Vol.2017-IOT-39 No.13
Vol.2017-SPT-25 No.13
2017/9/29

高集積マルチテナント Web サーバの大規模証明書管理と実運用上の評価

松本 亮介[1,a)]　平原 正裕[3]　三宅 悠介[1]　力武 健次[1,2]　栗林 健太郎[1]

概要：インターネットの利用に際して，ユーザーや企業においてセキュリティ意識が高まっている．また，HTTP のパフォーマンス上の問題を解消するために，HTTP/2 が RFC として採択された．それらを背景に，常時 HTTPS 化が進む中で，高集積マルチテナント方式の Web サーバで管理している大量のホストも HTTPS 化を進めていく必要がある．同方式は単一のサーバプロセスで複数のホストを管理する必要があるが，Web サーバの標準的な設定を用いて事前にホスト数に依存した数の証明書を読み込んでおく方法では，常に確保しておく必要のあるメモリ使用量が増大することで，リソース効率が低くなり，収容数が低下する．そこで，Server Name Indication(SNI) を利用可能である条件下において，事前にサーバプロセスに証明書を読み込んでおくことなく，SSL/TLS ハンドシェイク時にホスト名から動的にホストに紐づく証明書を読み込み，メモリ使用量を低減させる手法を提案する．実装には，我々が開発した，mruby を用いて高速かつ少ないメモリ使用量で Web サーバの機能を拡張するモジュール ngx_mruby を採用して，動的にサーバ証明書を選択する機能を実装した．また，筆者が所属する GMO ペパボ株式会社のホスティングサービスにおいて本手法を導入し，証明書の数やサーバリソースの使用量の関係性，性能に関する評価を行った．

Large-scale Certificate Management on Highly-integrated Multi-tenant Web Servers and The Evaluation on A Production System

Ryosuke Matsumoto[1,a)]　Masahiro Hirabaru[3]　Yusuke Miyake[1]　Kenji Rikitake[1,2]
Kentaro Kuribayashi[1]

Abstract: Introducing HTTPS to a large number of the hosts supervised under highly-integrated multi-tenant Web servers is critical to meet the security demand of the individual and corporate users, and to comply with the HTTP/2, an RFC to solve the HTTPS performance issues. Preloading the massive number of certificates for managing a large number of hosts under the single server process results in increasing the required memory usage due to the respective page table entry manipulation, which may be poor resource efficiency and reduced capacity. To solve this issue, we propose a method to dynamically load the certificates bound to the hostnames found during the SSL/TLS handshake sequences without preloading, provided the Server Name Indication (SNI) extension is available. We implement the function of choosing the respective certificates with ngx_mruby module, which we developed to extend Web server functions using mruby with small memory footprint while maintaining the execution speed. We also introduced the proposed method to the Web hosting services of GMO Pepabo, Inc., authors' employer, and evaluated the relationship between the numbers of certificates and the server resource usage, and the performance.

[1] GMO ペパボ株式会社 ペパボ研究所
Pepabo Research and Development Institute, GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka 810-0001 Japan
[2] 力武健次技術士事務所
Kenji Rikitake Professional Engineer's Office, Toyonaka City, Osaka 560-0043 Japan
[3] GMO ペパボ株式会社 ホスティング事業部

1. はじめに

HTTPS による通信が前提となる HTTP/2 プロトコルの RFC 採択 [1] と Google による常時 SSL/TLS 化の推進 [4]

[a)] matsumotory@pepabo.com

に伴い，Web ホスティング事業者によって管理されている Web サイトの HTTPS 化が急務となっている．一般に HTTPS 化は事業者にとってもサービス利用者にとっても，サーバ証明書の価格の高さや，HTTPS を行うための基盤整備のコストが高いとされてきた [10]．しかし，Let's Encrypt[5] などのような無料の DV 証明書の提供が開始されはじめ，比較的低コストで HTTPS 化が実現可能になってきている．

高集積マルチテナント方式 [9] による Web ホスティングサービスでは，高集積にホストを収容することで，ハードウェアコストや運用コストを低減し低価格化を実現するために，単一のサーバプロセス群で複数のホストを管理する必要がある [29]．ここでいう単一のサーバプロセス群とは，ホスト毎にサーバプロセスを起動させるわけではなく，複数のホストでサーバプロセスを共有することを示す．実際にサーバプロセスの処理を行うプロセスは，ホスト数には依存しないものの，Web サーバの実装によっては数十から数百存在することが多い．従来の Web サーバソフトウェアは HTTPS で通信を行うために，サーバ起動時に，サーバ証明書とペアとなる秘密鍵をホストごとに読み込んでおく必要 [21] がある．しかし，そのような仕組みでは，高集積マルチテナント方式でのメリットである性能と低価格化の両立が難しい．なぜなら，高集積にホストを収容すると，大量のサーバ証明書の読み込みによってサーバプロセスの起動に多くの時間を要したり，サーバプロセスのメモリ使用量がホスト数に依存して増加したりするからである．また，サーバ証明書をファイルで管理する必要があり，複数の Web サーバによる処理の分散や可用性の担保に支障をきたす．

本論文では，高集積マルチテナント方式による Web サーバにおいて，TLS 拡張の Server Name Indication(SNI)[2] を前提に，Web サーバプロセス起動時にサーバ証明書と秘密鍵を読み込んでおくのではなく，SSL/TLS ハンドシェイク時において，リクエストのあったホスト名を元に，リクエスト単位で対応するサーバ証明書と秘密鍵のデータをデータベースから動的に取得することで，Web サーバプロセスのメモリ消費量を大幅に低減する効率的なサーバ証明書の管理アーキテクチャを提案する．SSL/TLS ハンドシェイク時における証明書と秘密鍵の動的な読み込みは，筆者らが開発した，nginx[11] を mruby[26] で拡張できる ngx_mruby[28] に，証明書の取り扱いを制御できる機能追加を行うことで対応した．サーバ証明書と秘密鍵は KVS[6] の一種である Redis[16] に保存しておき，mruby のコードによってホスト名に対応した証明書と秘密鍵を取得するようにした．本手法は，HTTPS 通信を終端する Web サーバとして広く使われている nginx に対して，ngx_mruby を用いることにより nginx 本体を変更することなく簡単に組み込めるため，実用的である．また，実装は既に OSS として公開済みである[*1]．本論文は，筆者の研究報告 [30] の発表をさらに発展させたものである．

本論文の構成を述べる．2 章では，高集積マルチテナント方式の Web サーバにおける常時 HTTPS 化に向けた課題を整理する．3 章では，2 章の課題を解決するための提案手法のアーキテクチャおよび実装を述べる．4 章では，実験環境で従来手法の問題についての定量的検証を行った上で，従来手法と提案手法の性能比較を行って有効性を評価した．5 章では筆者が所属する GMO ペパボ株式会社のホスティングサービスの本番環境に実際に導入して，1 ヶ月間動作させ，従来手法を採用していた時期と比較した実運用上の評価を行い，6 章でまとめとする．

2. 従来の高集積マルチテナント方式のサーバ証明書管理

高集積マルチテナント方式の代表的なサービスである Web ホスティングとは，複数のホストでサーバのリソースを共有し，それぞれの管理者のドメインに対して HTTP サーバ機能を提供するサービス [14] である．Web ホスティングサービスにおいて，ドメイン名 (FQDN) によって識別され，対応するコンテンツを配信する機能をホストと呼ぶ．本論文では，単一のサーバプロセスで複数のホストを処理する仮想ホスト方式 [23] を採用したマルチテナントアーキテクチャにおいて数万以上のホストを収容できるものを高集積マルチテナントアーキテクチャと呼ぶ．

代表的な Web サーバソフトウェアである Apache httpd[19] や nginx では，仮想ホスト方式によって，単一のサーバプロセスで複数のホストを処理することができる．従来の Web サーバのサーバ証明書管理では，Web サーバプロセスの起動時にホストに紐づく証明書を読み込み，HTTPS 接続時に IP アドレスあるいはホスト名に対応した証明書をメモリ上から読み出し，SSL/TLS ハンドシェイクによってセッションを確立する．この管理方法の場合，事前にメモリ上に証明書を読み込んでおくため，SSL/TLS ハンドシェイク時に高速に処理することができる．

高集積マルチテナント方式は大量のホストを管理する必要があるため，仮想ホスト方式を利用してできるだけ設定内容やプロセスの起動アーキテクチャをホスト数に依存しない構成にする必要がある．実運用上では，一台のサーバにホスト数を数万以上収容することもあり，従来手法では，サーバ起動時に大量の証明書と秘密鍵をメモリ上に読み込んでおく必要がある．また，前段に TLS 専用のリバースプロキシを配置するようなシステム構成では，後段の複数のホスティング用のサーバに収容された全てのホストのドメインに対して，リバースプロキシ上で最初に TLS 通信をする必要があるため，リバースプロキシ上で数十万の単

[*1] https://github.com/matsumotory/ngx_mruby

位で証明書を管理する必要がある．その場合，サーバ証明書の読み込み数が増大することにより，起動時の読み込み時間が大幅に増え，サーバプロセスのメモリ使用量も大幅に増えることが大きな問題となる．また，ホストに紐づく証明書の設定を全て記述する必要があり，Web サーバ設定の行数も大幅に増えるため，設定ファイルの可読性が低くなり、サーバ管理に支障をきたす．

そのような課題から，従来手法では，収容数の増加に伴い，サーバ台数を増やしたり，一台のサーバに対するホストの収容数を減らしたりする必要がある．また，サーバプロセスの設定再読み込みにも時間がかかるため，サービス停止の時間が長くなってしまう．

例えば nginx を利用している場合に，10 万ホストの仮想ホストを設定し，各ホストに紐づく証明書と秘密鍵を読み込んだ場合，nginx の設定行数は 200 万行程度になる上に，サーバプロセスの起動は 50 秒かかる．そのため，設定変更時のサーバプロセスの再起動に，非常に時間がかかる．この起動時間に関する詳細については，4 章で言及する．

ここまで述べたことから次のことがわかる．HTTP/2 の実用化に伴い HTTPS 通信が既定の方式として使われる Web サービスにおいて，従来の Web サーバプロセス起動時に静的にサーバ証明書を読み込んでおく方式では，高集積マルチテナント方式を採用する Web サービスにおいて，サーバプロセス起動に時間がかかる問題と，ホストの収容数に比例してメモリ使用量が増加する問題がある．

3. 提案手法

3.1 効率的なサーバ証明書の管理アーキテクチャ

コンピュータリソースと性能効率のバランスの最大化，および，システム運用コストの効率化が求められる高集積マルチテナント方式において，2 章で述べた課題を解決するためには，以下の 3 つの要件を満たすことが必要である．

(1) 高集積化を実現するために，IP アドレスではなくホスト名に紐づくサーバ証明書を用いる Server Name Indication(SNI) 拡張を利用する．
(2) Web サーバプロセス起動を高速にするために，サーバ証明書は起動時に読み込まない．
(3) Web サーバプロセスのメモリ使用量がホスト数に依存しないようにするために，サーバ証明書は SSL/TLS ハンドシェイク時に動的に読み込む．

SNI[2] とは，SSL/TLS の拡張仕様の一つである．通常 TLS による通信では，IP アドレス単位でサーバ証明書を利用するため，高集積マルチテナント方式では，ホストの数だけ IP アドレスが必要となり，IP アドレスの取得コストを考慮すると，コストの制約を満たしながら低価格化を実現するという要件に向かない．一方，SNI では SSL/TLS ハンドシェイク時にアクセスしたいホスト名をサーバに伝えることにより，従来の IP アドレス単位ではなくホスト名単位でサーバ証明書を使い分けることができる．SNI は，高集積マルチテナント方式のように，単一のサーバプロセスかつ単一の IP アドレスで複数のホストを仮想的に処理するような方式において，各ホストとホスト名を用いて HTTPS 通信を行う場合に利用されることが多い．

そこで，SNI による SSL/TLS ハンドシェイクを前提に，HTTPS によって Web サーバにリクエストがあった場合に，ホスト名にもとづいてデータベース上からサーバ証明書と秘密鍵を取得し，SSL /TLS ハンドシェイクを行う．このようにすることで，高集積マルチテナント方式のように大量のサーバ証明書が必要な状況において，事前にサーバ証明書を読み込んでおく必要がないため，Web サーバプロセスの起動は速く，メモリ使用量もホスト数に依存して増加することがないため少なくて済む．サーバプロセスの起動が速いことから，証明書数が増加した状況で，サーバの設定の変更を適用したい場合に，サーバプロセスの再読み込みに長時間時間がかかる問題も解決できる．さらに，データを TCP 通信可能なデータベースやキャッシュに集約しておくことにより，HTTPS リクエストが増加してきた際に，Web サーバを複数台に増加させて容易に可用性と性能を担保することができるとともに，実サービスにおけるユーザとの SSL/TLS の契約から証明書を Web サーバに設定するシステムとの連携も，データベースを介して容易に実現できる．

3.2 提案手法の実装

提案手法の実装には，nginx の機能拡張を mruby で記述でき，高速かつ少ないメモリ使用量で動作する ngx_mruby を利用した．また，OpenSSL ライブラリ [12] のバージョン 1.0.2 以降からは SSL/TLS ハンドシェイク時にサーバ証明書や秘密鍵を読み込むような関数をコールバックできる関数 SSL_CTX_set_cert_cb()[13] が利用できる．この関数を使い，nginx における SSL/TLS ハンドシェイク時に呼び出すコールバック処理を，mruby で記述できるようにした [15]．これにより，サーバ管理者はシステムの用途に合わせて簡単に動的証明書の実装ができる．

ngx_mruby を利用した，SSL/TLS ハンドシェイク時に動的にサーバ証明書と秘密鍵を読み込む実装例を示す．**図 1** は，動的にサーバ証明書を，リクエストのあったホスト名からファイルのパスを決定して読み込む例である．Nginx::SSL のインスタンスの certificate メソッドおよび certificate_key メソッドにファイルのパスを渡すことで，SSL/TLS ハンドシェイク時に動的にサーバ証明書と秘密鍵を読み込むことができる．**図 2** は，サーバ証明書を Redis という Key-Value Store(KVS) に保存しておき，ホスト名からデータのキーを決定し，キーに紐づくサーバ証明書データを取得する例である．certificate_data メソッドおよび certificate_key_data メソッドは，サーバ証明書や秘密鍵

```
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"
  ';
}
```

図 1 動的なサーバ証明書読み込みの設定例 (ファイルベース)
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"]
  ';
}
```

図 2 動的なサーバ証明書読み込みの設定例 (KVS ベース)
Fig. 2 KVS-based Configuration Example of Dynamic Server Certificate Management.

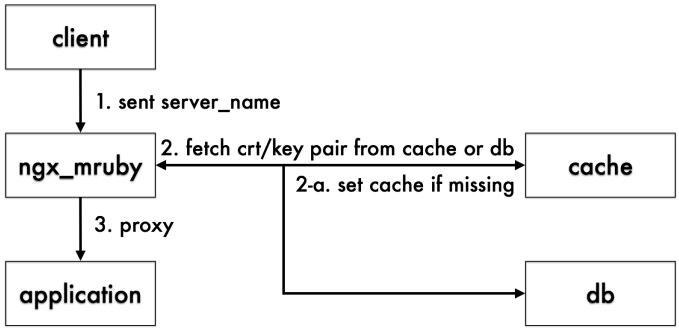

図 3 動的なサーバ証明書読み込みのシステム例
Fig. 3 System Example of Dynamic Server Certificate Management.

のデータを直接渡すことにより，SSL/TLS ハンドシェイク時に動的に読み込むことができる．

表 1 実験環境
Table 1 Experimental Environment.

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

実運用における設計例としては，**図 3** のように，サーバ証明書や秘密鍵のデータはデータベースに保存しておき，nginx が HTTPS リクエストを受信した際，ngx_mruby 経由で SSL/TLS ハンドシェイク時にデータベースから取得したサーバ証明書と秘密鍵を取得して SSL/TLS セッションを確立する．その際に，リクエスト毎にデータベースへ接続するコストを低減するため，保存できるデータサイズは少ないものの，高速にデータの取り出しを行える Redis のような KVS を使ってキャッシュとして一時的にデータを保存しておく．また，データベース及びキャッシュサーバと TCP 接続を行うことにより，可用性や性能のためにサーバ台数を増やしても，簡単にサーバ証明書に関するデータを共有できる．複数台の Web サーバで HTTPS 通信の負荷分散を行う場合に，Web サーバからキャッシュサーバへのネットワークレイテンシが性能面で問題となる場合は，データベースやキャッシュサーバに加えインメモリキャッシュを用いたシステム設計も可能である．

4. 実験と考察

本手法の有効性を確認するために，2 章で述べた従来の課題において，Web サーバプロセス起動時に予めデータを読み込んでおく従来の方式 (preload) の，起動時間とメモリ使用量に関する問題を，実験から明らかにする．続いて，Redis にサーバ証明書と秘密鍵データを保存しておき， SSL/TLS ハンドシェイク毎にデータをファイルや Redis から取得する提案手法 (dynamic load) と，従来の方式 (preload) の性能を比較する．**表 1** に実験環境を示す．

4.1 従来手法のメモリ使用量と起動時間の検証

2 章で述べた課題について，表 1 の環境において，nginx のバージョン 1.11.13 を利用して検証する．openssl コマンドによって生成した 10 万ホスト分の 4096bits の鍵長のサーバ証明書と秘密鍵を nginx のホスト設定毎に記述し，nginx のサーバプロセスの起動時間を計測した．nginx は，最初に起動する master プロセスがサーバ証明書のデータなどを全て読み込み，その処理が完了後にリクエスト処理を担当する worker プロセスが fork() システムコールにより複製される．本実験環境では，worker プロセスは CPU の論理コアの数だけ起動させるように設定し，24 個の worker プロセスが起動を完了するまでの時間を，Linux の time コマンドにより，実時間，システム CPU 使用時間，および，

表 2 従来手法のプロセスの起動時間と検証結果
Table 2 Result of Startup Time by Existing Method.

項目	値
プロセス起動の実時間	42.662 sec
プロセス起動のユーザ CPU 使用時間	37.280 sec
プロセス起動のシステム CPU 使用時間	5.387 sec

ユーザ CPU 使用時間を計測した．

表 2 に結果を示す．サーバ証明書等の読み込みは最初に単一の master プロセスが処理するため，CPU を一つだけしか利用できず，コアあたりの性能に依存する．そのため，CPU の使用効率をコア数で高めていく時代においては，この処理時間を大幅に短縮することは困難である．ユーザ CPU やシステム CPU の使用時間については特筆すべき点は見当たらなかった．

4.2 提案手法の性能評価

Web サーバプロセス起動時に静的にサーバ証明書等を読み込む方式による，メモリ使用量と起動時間が増加する問題を解決するための提案手法について，評価を行った．評価には，Web サーバソフトウェアとして ngx_mruby を組み込んだ nginx を用いた．**図 4** に ngx_mruby の設定を示す．ポート 58085 で Listen する設定は，提案手法によって SSL/TLS ハンドシェイク時に，リクエストされたホスト名をキーにサーバ証明書と秘密鍵を読み込む設定である．また，ポート 58086 は Web サーバプロセス起動時にサーバ証明書等を静的に読み込んでおく従来手法の設定である．両方の設定は，サーバ側での cipher suites を固定し，SSL/TLS ハンドシェイク時の影響を最大化するために，SSL/TLS セッションキャッシュが生じないようにした．従来手法の設定と提案手法の設定双方で読み込む証明書を一つにしているのは，複数の証明書であっても従来手法は nginx の中でハッシュとして扱われるため計算量は O(1) であり，KVS からドメインをキーに証明書を取得する処理も O(1) であるため，一つの証明書に関するデータで評価には必要十分であると推察したためである．

性能を比較するために，wrk[24] という HTTPS のベンチマークソフトウェアを使い，同時接続数を変化させながら総リクエストで 500 万リクエスト送信し，1 秒間に処理できるリクエストの数を計測した．一般的にベンチマークで広く使われる ab[20] コマンドは，シングルスレッドで動作するため，HTTPS のベンチマークを行う場合は，サーバソフトウェアよりも先にベンチマークコマンドが SSL/TLS ハンドシェイク時の CPU 使用時間によって一つの CPU コアを使い果たしてしまう．そこでこの問題を回避するため，マルチスレッドで動作する wrk を採用した．TLS のバージョンは TLSv1.2 を利用し，cipher suites は，現在セキュリティを担保するために Mozilla が推奨している cipher suites[8] の中から ECDHE-RSA-AES128-GCM-SHA256 を利用した．リクエストするコンテンツは nginx に同封されている 612Bytes の index.html を利用した．

```
# 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/;
  }
}
```

図 4 動的読み込みと事前読み込みの設定
Fig. 4 Configuration of Dynamic Loadding and Preloading.

表 3 実験結果
Table 3 Experimental Result of Proposed Method.

同時接続数	提案方式 dynamic load(req/sec)	従来方式 preload(req/sec)
10	171456.60	171914.98
100	172383.84	172758.28
500	172714.81	173631.06
1000	171872.24	173272.53

表 3 に結果を示す．実験結果では，事前にサーバ証明

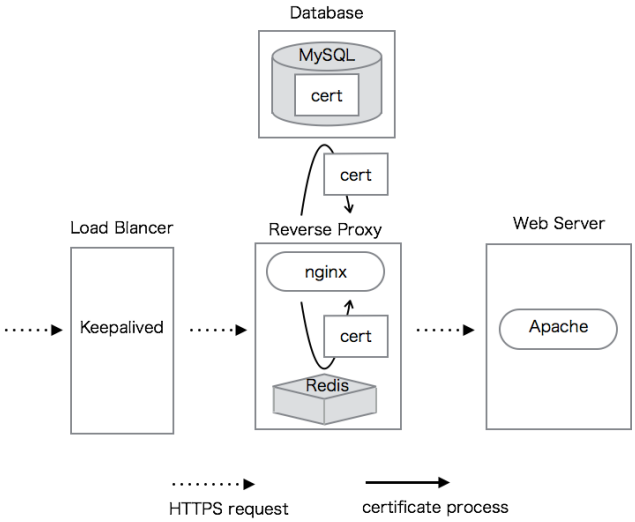

図 5 動的証明書読み込みの構成図
Fig. 5 System of Dynamic Server Certificate Management.

書を読み込む従来方式と今回提案した方式の間に性能差はほとんどないことを観測した．今回採用した cipher suites は，SSL/TLS ハンドシェイク時の鍵交換時に利用される暗号化アルゴリズムとして RSA を利用しており，暗号化と複合の処理が証明書を動的に読み込むか静的に読み込んだメモリ領域から取得するかの処理と比較してほとんど無視できる程度の処理であるためだと考えられる．また，同時接続数が 1000 の場合に従来方式と提案方式共に多少性能が劣化しているが，その差異も 1%未満であるため誤差の範囲だと判断した．

従来の静的に読み込む方式では，高集積マルチテナント方式においては，ホスト数に比例してメモリ使用量が大きくなる問題がある．これに対して提案手法である動的読み込み方式は Web サーバプロセスがサーバ証明書等のデータを起動時にメモリに保存しておく必要がないため，起動時のメモリ使用量は非常に少なく，アクセスのあったドメインの証明書だけを読み込むため，効率的である．実際に実運用上，ホスト数の増加に伴って，メモリ使用量がどの程度効率化されるかについては，5 章で述べる．

証明書の保存方法については，Redis のようなキャッシュサーバを TCP で接続できることで，HTTP と比較して多くの CPU 処理が必要となる HTTPS においても，サーバを複数台増やす事で簡単にスケールアウト [3] によるサーバ増強が可能となり，運用上のメリットも大きい．

予備実験として，Redis を使う方式以外に，動的にファイルパスをホスト名から決定して読み込む方式も評価を行ったが，ほとんど性能差は見られなかった．

5. 実環境での評価

筆者が所属する GMO ペパボ株式会社のホスティングサービスの HTTPS サイト増加に伴い，提案手法を適用し，実運用上の評価を行った．当該ホスティングサービスでは，提案手法適用前は Apache を利用して起動時に証明書を読み込む静的読み込み方式 (preload) を採用していた．評価方法として，従来の preload 方式を採用していた時期である 2017 年 3 月 4 日から 4 月 4 日の 1 ヶ月間の証明書の累積数，秒間のリクエスト処理数，CPU 使用率，メモリ使用量，それぞれの遷移を計測し，提案手法 (dynamic load) を適用した後の同年 7 月 22 日から 8 月 22 日の 1 ヶ月間に測定した同様の値との遷移を比較することにした．**図 5** に，提案手法を採用したシステム構成図を示す．また，preload 方式と dynamic load 方式で採用したサーバは同一スペックである．**表 4** に，サーバスペックを示す．

図 6 に，一ヶ月間の証明書数の遷移，**図 7** に 1 秒間のリクエスト処理数の遷移，**図 8** にサーバの CPU 使用率の遷移，**図 9** にサーバのメモリ使用量の遷移を示す．

表 4 本番環境のサーバスペック
Table 4 Production server specifications.

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

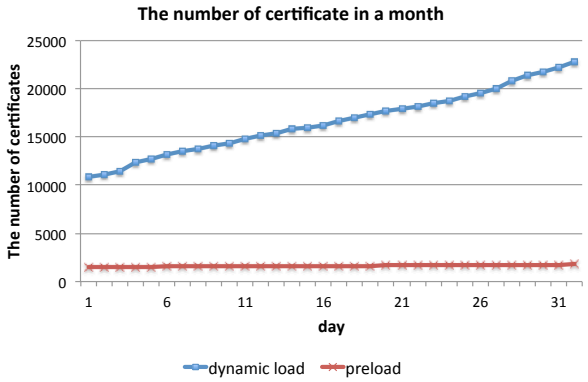

図 6 一ヶ月間の証明書数の遷移
Fig. 6 The number of certificates in a month.

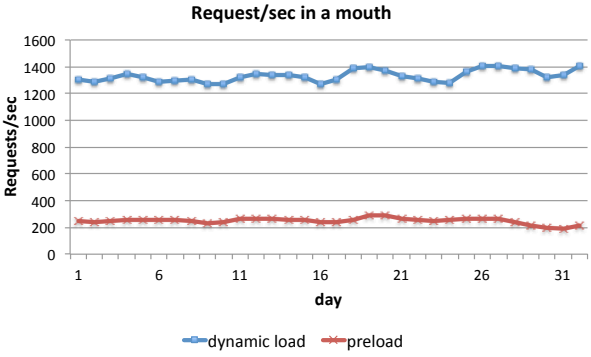

図 7 一ヶ月間の秒間リクエスト数の遷移
Fig. 7 Request per second in a month.

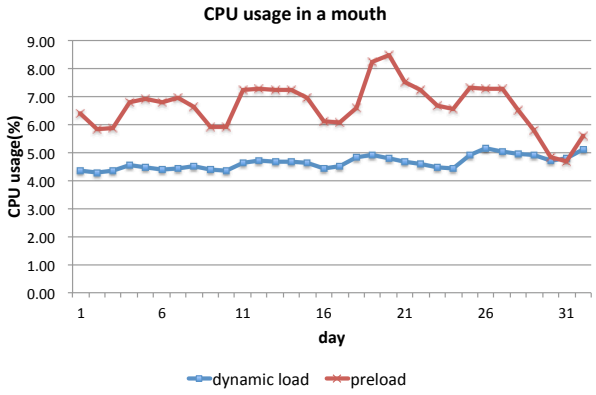

図 8 一ヶ月間の CPU 使用率の遷移
Fig. 8 CPU usage in a month.

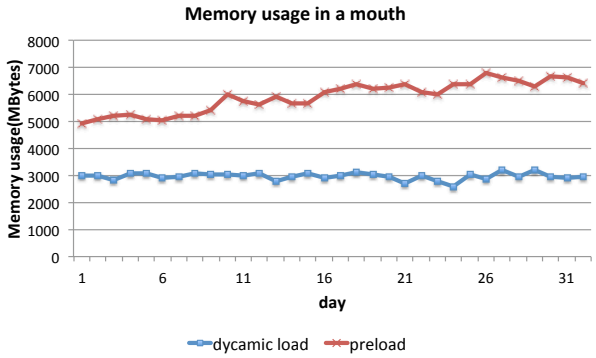

図 9 一ヶ月間のメモリ使用量の遷移
Fig. 9 Memory usage in a month.

図 6 では，従来手法の preload 方式は，1 ヶ月間で証明書の数が約 400 程度増加している．その場合，4.1 節で述べたように，preload 方式は全ての証明書を起動時に読み込むため，図 9 では数に比例してメモリ使用量が増加傾向であり，約 1GBytes 程度増加している．一方， 図 6 の dynamic load が示すように，提案手法の dynamic load 方式は，1 ヶ月間で証明書の数が 1 万以上増えている．これは，無料 SSL オプションサービス[*2]を開始したためである．dynamic load 方式で処理しているサーバの証明書数は preload 方式の 10 倍から 15 倍になっており，秒間のリクエスト数も，図 7 から 6 倍以上であることがわかる．

しかし，CPU 使用量やメモリ使用量の遷移は，図 8 と図 9 より，preload 方式で処理していたサーバよりも少なくなっている．preload 方式で問題となっていた，証明書数に依存してメモリ使用量が増加する課題も，図 9 が示すようにほんとんど増加しておらず，dynamic load 方式では大幅にメモリ使用量を低減できている．これは，アクセスのあったドメインに対応する証明書だけを読み込むことにより，アクセスのないドメインの証明書を読み込む必要がないことが挙げられる．また，Web サーバプロセスの設定再読み込みにかかる時間が大幅に短縮されることにより，サーバプロセスの再読み込みをリクエストを取りこぼすことなくオンラインで行うことができるようになった．この時間短縮により，メモリ使用量を頻繁に解放することも可能となるため，全体としてのメモリ使用量を低減できたと考えられる．Web サーバの実装によるが，通常リクエストを取りこぼすことなく設定を再読み込みするための graceful restart 機能は数秒で処理可能であるが，従来手法によって起動時に読み込む証明書数が増加した場合は，数十秒さらには数分間再読み込みに時間がかかるため，graceful restart とはいえ，リクエストタイムアウトにより，サービスが停止してしまっていた．

仮に，preload 方式で dynamic load 方式が示す証明書の数を処理しようとすると，図 6 と図 9 から，前述の通り preload 方式では 1 ヶ月間で証明書の数が約 400 程度増加し，その結果として図 9 の通りメモリが約 1GBytes 増加していることが分かる．証明書あたりのメモリ増加量の内訳は，ホストの設定，証明書と秘密鍵のデータ，Web サーバが HTTPS のリクエストを処理する際に利用するメモリ使用量と考えられる．つまり，提案手法を適用したサーバの証明書の数 20000 個を従来手法で処理しようとした場合，メモリが 50GBytes 追加で必要になるという計算になる．このことから，提案手法の dynamic load 方式では 20000 個の証明書を 3GBytes 程度で処理できているため，大幅にリソース使用量を改善できている．

表 4 のメモリ 32GBytes 搭載したサーバであれば，今後証明書数が 20 万以上になった場合に，前述したメモリ使用量の観点では計算上，500GBytes 以上のメモリが追加で必要であることから，32GBytes メモリを搭載したサーバも追加で 15 台以上必要であるが，提案手法では，図 9 からもメモリ使用量が証明書数にほとんど依存していないことから，メモリ使用量の観点では，1 台でも処理できると見積もることができる．提案手法によって，今後の HTTPS 通信が当たり前となる状況において，必要なサーバ台数も大幅に削減できることがわかった．

また，従来手法は証明書数の増加に伴いサーバプロセスの再起動に時間がかかったため，新しい設定の読み込みや新規証明書登録の際の，サーバプロセス再読み込みによるサービス停止時間が長くなる課題があったが，提案手法ではサーバプロセス起動時に証明書を読み込まないため，短時間でプロセスの再読み込みが可能となり，サービス停止時間を短縮することができた．また，新規証明書追加の際に，データベースに証明書データを登録すれば，サーバプロセスを再読み込みすることなく HTTPS の適用が可能となるため，全体としてもサービス停止時間を短縮でき，運用し易いシステム構成をとれるようになった．

[*2] https://lolipop.jp/info/news/5759/

6. まとめ

HTTP/2 の RFC 採択に伴い，常時 HTTPS が進む中では，高集積マルチテナント方式を採用している Web サーバにおいて，Web サーバプロセス起動時にホストに紐づくサーバ証明書を大量に読み込む必要があり，Web サーバプロセス起動に時間がかかる問題があった．

提案手法では，SSL/TLS ハンドシェイク時に SNI を前提にリクエストのあったホスト名から該当するサーバ証明書と秘密鍵を読み込み，HTTPS 通信を行うことによって，起動時に大量のサーバ証明書を読み込むことなく動的に HTTPS 通信を行うことができる．また，TLS のハンドシェイク時の CPU 使用時間のコストと比較し，動的に証明書を読み込む処理はコストの低い処理となるため，実用上問題にならない性能がでることを実験から示した．また，実際のホスティングサービスに導入して考察した結果，従来手法と比較して大幅にリソース使用量を低減できたことから，実運用上においても十分に有効性があることがわかった．さらに，サーバ証明書データを前段に配置したリバースプロキシで一元管理することにより，HTTP よりも CPU の処理コストの高い HTTPS 通信においても，SSL/TLS ハンドシェイク時の性能不足を簡単にスケールアウトによるサーバ増強が可能となるため，今後の高集積マルチテナント方式の常時 HTTPS 化を達成するための実運用可能なシステム設計を実現する上で有望な方式の一つということができる．

参考文献

[1] Belshe M, Thomson M, Peon R, Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.
[2] Eastlake D, Transport Layer Security (TLS) Extensions: Extension Definitions, RFC 6066, 2011.
[3] Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, 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, March 2012.
[4] Ilya Grigorik, Pierre Far, Google I/O 2014 - HTTPS Everywhere, https://www.youtube.com/watch?v=cBhZ6S0PFCY.
[5] Internet Security Research Group (ISRG), Let's Encrypt - Free SSL/TLS Certificates, https://letsencrypt.org/.
[6] Han J, Haihong E, Le G, Du J, Survey on NoSQL database. 2011 6th International Conference on Pervasive computing and applications (ICPCA), pp. 363-366, October 2011.
[7] Kegel D, The C10K problem, http://www.kegel.com/c10k.html.
[8] Mozilla Project, mozilla wiki Security/Server Side TLS, https://wiki.mozilla.org/Security/Server_Side_TLS.
[9] Mietzner R, Metzger A, Leymann F, Pohl K, Variability Modeling to Support Customization and Deployment of Multi-tenant-aware Software as a Service Applications, the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25, May 2009.
[10] Naylor D, Finamore A, Leontiadis I, Grunenberger Y, Mellia M, Munaf M, Steenkiste P, The cost of the S in HTTPS, the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT '14), pp. 133-140, ACM, December 2014.
[11] Nginx, Nginx, http://nginx.org/ja/.
[12] OpenSSL Software Foundation, OpenSSL, https://www.openssl.org/.
[13] OpenSSL Software Foundation, SSL_CTX_set_client_cert_cb,SSL_CTX_get_client_cert_cb - handle client certificate callback function, https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_set_client_cert_cb.html.
[14] Prodan R, 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, October 2009.
[15] Ryosuke M, ngx_mruby: Support ssl_handshake handler and dynamic certificate change, https://github.com/matsumotory/ngx_mruby/pull/145.
[16] Sanfilippo S, Noordhuis P, Redis, https://redis.io/.
[17] strace - linux syscall tracer, https://strace.io/.
[18] Takahiro Okumura, Dynamic certificate internals with ngx_mruby, https://speakerdeck.com/hfm/dynamic-certificate-internals-with-ngx-mruby-number-nagoyark03.
[19] The Apache Software Foundation, Apache HTTP Server, http://httpd.apache.org/.
[20] The Apache Software Foundation, ab - Apache HTTP server benchmarking tool, https://httpd.apache.org/docs/2.4/programs/ab.html.
[21] The Apache Software Foundation, Apache HTTP Server Version 2.4 Apache Module mod_ssl, http://httpd.apache.org/docs/current/mod/mod_ssl.html.
[22] The Apache Software Foundation, Apache Tutorial: Dynamic Content with CGI, http://httpd.apache.org/docs/2.2/en/howto/cgi.html.
[23] The Apache Software Foundation, Apache Virtual Host documentation, http://httpd.apache.org/docs/2.2/en/vhosts/.
[24] Will Glozer, wrk - a HTTP benchmarking tool, https://github.com/wg/wrk.
[25] 松本亮介, 岡部寿男, mod_mruby：スクリプト言語で高速かつ省メモリに拡張可能な Web サーバの機能拡張支援機構 , 情報処理学会論文誌, Vol.55, No.11, pp.2451-2460, 2014 年 11 月
[26] 松本亮介, 川原将司, 松岡輝夫, 大規模共有型 Web バーチャルホスティング基盤のセキュリティと運用技術の改善, 情報処理学会論文誌, Vol.54, No.3, pp.1077-1086, 2013 年 3 月.
[27] 松本 亮介, 三宅 悠介, 力武 健次, 栗林 健太郎, 高集積マルチテナント Web サーバの大規模証明書管理, 研究報告インターネットと運用技術（IOT）, Vol.2017-IOT-37(1), pp.1-8, 2017 年 5 月.