Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

Large-scale Certificate Management on Multi-tenant Web Servers

Ryosuke Matsumoto[1,a)] Kenji Rikitake[2,3] Kentaro Kuribayashi[2]

Received: November 8, 2018, Accepted: June 11, 2019

Abstract: For large-scale certificate management of multi-tenant web servers, preloading numerous certificates for managing numerous hosts under the single server process results in increasing the required memory usage because of the respective page table entry manipulation, which might be a poor resource efficiency and a reduced capacity. To resolve this issue, we propose a method for dynamic loading of certificates bound to the hostnames found during the SSL/TLS handshake sequences without preloading, provided that the Server Name Indication (SNI) extension is available. We implemented the function of choosing the respective certificates with the ngx_mruby module, which extends web server functions using mruby with a small memory footprint while maintaining the execution speed. The proposed method was evaluated by a web hosting service employing the authors.

Keywords: web Server, TLS, operation technology, multi-tenant, large-scale, nginx, mruby

1. Introduction

Because RFC adoption of HTTP/2 protocol is required on HTTPS [1] and because ongoing SSL/TLS support by Google [6] is gaining popularity, supporting HTTPS has become an urgent task of web hosting companies. Although the cost of operators and service users is high for HTTPS, the price of server certificates and the cost of infrastructure for HTTPS execution are also high [12]. However, free Domain-validated (DV) certificates such as Let's Encrypt [7] are beginning to be provided; supporting HTTPS becomes inexpensive.

In a web hosting service based on the multi-tenant architecture [11], a single server process group must manage numerous hosts [21] to provide service at a low price by reducing the hardware cost and operation cost by accommodating hosts with high integration. The term single-server process group means that a server process is shared by numerous hosts, not with a process activated for each host on the highly integrated multi-tenant architecture. The number of server processes is independent of the number of hosts; rather it depends on the web server implementation, which might initiate hundreds of server processes at the startup.

To communicate with HTTPS, the existing web server software must load the secret key paired with the server certificate for each host at the server process startup [26]. However, the highly integrated multi-tenant architecture does not take advantage of reduced hardware costs and operation costs with the existing mechanism because, if hosts are accommodated in a high degree of integration, starting the server process by reading numerous server certificates and secret keys takes much time. Also, the memory usage of the server process increases depending on the number of hosts.

As described in this paper, we propose a large-scale certificate management method that reduces the memory consumption of the web server process effectively and efficiently by dynamically acquiring a corresponding server certificate and secret key data for each request in a highly integrated multi-tenant web server.

The proposed method does not preload a server certificate and a secret key at the web server process startup, but rather dynamically loads the server certificate and the secret key from a database for each request based on the requested hostname or IP address/port during the SSL/TLS handshake.

We implemented the new feature of ngx_mruby [20], which can accommodate the loading phase of certificates. ngx_mruby [20] is a fast and memory-efficient web server extension mechanism scripting with mruby [13] for nginx [14]. Server certificates and secret keys are stored in Redis [23], which is a kind of KVS [8]. The certificate and the secret key corresponding to a hostname are acquired using the mruby code.

The implementation of the proposed method is practical because ngx_mruby can readily control nginx internals without changing the nginx source code, which is widely used as a web server to terminate HTTPS. The implementation of ngx_mruby ver. 1.16.0 was published as OSS [*1] in February 2016.

The remainder of the paper is organized as described below. First, we summarize the tasks for constantly using HTTPS in a highly integrated multi-tenant web server as described in Section 2. We describe the architecture and implementation of the

[1] SAKURA Research Center, SAKURA Internet Inc., Fukuoka 810–0042, Japan
[2] Pepabo Research and Development Institute, GMO Pepabo, Inc., Shibuya, Tokyo 150–0031, Japan
[3] Kenji Rikitake Professional Engineer's Office, Toyonaka, Osaka 560–0043, Japan
[a)] r-matsumoto@sakura.ad.jp

[*1] https://github.com/matsumotory/ngx_mruby/releases/tag/v1.16.0

proposed method in Section 3. In Section 4, we quantitatively validate the problems of the existing method and evaluate the efficiency of the proposed method. In Section 5, we evaluate the proposed method in a production of hosting service of our employer for one month. Section 6 concludes the paper.

2. Related Works

A web hosting service [17] is a typical application service of a highly integrated multi-tenant architecture. The web hosting service shares server resources among multiple hosts and provides an HTTP server function for each hostname. In the web hosting service, the function that is identified using a Fully Qualified Domain Name (FQDN) and serves the corresponding content is called a host. As described in this paper, we designate a multi-tenant architecture that can accommodate tens of thousands of hosts as a highly integrated multi-tenant architecture [18].

The highly integrated multi-tenant architecture adopts the virtual host method [27], which processes multiple hosts by a single server process group. Popular web server software such as Apache httpd [24] and nginx can handle multiple hosts by a single server process group using the virtual host method such as the VirtualHost configuration of Apache httpd.

In the existing server certificate management of web servers, the web server loads the certificate associated with each hostname into memory at a web server process startup [26]. The web server reads out the certificate corresponding to an IP address/port or a hostname from memory at each SSL/TLS handshake and starts the HTTPS session. This method can perform processes at a high speed during the SSL/TLS handshake because the certificate is loaded to the memory in advance.

A highly integrated multi-tenant architecture must make the configuration and adopt the process model independently from the number of hosts with the virtual host method because the architecture manages numerous hosts. In operation in a production environment, a single server process might accommodate more than several tens of thousands of hosts.

The existing method [9], [22] must load numerous certificates and secret keys into memory at the server process startup. In the system configuration of a reverse proxy for the TLS termination, the system must first perform TLS communication on the reverse proxy to the hostnames of all hosts accommodated in numerous hosting servers. The reverse proxy must manage configurations and certificates on hundreds of thousands of hostnames. In that case, as the number of server certificates increases, the loading time necessary for configurations and certificates data at the server process startup greatly increases. In addition, the memory usage of the server process increases greatly. These increasing resources might lead to important difficulties.

The existing method must describe all configurations of the certificate associated with each host. The number of lines of the web server configuration files also increases greatly. Those configuration files reduce the readability, which makes server management difficult.

There is an existing method of dynamically describing the setting of a large number of virtual hosts [2], but it is not possible to reduce the number of reading certificates. Also, there is an existing method to reduce the number of certificates with a wild card certificate [3], but it can not be applied to a large number of individual domains.

For example, if using nginx as the web server and setting up hundreds of thousands of virtual hosts and loading certificates and secret keys associated with each host, then the number of nginx configuration lines will be about 2 million lines. It takes about 40 s to start the server process, such as reloading the server process for a new configuration. We describe the details of this startup time in Section 4.1.

The web server using these existing methods is inefficient because it reads all certificates of web sites that are not yet accessed at startup.

3. Proposed Method

3.1 Large-scale Certificate Management

Our proposed method meets the following three requirements, which are fundamentally important for the highly integrated multi-tenant architecture: maximization of balance between computer resource, performance efficiency, and efficiency of the system operation cost while solving the problem described in Section 2:

(1) To support the Server Name Indication (SNI) extension to accommodate hosts
(2) To avoid loading all web server certificates for faster startup of the server processes
(3) To ensure that the memory usage of the web server process is independent of the number of hosts, by dynamically loading the associated server certificates during each SSL/TLS handshake

SNI [4] in requirement (1) is an extended specification of SSL/TLS. Serving multiple HTTPS servers by a small number of IP addresses is crucially important to provide service at a low price under the cost constraint of the highly integrated multi-tenant architecture. SNI allows the selective use of the server certificate in the hostname because SNI tells the unencrypted hostname to the server during the SSL/TLS handshake. SNI is commonly used to accommodate numerous hosts virtually with a single server process group and a single IP address in the highly integrated multi-tenant architecture.

We propose a method by which the server certificate and secret key of the request are dynamically loaded from data-store such as a database, a file system, or API, based on the requested hostname during the SSL/TLS handshake when an HTTPS request comes to the web server, with SNI in requirement (1). The dynamic certificate loading meets the requirement (3).

Using the proposed method, the startup time of the web server process does not depend on the number of hosts and memory usage, and does not increase, which meets the requirement (2). The dynamic certificate loading need not load numerous server certificates beforehand at the startup. Even if the number of certificates increases, the proposed method does not cause the difficulty of taking a long time to reload the server process when a configuration change occurs because the startup speed of the server process is not slowed by the dynamic certificate loading of each request. In addition, adding more hosts by changing the configuration re-

```
server {
  listen              443 ssl;
  server_name         _;
  ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers         HIGH:!aNULL:!MD5;
  ssl_certificate     /path/to/dummy.crt;
  ssl_certificate_key /path/to/dummy.key;

  mruby_ssl_handshake_handler_code '
    ssl = Nginx::SSL.new
    host = ssl.servername
    ssl.certificate = "/path/to/#{host}.crt"
    ssl.certificate_key = "/path/to/#{host}.key"
  ';
}
```

Fig. 1 File-based configuration example of dynamic server certificate management.

```
server {
  listen              443 ssl;
  server_name         _;
  ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers         HIGH:!aNULL:!MD5;
  ssl_certificate     /path/to/dummy.crt;
  ssl_certificate_key /path/to/dummy.key;

  mruby_ssl_handshake_handler_code '
    ssl = Nginx::SSL.new
    host = ssl.servername
    redis = Redis.new "127.0.0.1", 6379
    ssl.certificate_data = redis["#{host}.crt"]
    ssl.certificate_key_data = redis["#{host}.key"]
  ';
}
```

Fig. 2 KVS-based configuration example of dynamic server certificate management.

quires no server process reloading because the proposed method can dynamically analyze the certificate location from the hostname.

By aggregating data in databases and caches that can communicate via TCP, Our proposed method ensures the availability and performance of the service system by increasing the number of web servers using a scale-out model as the number of HTTPS requests increases. Our system can readily prepare an HTTPS environment by linking the databases that store the user data such as the hostname, certificates, and secret key data, under TLS options contracts with customers in a production environment.

3.2 Implementation

In our proposed method implementation, we used ngx_mruby, which can extend nginx scripting with mruby and process at high speed with less memory usage. In addition, the OpenSSL [15] ver. 1.0.2 or later has a function that calls back an extension function of the SSL/TLS handshake behavior such as custom loading server certificates and secret keys during the SSL/TLS handshake, `SSL_CTX_set_ cert_cb()` [16]. By making this function executable from ngx_mruby, the callback function using mruby during the SSL/TLS handshake on nginx [19] can be written using ngx_mruby, which enables the server administrator to implement the dynamic certificate loading algorithm easily for various systems.

Figure 1 presents an example of dynamically loading the server certificate by determining the file path from the requested hostname of each request. By passing the file path to the `certificate` method and the `certificate_key` method of the instance of `Nginx::SSL`, the server certificate and the secret key are dynamically loaded during the SSL/TLS handshake.

Figure 2 presents an implementation example of dynamically loading server certificates and secret keys at the SSL/TLS handshake using ngx_mruby. The server certificates are stored in Key-Value Store (KVS) such as Redis for the requested hostname of each request. The `certificate_data` and the `certificate_key_data` methods pass data of a certificate and a private key themselves: not a file.

Figure 3 depicts a design example in operation of production. The administrator saves the server certificate and secret key data in the database. When nginx receives an HTTPS request, it loads the server certificate and secret key from the database via ngx_mruby scripts and establishes the SSL/TLS session. The

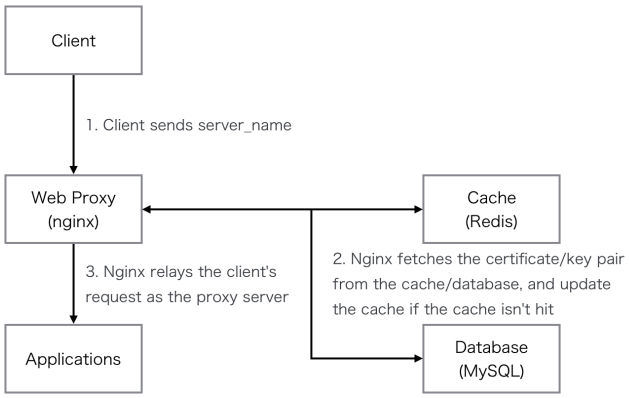


Fig. 3 System example of dynamic server certificate management.

proposed method implementation temporarily stores cache data using KVS such as Redis, which enables high-speed access to the data to reduce the connection cost to the database for each request.

In addition, applying a configuration of a new certificate of other host requires no reloading server process. The proposed method can dynamically analyze where the certificate is located from the hostname, like a file path including a hostname or database with a hostname as key. When adding a new certificate, if the server administrator registers certificate data in the database, then the proposed method can apply HTTPS to the existing site without reloading the server process.

Even if the number of servers is increased for availability or performance, our proposed method implementation can readily share server certificate data via a TCP connection with a database or cache server. The in-memory cache can also be used for mitigating performance degradation if the network latency from the web server to the cache server becomes a performance problem. The implementation of ngx_mruby ver. 1.16.0 has already been published as OSS as of February 2016 [*2].

4. Evaluation and Consideration in the Production Environment

To confirm the efficiency of the proposed method, we clarify the problem of the time of startup (preloading) of the existing method from the experiment in the existing problem described in Section 2. Next, we compare the existing method (preload-

[*2] https://github.com/matsumotory/ngx_mruby/releases/tag/v1.16.0

Table 1 Experimental environment.

	Specification
CPU	Intel Xeon E5-2620 v3 2.40 GHz 24 thread
Memory	32 GBytes
Server	NEC Express5800/R120f-2E

Table 2 Result of startup time achieved using the existing method.

item	value
real time	42.662 s
system CPU usage time	37.280 s
user CPU usage time	5.387 s
virtual memory size (VSZ)	3,207,592 Kbytes
physical memory size (RSS)	3,175,912 Kbytes

ing) with the proposed method (dynamic loading), which saves the server certificate and secret key data in Redis and which acquires data from the file and Redis for each SSL/TLS handshake. **Table 1** shows the experiment environment.

4.1 Measurement of Memory Usage and Startup Time of Existing Methods

In the environment presented in Table 1, we validate the problem described in Section 2 using nginx ver. 1.11.13. We generated server certificates and secret keys of 4,096 bit key length for 100,000 hosts by the `openssl` command for each host configuration of nginx and measured the memory usage and startup time of nginx server processes.

The master process of nginx initially loads all the server certificate data and copies the worker process for request processing using a fork() system call after the initial processing of the master process is completed. In this experimental environment, the number of worker processes is 24, which is the number of logical cores of the CPU using the configuration setting parameter `worker_processes auto`.

We measured CPU metrics using the Linux time command until all worker processes completed the initialization. We also determined memory usage size of a certain worker process, selected arbitrarily from whole worker processes using the Linux `ps` command: we adopted both fields on virtual memory (VSZ) and physical memory (RSS). We later discuss a whole memory usage size comparison of the existing method and the proposed method in Section 4.2.

Table 2 presents the result. Loading of server certificates depends on the performance per core because a single master process uses only one CPU at first. For an era in which CPU usage efficiency increased by the number of cores, decreasing this processing time is difficult. No notable point exists for the usage time of the user CPU and system CPU.

The memory usage size of a worker process acquired from both virtual memory (VSZ) and physical memory (RSS) fields from the `ps` command is about 3 GBytes. Physical machines in recent days often have large physical memory capacity: the size is usually over tens or thousands of gigabytes. Therefore, it is no problem if a process occupies even 3 GBytes or so of memory space.

4.2 Performance Evaluation of the Proposed Method

We evaluated the performance of the proposed method. We used nginx, which is linked ngx_mruby module as web server software for evaluation.

```
# dynamic certificate
server {
  listen               58085 ssl;
  server_name          _;
  ssl_protocols        TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers          HIGH:!aNULL:!MD5;
  ssl_certificate      /path/to/dummy.crt;
  ssl_certificate_key /path/to//dummy.key;

  ssl_prefer_server_ciphers on;
  ssl_session_cache off;

  mruby_ssl_handshake_handler_code '
    ssl = Nginx::SSL.new
    redis = Userdata.new.redis
    domain = ssl.servername
    ssl.certificate_data = redis["#{domain}.crt"]
    ssl.certificate_key_data = redis["#{domain}.key"]
  ';

  location / {
    root /path/to/html/;
  }
}

# preload certificate
server {
  listen               58086 ssl;
  server_name          _;
  ssl_protocols        TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers          HIGH:!aNULL:!MD5;
  ssl_certificate      /path/to/dummy.crt;
  ssl_certificate_key /path/to/dummy.key;

  ssl_prefer_server_ciphers on;
  ssl_session_cache off;

  location / {
    root /path/to/html/;
  }
}
```

Fig. 4 Configuration of dynamic loading and preloading.

We set the nginx configuration of both the existing method and the proposed method to different ports. **Figure 4** shows ngx_mruby configuration. The configuration designed to listen on port 58085 of the proposed method is a configuration to load the server certificate and secret key with the requested hostname as the key during the SSL/TLS handshake. The configuration to listen on port 58086 of the existing method is a configuration of preloading server certificates at the startup time of the web server process. Both configurations fixed the cipher suites on the server side to prevent the SSL/TLS session cache to maximize the impact during the SSL/TLS handshake.

In the case of configuration to load multiple certificates, the computational complexity of loading certificates required for the existing method is O(1) because the method treats the correspondence between a hostname and a certificate as a hash algorithm in nginx. On the other hand, in the proposed method of reading the certificate of each HTTPS request, the computational complexity of loading certificates required for the proposed method for each HTTPS request is O(1) because the method acquires the certificate from the KVS using the requested hostname as the key. From the above, we found that it is necessary and sufficient for the evaluation in this experiment to set one certificate to be read in the configuration of the existing method and the proposed method.

In general, the ab [25] command is widely used in HTTP

Table 3 Experimental results obtained using the proposed method.

Nsc	proposed method dynamic loading (req/s)	existing method preloading (req/s)
10	171,456.60	171,914.98
100	172,383.84	172,758.28
500	172,714.81	173,631.06
1,000	171,872.24	173,272.53

benchmark experiment in a single thread. In a single thread, when benchmarking HTTPS, the benchmark command runs out of one CPU core before using hardware by server software because the CPU usage of the SSL/TLS handshake is large for the client.

To avoid this difficulty, we used HTTPS benchmark software wrk [28], which supports multi-threading to compare the performance. While changing the number of simultaneous connections, we sent 5 million requests in all and measured the number of requests per second.

We adopted TLSv1.2 of nginx configuration that enables TLS ver. 1.2. We also adopted ECDHE-RSA-AES128-GCM-SHA256 [10] among the cipher suites recommended by Mozilla. The requested content uses index.html of 612 bytes enclosed with nginx by default.

Table 3 presents the results. Nsc in the table is an acronym representing the number of simultaneous connections. In the experimentally obtained results, we observed that almost no performance difference exists between preloading and the dynamic loading method.

We considered that the process necessary for dynamically loading a certificate is almost negligible because encryption and compound processing in the SSL/TLS handshake are extensive. When the number of simultaneous connections is in the thousands, both the existing method and the proposed method are somewhat degraded in performance, but this result is within the error range because the difference is also less than 1%.

In the preloading method, a difficulty exists in that the amount of memory usage increases as the number of hosts increases in the highly integrated multi-tenant architecture. However, the memory usage of the proposed method at startup is small because the dynamic loading method requires no initialization process to store all SSL/TLS configuration data, such as server certificates, in memory at startup. Even when compared with nginx not using TLS, there was no difference in memory usage and startup time.

In HTTPS communication, which requires more CPU processing than HTTP, the proposed method makes it easy to scale out the server merely by increasing the number of servers because certificate management is performed via a cache server such as Redis via TCP. That feature has great operational benefits.

4.3 Evaluation in Production

We applied the proposed method to a hosting service production of our employer and evaluated it on the operation in the production.

The hosting service adopted the existing method (preloading), which loads the certificate at the time of startup using Apache httpd before applying the proposed method.

As the evaluation method, we measured the transition of the total number of certificates the number of requests processed per second, the CPU usage rate, and memory usage for one month during March 4 through April 4 of 2017 when the existing preloading method was adopted. We compared the transition with measured values of the same kind during one month from July 22 through August 22 of the same year after applying the proposed method (dynamic loading).

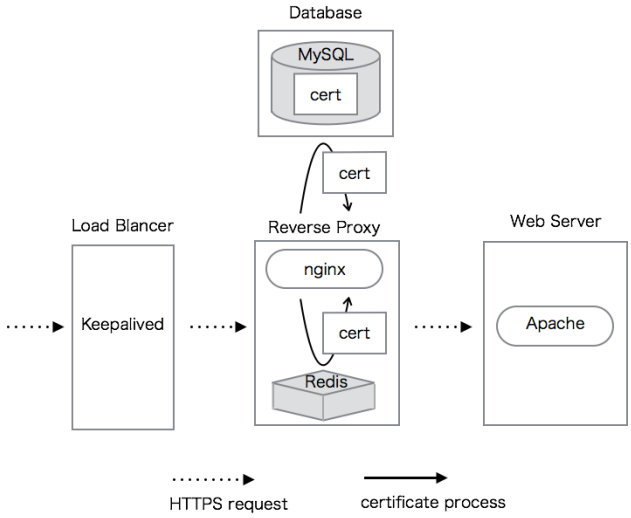


Fig. 5 System of dynamic server certificate management.

Table 4 Production server specifications.

	Specifications
CPU	Intel Xeon CPU E5-2430 v2 2.50 GHz 12 thread
Memory	32 GBytes
Server	NEC Express5800/E120e-M

We developed a production environment shown in **Fig. 5** based on Fig. 3. In addition, the server hardware adopted for the preloading method and the dynamic loading method have the same specifications.

In the experiment of the production environment, Apache httpd is reloaded in the conventional environment when new hosts are added, or operational configuration changes occur. In the conventional environment, about 400 sites of the service users' sites were compatible with HTTPS in one month. The number of certificates also increased by 400. On the other hand, in the new production environment after the replacement using nginx with the proposed method, the web server did not reload for the addition of hosts because the addition is processed dynamically. The web server reloads periodically to release the memory every day. Also, in the new production environment, the number of certificate increases in one month was about 10,000.

In this section, we show reference values about how much resources are used in production for companies and engineers considering using the proposed method, not superiority/inferiority of each method.

Table 4 presents the server specifications. **Figure 6** depicts the transition of the number of certificates in a month. Figure 6 shows the transition of the number of web sites supporting HTTPS, that is, the total number of SSL/TLS certificates in the hosting service production environment. **Figure 7** portrays the transition of the number of requests processed per second. **Figure 8** shows the transition of CPU utilization of the server. **Figure 9** shows the

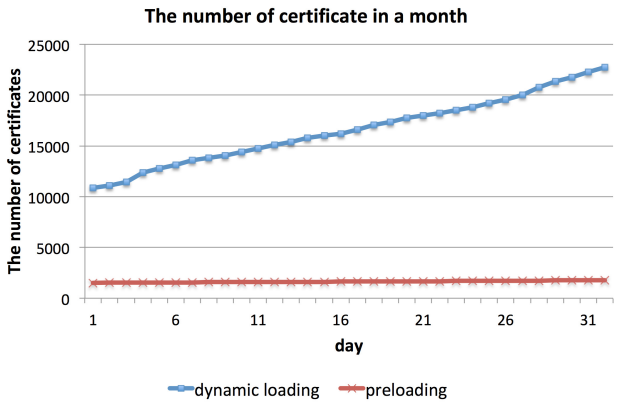


Fig. 6 The transition of the number of web sites/certificates supporting HTTPS in a month.

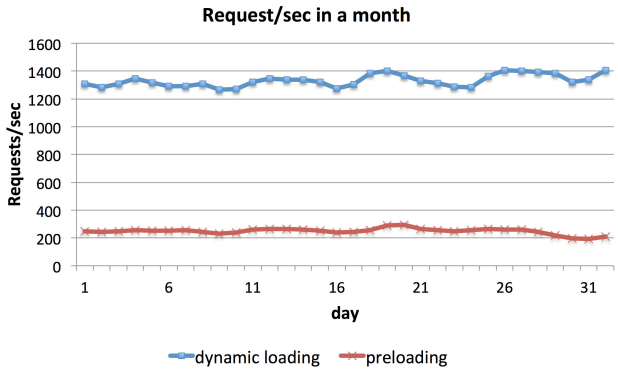


Fig. 7 HTTPS Requests per second in a month.

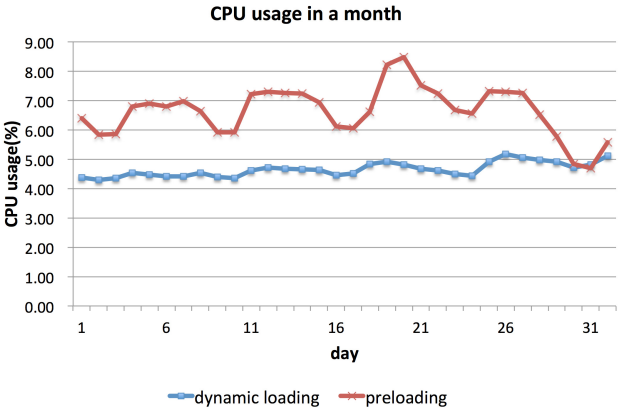


Fig. 8 CPU usage in a month.

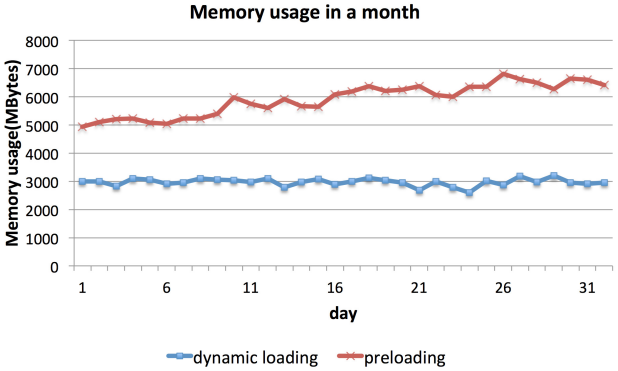


Fig. 9 Memory usage in a month.

transition of the server memory usage.

Figure 6 shows that the number of certificates increased by about 400 during one month using the preloading method with the existing method. In that case, as described in Section 4.1, Fig. 9 shows that the usage has increased by about 1 GByte. The dynamic loading method of the proposed method of Fig. 6 shows that the number of certificates has increased by more than 10,000 in one month. The increase in the number of certificates resulted from a newly provided free HTTPS certificate service of our employer. The number of certificates of servers processed using the dynamic loading method is 10–15 times that of the preloading method.

The number of requests per second is more than six times, as shown in Fig. 7. In system replacement in the production environment, we changed from Apache httpd using the existing method to nginx using the proposed method using the same hardware. From the observation result, as shown in Fig. 7, we consider that the average of the number of requests per second of the existing method is lower than the proposed method because reloading of the web server frequently occurs when settings change or due to a certificate addition. However, as shown by Figs. 8 and 9, the transition of CPU usage and memory usage is less than in the server which was processing using the existing preloading method. This result only shows the indication of the resource usage when using Apache httpd with the existing method and when using nginx with the proposed method in production because the difference in the CPU usage heavily depends on the web server implementation.

Regarding the difficulty of increasing the memory usage depending on the number of certificates in the preloading method, as shown by Fig. 9, the proposed method does not markedly increase the memory usage. The memory usage can be reduced greatly using the proposed method because it is not necessary to read a certificate of a domain without access by loading only the certificate corresponding to the domain HTTP requested from clients.

The memory usage of the existing method using Apache httpd is increased by about 1 Gbytes against the increase of 400 pairs of server certificates and secret keys. In Section 4.1, the memory usage of the existing method using nginx is about 3 Gbytes for 100,000 pairs of server certificates and secret keys. The result of Section 4.1 is the memory usage immediately after loading a certificate of 100,000 domains at startup and activating nginx. Therefore, nginx itself is not handling an HTTPS request at all. On the other hand, Apache httpd has already processed many HTTPS requests with the existing method in production. In addition to the certificate and secret key data, Apache httpd allocates memory for a TLS session cache required for processing requests, Apache httpd configuration and processing data of each HTTPS request, and other modules data in production. In order to apply such a configuration change quickly, reload/graceful-restart was executed every 15 minutes when there is a change. Therefore, the memory usage per pair of server certificates and secret keys in Section 4.3 is more than 80 times that in Section 4.1.

It is possible to reload with the graceful restart command by nginx, the server process online, without failing the request be-

cause the time required for reloading the configuration of the web server process is greatly shortened. By decreasing this time, the proposed method can release the memory usage freely. The memory usage as a whole can be reduced.

In a highly integrated multi-tenant web server, the reload/ graceful restart function does not operate properly when the number of certificates becomes enormous. Depending on the web server implementation, the graceful restart function to reload the configuration without missing the request normally completes itself in seconds. However, by the existing method, when the number of certificates to be loaded at the time of activation increases, it takes several tens of seconds of time to reload, or even several minutes. Even if graceful restarting, the service is stopped because of the request timeout.

4.4 Discussion of the Evaluation Result

Figures 6 and 9 show that the number of certificates increases by about 400 in one month using the preloading method and that the memory increases by about 1 GByte. The breakdown of the memory increases the amount per certificate, the configuration of the host, the data of the certificate and the secret key, and the memory amount used when the web server processes HTTPS requests. In other words, when trying to process 20,000 server certificates using the preloading method, it is calculated that 50 GBytes of additional memory are required. The proposed method can process 20,000 certificates with about 3 GBytes, so that the resource usage can be improved greatly.

When using the server equipped with 32 GBytes of memory in Table 4, if the number of certificates reaches 200,000 or higher in the future, then the existing method would require over 500 GBytes of memory from the viewpoint of the memory usage described above. In other words, in the existing method, more than 15 servers with 32 Gbytes of memory installed are required.

However, using the proposed method, Fig. 9 shows that the memory usage depends only slightly on the number of certificates. From the viewpoint of memory usage, this result shows that even if the number of certificates reaches 200,000, even one server can process it. Using the proposed method, under circumstances in which future HTTPS communication becomes commonplace, the number of servers can be greatly reduced.

With the existing method, the time for reloading the server process increases as the number of certificates increases. Consequently, the difficulty arose that the service stoppage time attributable to reloading becomes long when loading a new configuration or registering a new certificate.

However, in the proposed method, the proposed method can reload the process in a short time and can shorten the service downtime because the certificate is not loaded at the server process startup.

Therefore, the service stoppage time can be shortened overall using the proposed method. It becomes possible to adopt a system configuration that is easy to operate.

5. Conclusion

Because RFC adoption of HTTP/2 protocol is required on HTTPS, the existing sites must support HTTPS. The existing method takes time to start up because a highly integrated multi-tenant web server must load numerous server certificates at a server process startup.

The proposed method dynamically loads the server certificate and secret key corresponding to the requested hostname using SNI during the SSL/TLS handshake. It then communicates via HTTPS. Using the proposed method, the server can communicate via HTTPS without loading numerous server certificates at startup. Moreover, the cost of dynamically loading a certificate is low compared to the cost of CPU usage time from the whole SSL/TLS handshake. The experimentally obtained results demonstrate that the performance does not lead to difficulties in practical use.

As a result of introducing the proposed method to the production environment of a hosting service, resource usage can be greatly reduced compared with that necessary for the existing method. Also, the proposed method is sufficiently effective to support the operation of the production environment.

Furthermore, even if the performance becomes insufficient because of the processing of HTTPS, the proposed method can readily scale up servers using the scale-out model through a centralized management of the server certificate data with a reverse proxy put in front of the HTTPS servers.

We conclude that the proposed method is a promising practical system design for supporting HTTPS of a highly integrated multi-tenant architecture.

References

[1] Belshe, M., Thomson, M. and Peon, R.: Hypertext Transfer Protocol Version 2 (HTTP/2), RFC 7540 (2015).

[2] Bowen, R. and Coar K.: Apache Cookbook. O'Reilly and Associates (2003).

[3] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R. and Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC 5280 (2008).

[4] Eastlake, D.: Transport Layer Security (TLS) Extensions: Extension Definitions, RFC 6066 (2011).

[5] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D. and Falsafi, B.: Clearing the clouds: A study of emerging scale-out workloads on modern hardware, *ACM SIGPLAN Notices*, Vol.47, No.4, pp.37–48 (2012).

[6] Grigorik, I. and Far, P.: Google I/O 2014 – HTTPS Everywhere, available from ⟨https://www.youtube.com/watch?v=cBhZ6S0PFCY⟩.

[7] Internet Security Research Group (ISRG): Let's Encrypt – Free SSL/TLS Certificates, available from ⟨https://letsencrypt.org/⟩.

[8] Han, J., Haihong, E., Le, G. and Du, J.: Survey on NoSQL database, *2011 6th International Conference on Pervasive Computing and Applications* (*ICPCA*), pp.363–366 (2011).

[9] Let's Encrypt Community Support: Apache Module mod_vhost_alias & LE, available from ⟨https://community.letsencrypt.org/t/apache-module-mod-vhost-alias-le/9476⟩.

[10] Mozilla Project: Mozilla wiki Security/Server Side TLS, available from ⟨https://wiki.mozilla.org/Security/Server_Side_TLS⟩.

[11] Mietzner, R., Metzger, A., Leymann, F. and Pohl, K.: Variability Modeling to Support Customization and Deployment of Multi-tenant-aware Software as a Service Applications, *2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pp.18–25 (2009).

[12] Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M. and Steenkiste, P.: The cost of the S in HTTPS, *10th ACM International on Conference on Emerging Networking Experiments and Technologies* (*CoNEXT '14*), pp.133–140, ACM (2014).

[13] NPO mruby forum, available from ⟨http://forum.mruby.org/⟩.

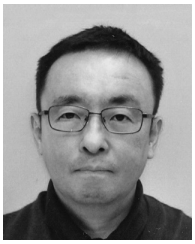
[14] Nginx: Nginx, available from ⟨http://nginx.org/ja/⟩.

[15] OpenSSL Software Foundation: OpenSSL, available from ⟨https://www.openssl.org/⟩.

[16] OpenSSL Software Foundation: SSL_CTX_set_client_cert_cb, SSL_CTX_get_client_cert_cb - handle client certificate callback function, available from ⟨https://www.openssl.org/docs/man1.0.2/ssl/SSL_

CTX_set_client_cert_cb.html⟩.
[17] Prodan, R. and Ostermann, S.: A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers, *10th IEEE/ACM International Conference on Grid Computing*, pp.17–25 (2009).
[18] Matsumoto, R.: Studies on Highly Integrated Multi-Tenant Architecture for Web Servers, available from ⟨https://dx.doi.org/10.14989/doctor.k20590⟩, Kyoto University, Ph.D. Thesis (2017).
[19] Matsumoto, R.: ngx_mruby: Support ssl_handshake handler and dynamic certificate change, available from ⟨https://github.com/matsumotory/ngx_mruby/pull/145⟩.
[20] Matsumoto, R. and Okabe, Y.: mod_mruby: A Fast and Memory-Efficient Web Server Extension Mechanism Using Scripting Language, *IPSJ Journal*, Vol.55, No.11, pp.2451–2460 (2014).
[21] Matsumoto, R., Kawahara, M. and Matsuoka, T.: Improvement of Security and Operation Technology for a Highly Scalable and Large-scale Shared Web Virtual Hosting System, *IPSJ Journal*, Vol.54, No.3, pp.1077–1086 (2013).
[22] Reese, W.: Nginx: The high-performance web server and reverse proxy, *Linux J.*, No.9, pp.1–4 (2008).
[23] Sanfilippo, S. and Noordhuis, P.: Redis, available from ⟨https://redis.io/⟩.
[24] The Apache Software Foundation: Apache HTTP Server, available from ⟨http://httpd.apache.org/⟩.
[25] The Apache Software Foundation: ab – Apache HTTP server benchmarking tool, available from ⟨https://httpd.apache.org/docs/2.4/programs/ab.html⟩.
[26] The Apache Software Foundation: Apache HTTP Server Version 2.4 Apache Module mod_ssl, available from ⟨http://httpd.apache.org/docs/current/mod/mod_ssl.html⟩.
[27] The Apache Software Foundation: Apache Virtual Host documentation, available from ⟨http://httpd.apache.org/docs/2.2/en/vhosts/⟩.
[28] Glozer, W.: wrk – A HTTP benchmarking tool, available from ⟨https://github.com/wg/wrk⟩.

Ryosuke Matsumoto received his Ph.D. degree in informatics from Kyoto University, Kyoto, Japan, in 2017. From 2015 to 2018, he worked for Pepabo Research and Development Institute, GMO Pepabo, Inc., as a chief engineer and a chief researcher. From 2018, he is currently a senior researcher at SAKURA Research Center, SAKURA Internet Inc. His research interests include OS, middleware, Internet operation technology, and security. He is a member of IEEE, ACM, and IPSJ.



Kenji Rikitake is the Founder of Kenji Rikitake Professional Engineer's Office. He is a Gijyutsushi (Japan's government-licensed professional engineer) of Information Engineering since 2001, and a Registered Information Security Specialist of Japan since 2018. He received B.Eng. and M.Eng. degrees from the University of Tokyo in 1988 and 1990, and received a Ph.D. in Information Science from Osaka University in 2005. He was a Professor of Academic Center for Computing and Media Studies and the Institute for Information Management and Communication (ACCMS/IIMC) of Kyoto University, from 2010 to 2013. His research interests include distributed systems, Erlang/OTP, Internet security, and radio engineering. He is ACM Senior Member, IEICE Senior Member, and a member of IPSJ and IPEJ.

Kentaro Kuribayashi is a director of GMO Pepabo, Inc., and is in charge of technology and engineering as CTO. He is a Registered Information Security Specialist (RISS) since 2018. He received a Bachelor's Degree in Law from Tokyo Metropolitan University, Japan in 1999. From 2008, he started his actual career as a software engineer. In 2015, he launched Pepabo Research and Development Institute, GMO Pepabo, Inc. and has been the head of the institute. He is interested in wide range of emerging technology, especially, information security, blockchain-based architecture, speech I/O, and so on. He is a member of IPSJ and JSAI.