has minimal storage overhead. The size of the ng-CCF's "scratch" image is only negligibly larger than the executable's size, making it extremely resource efficient in cloud deployment scenarios.

The original size of the ng-CCF executable is 15 MB. However, optionally, it's possible to further reduce its size by using an executable packer such as Ultimate Packet for eXcutables (UPX). We used UPX to compress the ng-CCF's executable, which doesn't affect the requirements on the running system and doesn't add any dependencies, yet it reduced the size of the ng-CCF's executable from 15 MB to 3.7 MB which is only 24.67% of its original size.

Having such a compact executable could have benefits in deployment scenarios. Although it won't optimize memory consumption because the executable is auto-decompressed before running, it significantly reduces ng-CCF's storage footprint and potentially results in more efficient network usage during software updates, especially if we consider future development and extensions being applied to the ng-CCF where the executable size may continue to increase to hundreds of megabytes.

5.5. Configuration

Configuration challenges are a major pain point of the initial version of the CCF. Therefore, simplifying the ng-CCF's configuration is one of the high priority focuses of its development initiative.

The initial version of the CCF typically requires a certain level of knowledge in software engineering, Linux, and network engineering, and it requires 3 separate configuration files that consist of around 60 lines of JSON for the users to work through to get a minimal setup.

In the ng-CCF, because of the addition of a built-in TFTP server and a built-in data store, with default parameters, the user can start the ng-CCF using only one command and specifying only one command line parameter. The integration of ng-CCF and external TFTP servers and data stores is also largely simplified. For example, specifying an external Redis data store only requires three additional command line parameters, and replacing the built-in TFTP server with an external TFTP server only requires two additional fields in the API request payload.

From the trial experiences of the ng-CCF after its release, we've heard significantly less confusion regarding the setup and configuration of the data collector. The users have found it intuitive to start the data collector with one command, without having to work with the source code, installing dependencies and OS dependencies on offline machines, configuring microservices and debugging potential connectivity issues and package compatibility issues. The configuration improvement is an overall significant user experience improvement, and it largely reduces the friction of deployment of the ng-CCF as the data collector.

5.6. Scaling

Because the ng-CCF has a highly efficient concurrency implementation and one instance of the ng-CCF can fully utilize the computing resources on the host machine, it can be flexibly scaled vertically and horizontally depending on the use case needs.

Scaling the ng-CCF vertically requires computing resource upgrades on the host machine. Depending on the data collection requirements such as the number of devices, the number of measurement types, and the frequency of data collection, upgrading the computing resource on the host machine may not be viable if the data collection requirements exceed a limit. However, because of the ng-CCF's capabilities, this limit could be very high on a server with reasonable computing power. Therefore, the ng-CCF may have the potential to support large scale data collection using the computing power of a single server. The details and reference performance numbers are described in the following performance testing section.

Scaling the ng-CCF horizontally could be a more reasonable approach to consider in deployment scenarios. It's possible to balance the data collection loads by assigning dedicated ng-CCF instances to each CMTS's data collection needs. However, since the ng-CCF instances can share remote Redis caches for data storage and state storage, a better approach could be flexibly managing the number of running ng-CCF instances or containers based on the immediate data collection needs, and load balance by routing the ng-CCF's API calls to the ng-CCF instance pool. This approach draws an overall simpler and more flexible deployment system and can more efficiently utilize the computing resources as the granularity of load balancing becomes a single API call. Note that, in this approach, the Redis caches may need to be scaled to satisfy high volumes of access requests. An example drawing of such a deployment architecture is shown in Figure 6.

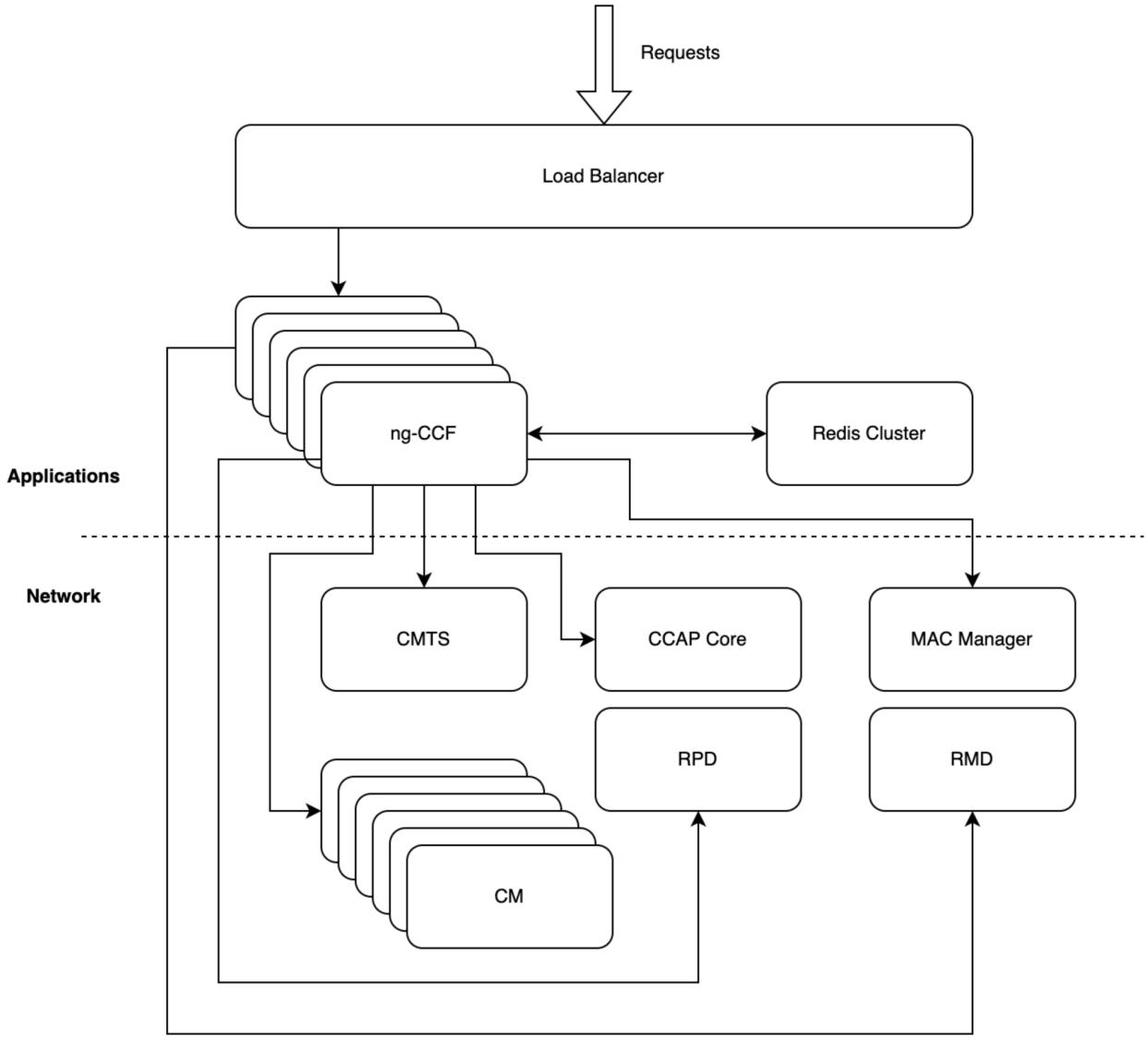

Figure 6 – ng-CCF deployment architecture (horizontal scaling)

5.7. Performance Testing

The performance testing of the ng-CCF focuses on three aspects:

- API calls handled per second,
- API call latency, and
- Mock data collection performance.

The number of API calls handled per second is an indicator of the API performance of the ng-CCF. And the API call latency is the indicator of the API responsiveness of the ng-CCF, which is another aspect of the ng-CCF's API performance. And finally, the mock data collection test uses a mock driver to simulate large scale data collection scenarios and measures the ng-CCF's data collection performance with a reasonable number of concurrent tasks and also pushes the ng-CCF to the limit to see how many concurrent data collection tasks a single ng-CCF instance could handle on a powerful server.

The tests were conducted on an Ubuntu 20.04 VM that's running on a 2021 Macbook Pro with 4 cores of the Apple M1 Pro processor and 8 GB of random-access memory (RAM). We used an open-source HTTP benchmarking tool, called wrk [5] (https://github.com/wg/wrk), running on a separate machine to start 12 threads with 400 concurrent connections to send API requests to both the CCF and the ng-CCF instances as fast as possible for 30 seconds. The measurement results are shown in Figure 7, Figure 8, and Figure 9.

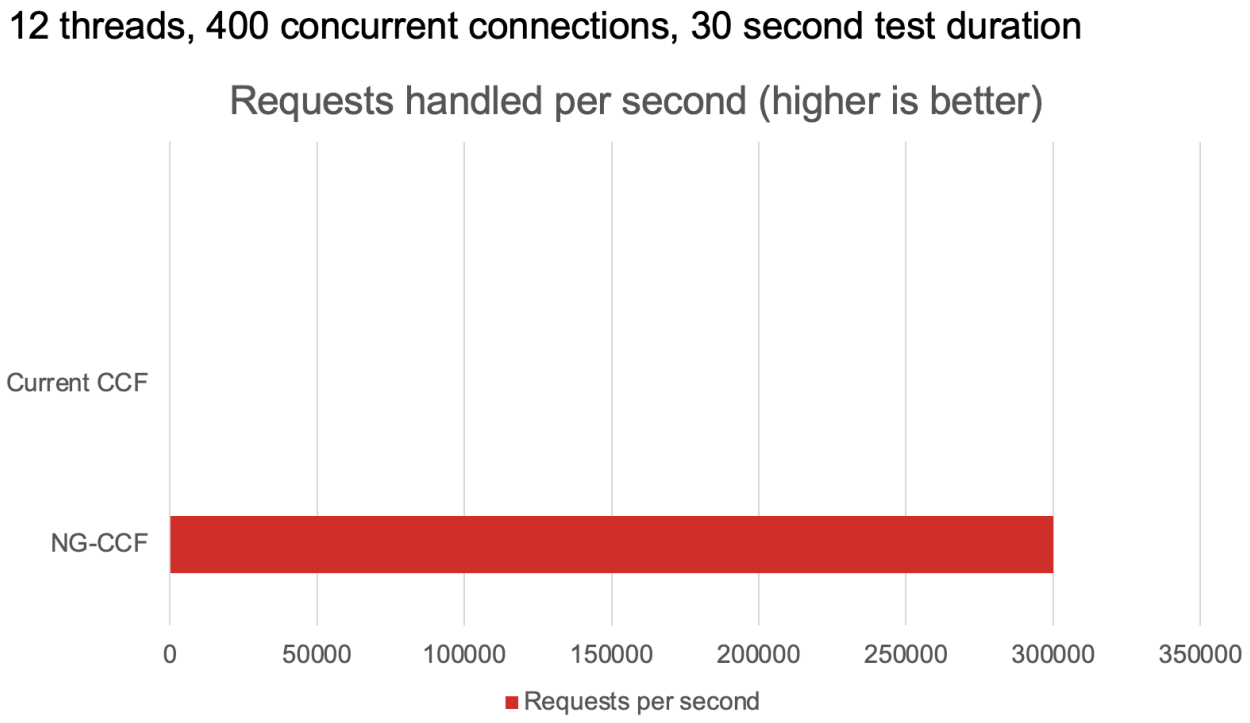

Figure 7 – ng-CCF and CCF's API performance (requests per second)

The initial version of the CCF only could handle 269.78 requests per second whereas the ng-CCF could handle more than 300,000 API requests per second, thanks to the Apple M1 Pro's high bandwidth memory. On an ordinary VM assigned with 4 heavily shared Xeon cores and a memory with lower bandwidth, the ng-CCF still was able to handle more than 110,000 API requests per second.

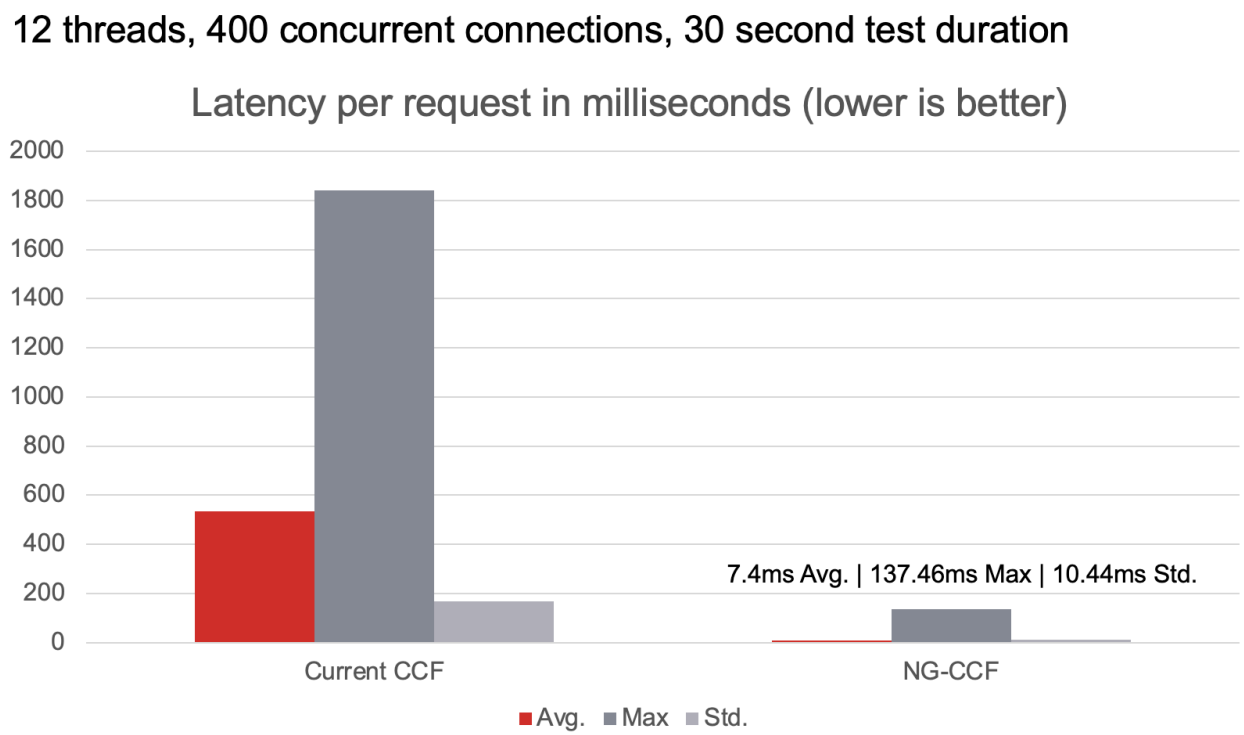

Figure 8 – ng-CCF and CCF's API performance (request latency)

On the API latency measurement for the CCF and the ng-CCF, the difference is significant. The initial version of the CCF averaged more than 500 ms on API responses due to its usage of the Linux filesystem as its data store, and its API response latency peaked at almost 2 seconds. Whereas the ng-CCF averaged 7.4 ms of API response time and peaked at 137.46 ms which is possibly affected by garbage collection in Golang.

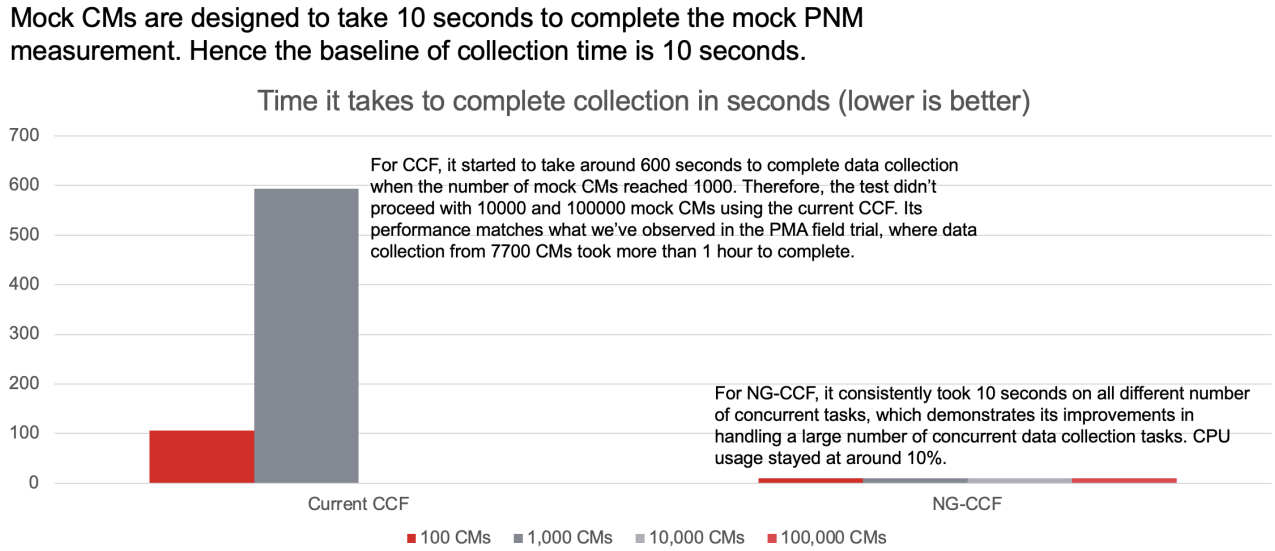

Figure 9 – ng-CCF and CCF's data collection performance

During the simulated data collection testing, we design each measurement to take 10 seconds to complete, most of the time used by the driver waiting for the device to complete the measurement. The initial version of the CCF matched its performance we observed during our first PMA field trial where it took more than an hour to complete measurement from about 7700 CMs. Meanwhile, the ng-CCF effortlessly handled 100,000 concurrent measurements while using only 10% of the 4 core CPU resource, and all tasks completed after the same 10 second wait time.

To find the limit of the ng-CCF for large scale data collection and estimate its requirements for vertical scaling, we conducted another test where one instance of the ng-CCF was hosted on a powerful workstation that has 256 GB of RAM and a 16-core Intel® Xeon® processor (Xeon® Gold 5218 CPU @ 2.30 GHz).

We let the ng-CCF run 0.5 million, 1 million, 2 million, 4 million, 8 million, and 10 million concurrent tasks during the test. Considering heap allocation bottlenecks, each task was designed to run 180 seconds to allow the ng-CCF to complete the allocation of large numbers of tasks.

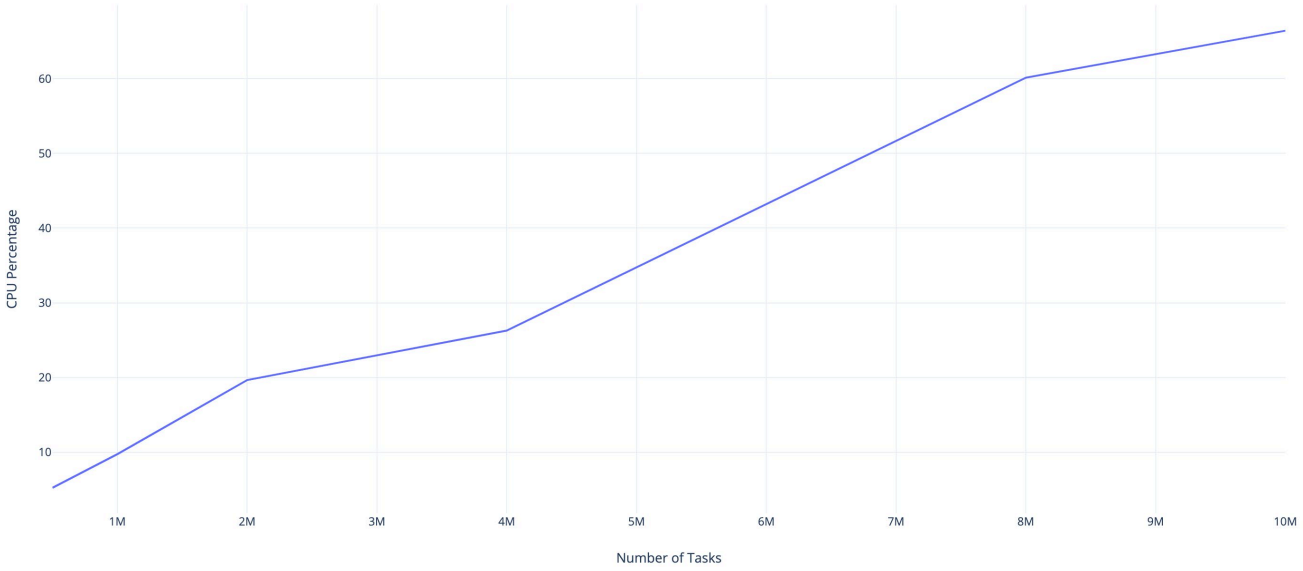

Figure 10 – ng-CCF's CPU usage (16-core) with different numbers of tasks

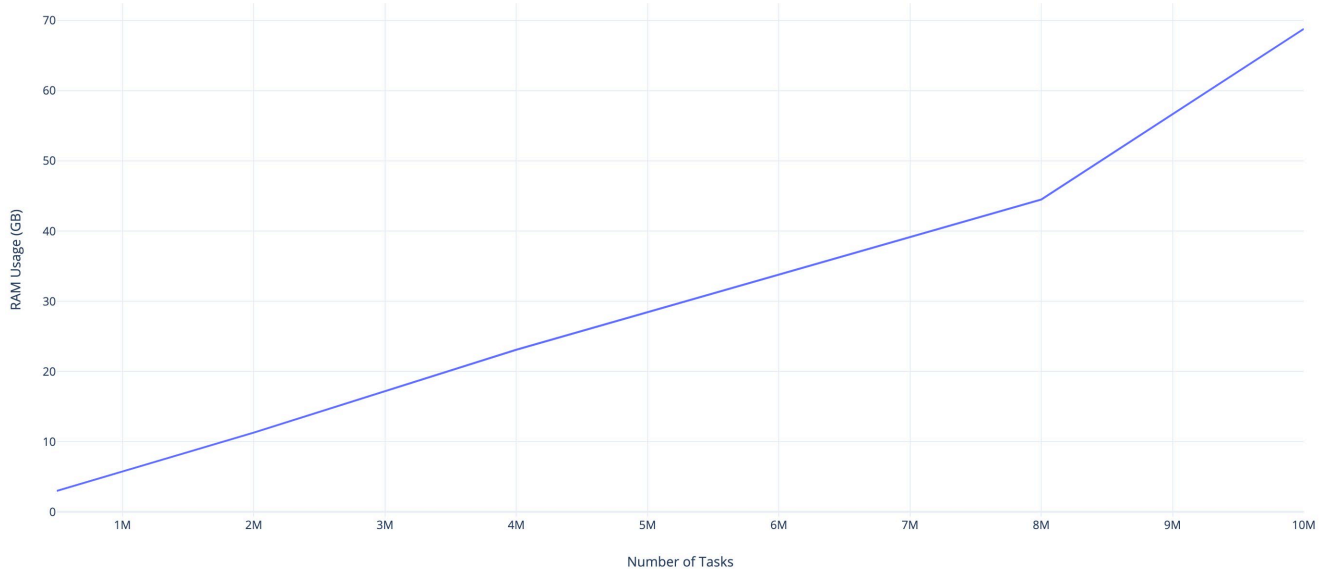

Figure 11 – ng-CCF's memory consumption with different numbers of tasks

When the number of tasks surpassed 2 million, we started to observe slowdowns in the ng-CCF's ability to instantiate new tasks quickly. This is likely bound by heap allocation speed as the CPU usage was still low.

When the number of tasks reached 10 million, although the ng-CCF still didn't fully utilize the processing power of 16 CPU cores, the heap allocation delay became significant enough that 10

million was the maximum number of tasks the ng-CCF could allocate during a 180 second window.

When handling 1 million concurrent tasks, the ng-CCF used 9.75% of the 16-core CPU, and used 5.8 GB of RAM, making it promising to handle large scale data collection using only a few servers. When starting 10 million concurrent tasks on the ng-CCF, it used 66.41% of the 16-core CPU and used 68.8 GB of RAM at peak. The heap memory allocation speed could be a limiting factor for vertically scaling the ng-CCF any further. Therefore, it would be recommended to limit the number of concurrent tasks on each ng-CCF instance to less than 10 million.

6. Conclusion

In this paper, we shared the experience of how we identified the pain points and challenges in an existing data collector and analyzed the resource usage for data collection and potential approaches to improve the efficiency of a data collector. And we shared the details of how we designed and developed the Next-Generation Common Collection Framework (ng-CCF) to overcome the issues we've identified and demonstrated improvements in many different aspects such as improved performance and scalability, small and easily deployable executable, enhanced data collection functionalities, and significantly simplified configuration and setup process.

We hope to share our experience with the industry to help others target potential improvements that could be done in their data collectors. And we hope to share the ng-CCF's source code with the industry to help others take advantage of what we've developed. As the data collection continues to grow, a scalable, performant, and reliable data collector is highly desirable to be the foundation of future network maintenance innovations and cost savings.

Abbreviations

API	application programming interface
Bps	bits per second
CCAP	converged cable access platform
CCF	common collection framework
CM	cable modem
CMTS	cable modem termination system
FEC	forward error correction
Gbps	gigabits per second
gNMI	gRPC network management interface
gRPC	Google remote procedure call
HTTP	hypertext transfer protocol
Hz	hertz
JSON	JavaScript object notation
MER	modulation error ratio
OFDM	orthogonal frequency-division multiplexing
OFDMA	orthogonal frequency-division multiple access
OS	operating system
PMA	profile management application

PNM	proactive network maintenance
RAM	random-access memory
REST	representational state transfer
RxMER	receive modulation error ratio
SCTE	Society of Cable Telecommunications Engineers
SFTP	SSH/secure file transfer protocol
SNMP	simple network management protocol
TFTP	trivial file transfer protocol
UDP	user datagram protocol
VM	virtual machine

Bibliography & References

[1] CableLabs Proactive Network Maintenance Combined Common Collection Framework Architecture Technical Report, CL-TR-XCCF-PNM-V01-180814, August 14, 2018, Cable Television Laboratories, Inc.

[2] Karthik Sundaresan, Jay Zhu, Mayank Mishra, and James Lin, "Practical Lessons from D3.1 Deployments and a Profile Management Application", SCTE 2019

[3] The Next-Generation Common Collection Framework (https://code.cablelabs.com/CCF/ng-ccf)

[4] The Common Collection Framework (https://code.cablelabs.com/CCF/dccf)

[5] Wrk: a modern HTTP benchmarking tool (https://github.com/wg/wrk)

[6] Fiber: a web framework for Go (https://github.com/gofiber/fiber)

[7] gosnmp: an SNMP library written in Go (https://github.com/gosnmp/gosnmp)

[8] tftp: TFTP server and client library for Go (https://github.com/pin/tftp)

[9] flask: the Python micro framework for building web applications (https://github.com/pallets/flask)

[10] requests: a simple, yet elegant, HTTP library (https://github.com/psf/requests)

[11] fasthttp: fast HTTP package for Go (https://github.com/valyala/fasthttp)

