Vol.2017-CSEC-77 No.1
Vol.2017-IOT-37 No.1
2017/5/25

高集積マルチテナントWebサーバの大規模証明書管理

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

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，必要なメモリ使用量が増大することで，サーバプロセスの起動時や，CGIのようなプロセスの複製時にOSのページテーブルのエントリ数に依存するシステムコールの性能が著しく低下する．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，サーバ証明書データは，Redisによるキャッシュサーバによって管理し，本手法の有効性を評価した．

Large-scale Certificate Management on Highly-integrated Multi-tenant Web Servers

Ryosuke Matsumoto[1,a)]　Yusuke Miyake[1]　Kenji Rikitake[1,2]　Kentaro Kuribayashi[1,b)]

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 largely degrade the performance of system calls during the creation of the server process and process replication required for CGI invocation. 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 evaluate the feasibility of our proposal with a cache server of the server certificate data running on Redis.

[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
[a)] matsumotory@pepabo.com
[b)] antipop@pepabo.com

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[2]とGoogleによる常時SSL/TLS化の推進[5]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

© 2017 Information Processing Society of Japan

Vol.2017-CSEC-77 No.1
Vol.2017-IOT-37 No.1
2017/5/25

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 [11]．しかし，Let's Encrypt[6] 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 [10] 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[*1]で複数のホストを管理する必要がある [29]．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 [22] 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増加したりするからである．さらには，サーバプロセスの起動処理や，CGI[23] のようなプロセス複製の処理が大幅に遅くなり，性能への影響が大きくなるというデメリットもある．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)[3] 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，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[12] 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[7] の一種であるRedis[17] 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として公開しており[*2]，複数のドメインのHTTPS化が必要となるWebサービス事業者によって，幾つかの採用事例も報告されている [19], [25], [27]．

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，2章における性能面での課題をより明確化するために従来手法の性能劣化やメモリ使用量の問題についての検証を行った上で，Redisにサーバ証明書等のデータを保存しておきSSL/TLSハンドシェイク毎にサーバ証明書と秘密鍵をRedisから動的に取得する提案手法と，従来の事前にデータを読み込んでおく方式との性能比較を行って有効性を評価し，5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 [15] 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．

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．一方，高集積マルチテナント方式の場合，サーバに収容するホストが非常に多いため，事前に読み込んでおくサーバ証明書や秘密鍵の数もホスト数に応じて多くなり，Webサーバプロセス起動にで時間がかかったり，プロセスが使用するメモリ使用量が増大したりする．それにより，再起動に時間がかかるため，サービス停止の時間が長くなってしまう．また，プロセスのメモリ使用量増大に伴うOSのメモリ管理の影響により，特定の処理時に著しく性能が低下するという課題もある．

2.1 プロセスの再起動時間の課題

代表的なWebサーバソフトウェアであるApache httpd[20] やnginxでは，仮想ホスト機能によって，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．ただし，ここでいう単一のサーバプロセスとは，ホスト毎にサーバプロセスを起動させるのではなく，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，ホスト数とは無関係であるが，複数存在する．

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

[*1] Webサーバソフトウェアの実装により複数の処理プロセスが起動するが，ホスト数に依存しないことを示すために単一のサーバプロセスと呼ぶことにする
[*2] https://github.com/matsumotory/ngx_mruby

© 2017 Information Processing Society of Japan

要があるため，仮想ホスト機能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，ホスト数を万の単位で収容することも多く，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．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．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．

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 秒かかり，プロセスのメモリ使用量は証明書の設定だけで 3GB 程度になる．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．このメモリ使用量に関する詳細については，4 節で言及する．

2.2 メモリ使用量増大によるシステムコールの性能低下に関する課題

サーバ証明書を事前に読み込んでおく場合，Web サーバプロセスのメモリ使用量は証明書の数に従って単調増加する．ところで，一般的に，Linux においてプロセスのメモリ使用量が増加した場合に，プロセスを fork() システムコールにより複製しても，Copy on Write[1] の機能により，メモリ上のデータに変更がかからない限りメモリ領域のコピーは発生しない．そのため，高速にプロセスの複製を行うことができる．しかし，プロセスのメモリ使用量が数 GB 単位になった場合，プロセス毎に保持しているページテーブルと呼ばれる仮想メモリと物理メモリのアドレスの対応表のサイズが増大することにより性能低下を招く．ページテーブルサイズ T は，プロセスのメモリ使用量を P，ページテーブルエントリサイズを e，ページテーブルエントリ数を E，ページサイズを t[*3]とした場合，以下の通り求められる．

$$\frac{P}{t} = E \quad (1)$$

$$eE = T \quad (2)$$

この状況下では，P が大きくなればなるほど，(1) 式によるページテーブルエントリ数 E および (2) 式によるページテーブルサイズ T が増大する．また，fork() システムコールを実行すると，Copy on Write でメモリ領域がコピーされない場合でも，ページテーブルに保存されているページテーブルエントリは全てコピーされるため，fork() システムコールの実行に非常に時間がかかる．また，fork() システムコールとともに実行されることの多い execve() システムコールではページテーブルエントリの大部分を削除する処理が実行されるため，fork() システムコールによるページテーブルエントリのアドレスのコピーと同程度の処理が行われる．

Web サーバ上で利用される CGI 実行方式では，リクエスト毎に fork() システムコールと execve() システムコールが実行されるため，大量のページテーブルエントリのアドレスのコピーと削除が実行され，非常に遅くなる．プロセスのメモリ使用量増大による，fork() システムコールの CPU 使用時間の増加については，4 節でより詳細に述べる．

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がある．また，サーバプロセスのメモリ使用量が増大することによりシステムコールの実行が遅くなり，証明書を事前に読み込んでおくことによる性能面の利点が失われ，結果的に性能が劣化するという問題もある．

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 節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[3] 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．その場合，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

[*3] Linux 標準では 4096Bytes

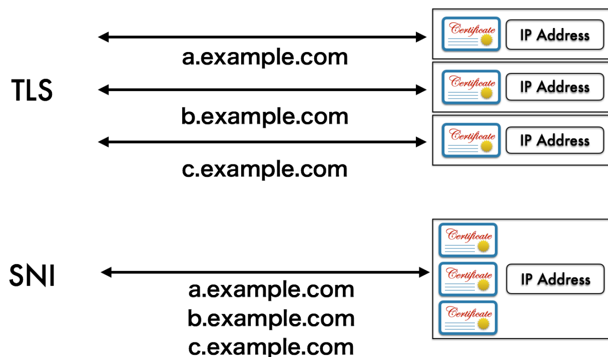

図 1 TLS の SNI 拡張の違い
Fig. 1 Characteristics of TLS SNI Extension.

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．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 [13] のバージョン 1.0.2 以降から，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()[14] が利用できるようになったため，その関数を利用して，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 [16]．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可能となる．

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 2** 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．

```
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 2 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. 2 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 3 動的なサーバ証明書読み込みの設定例 (KVS ベース)
Fig. 3 KVS-based Configuration Example of Dynamic Server Certificate Management.

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 3** 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．

実運用における設計としては，**図 4** 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

© 2017 Information Processing Society of Japan

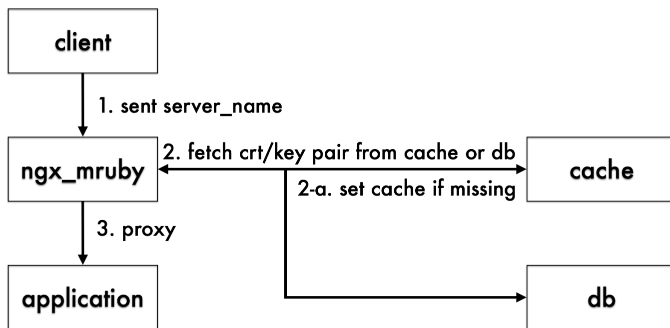

図 4 動的なサーバ証明書読み込みのシステム例
Fig. 4 System Example of Dynamic Server Certificate Management.

表 1 実験環境
Table 1 Experimental Environment.

	仕様
CPU	Intel Xeon E5-2620 v3 2.40GHz 24core
Memory	32GBytes
Server	NEC Express5800/R120f-2E
OS	CentOS6 Linux Kernel 4.8.14

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，キャッシュサーバに加えて，インメモリキャッシュも用いるといっ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，2.1 節と 2.2 節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，および，CGI で多用される fork() システムコールと execve() システムコールの実行に関する問題を，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．

表 2 従来手法のプロセスの起動時間とメモリ使用量の検証結果
Table 2 Result of Startup Time and Memory Usage of Process 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
プロセスのメモリ使用量（VSZ）	3207592 KBytes
プロセスのメモリ使用量（RSS）	3175912 KBytes

4.1 従来手法のメモリ使用量と起動時間の検証

2.1 節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とメモリ使用量を計測した．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，ユーザ CPU 使用時間を計測した．また，起動完了後の worker プロセスのメモリ使用量は， ps コマンドの仮想メモリ使用量 (VSZ) と物理メモリ使用量 (RSS) 項目を用いて計測した．

表 2 に結果を示す．表 1 の環境の CPU では，10 万ホスト分のサーバ証明書と秘密鍵を読み込むのに 42.662 秒かかった．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ないため，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．

メモリ使用量は，ps コマンドから得られるプロセスの仮想メモリ使用量（VSZ）と物理メモリ使用量（RSS）を計測した結果，両項目共に 3GB 程度となった．昨今の Web サーバが搭載している物理メモリは数十 GB から数百 GB も一般的になってきているため，プロセスが占有する 3GB のメモリ使用量では，サーバの物理メモリ量の不足という観点では問題にならない場合も多い．しかしながら，2.2 節で述べたとおり，プロセスが数 GB のメモリを使用することによる OS のプロセスとメモリ管理に起因する性能面の問題が生じ得る．

4.2 従来手法のメモリ使用量と性能の検証

2.2 節で述べた課題について，表 1 の環境において，高

© 2017 Information Processing Society of Japan

集積マルチテナント方式によって多数のホストを収容し，プロセスのメモリ使用量が増加した場合の性能への影響を検証するために，筆者の勤務先である GMO ペパボ株式会社にて Apache httpd バージョン 2.4.9 を利用して実際に稼動しているホスティングサービス環境において，収容ホスト数の違うパターンで動作しているサーバを選択し，メモリ使用量によって fork() システムコールの CPU 使用時間がどの程度変動するかを検証した．収容数は，2456 ホスト，13408 ホスト，22259 ホストのパターンを用意し，その時のプロセスのメモリ使用量と CGI を実行した時の fork() システムコールの処理時間を Linux の strace[18] によって計測した．fork() システムコールは，表 1 で採用している Linux カーネルバージョン 4 系では，clone() システムコールとして実行されているため，clone() システムコールの処理時間を計測した．

表 3 に結果を示す．ホスト数に概ね依存してメモリ使用量も増加しており，メモリ使用量に応じて clone() システムコールの CPU 使用時間が増加していることがわかる．2.2 節で述べたとおり，clone() システムコールの実行時においては，Copy on Write によりメモリコピーはほとんど生じないが，ページテーブルエントリのコピーは必要である．そのため，プロセスのメモリ使用量が増加したことによってページテーブルエントリ数が増加し，ページテーブルエントリのコピーによってシステム CPU 使用時間が増加している．

プロセスのメモリ使用量が 1.6GB 程度になると，clone() システムコールは表 1 の環境で 0.05 秒の CPU を使用することになる．clone() システムコールの CPU 使用時間がシステム CPU であることを考えると，1 コアで 1 秒間に最大 20 回程度しか clone() システムコールを実行できないことになり，clone() システムコールや，ページテーブルエントリを削除する処理が実行される execve() システムコールの実行時間，実際のユーザ CPU による CGI のコンテンツの処理も考慮すると，clone() システムコールと execve() システムコールが実行される CGI のような処理では，1 コアで少なくとも 1 秒間に数回程度しか実行することができない．

Web サーバの処理において，C10K 問題 [8] のように同時並行的に処理することが当たり前とされる時代において，1 コアで 1 秒間に数回程度のリクエスト処理しかできないという状態は非常に問題である．また，サーバ証明書を読み込むことでさらにメモリ使用量が増大するため，CGI のような処理を効率的に処理できないことは明らかである．

上記のホスティングサービス環境で，サーバの高負荷状態に陥った状況で，CPU の使用率を調査すると 80%以上システム CPU が使用しており，その状態でのカーネルの処理を perf コマンドにより調査したところ，システム CPU 使用率の 50%以上は，ページテーブルエントリの操作に利用されていた．**図 5** は，調査時のカーネル内の処理である．copy_pte_range は，clone() システムコールの内部処理で，ページテーブルエントリのアドレスをコピーする処理であり，zap_pte_range は execve() システムコールの内部処理で，ページテーブルエントリのアドレスを削除する処理である．図 5 の perf のログからも，ページテーブルエントリのアドレスの処理の負荷は，プロセスのメモリ使用量に強く依存することがわかる．

```
22.38% [kernel] [k] copy_pte_range
18.44% [kernel] [k] zap_pte_range
11.13% [kernel] [k] change_pte_range
 3.58% [kernel] [k] page_fault
 3.32% [kernel] [k] page_remove_rmap
```

図 5 高負荷時のカーネルのデバッグログ
Fig. 5 Kernel Debug Log Under Heavy Load.

4.3 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 6** に ngx_mruby の設定を示す．ポート 58080 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 58081 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[21] 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

表 3 従来手法のプロセスのメモリ使用量と性能の検証結果
Table 3 Result of Memory Usage and Performance of Process by Exisiging method.

ホスト収容数	プロセスのメモリ使用量（RSS）	clone() システムコールの実行時間
2456	256624 KBytes	0.010114 sec
13408	611368 KBytes	0.028492 sec
22259	1624012 KBytes	0.053687 sec

```
# 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 6 動的読み込みと事前読み込みの設定
Fig. 6 Configuration of Dynamic Loadding and Preloading.

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[9] 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．

表 4 実験結果
Table 4 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 4 に結果を示す．実験結果では，事前にサーバ証明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，ホスト数に依存してメモリ使用量が大きくなり，4.2 節で述べた問題がある．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メモリに保存しておく必要がないためメモリ使用量は非常に少なく，かつ，性能面でも実用的である．証明書の保存方法についても，Redis のようなキャッシュサーバを TCP で接続できるようにしておくことで，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 [4] によるサーバ増強が可能となって，運用上のメリットも大きい．

予備実験として，Redis からサーバ証明書データを取得する方式以外に，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. 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

© 2017 Information Processing Society of Japan

ロセス起動に時間がかかる問題や Web サーバプロセスがメモリを大量に使用することにより性能が劣化する問題があった．

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．さらに，サーバ証明書データを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．提案手法の応用例として，Let's Encrypt と連携して，Web サーバに対して A レコードさえ向いていれば，最初の HTTPS アクセス時に自動的に DV 証明書を取得して設定し，HTTPS 通信を行うようなシステム設計も可能である [30]．

参考文献

[1] Accetta M, Baron R, Bolosky W, Golub D, Rashid R, Tevanian A, Young M, Mach: A new kernel foundation for unix development. In USENIX Conference, pages 93-112, 1986.
[2] Belshe M, Thomson M, Peon R, Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.
[3] Eastlake D, Transport Layer Security (TLS) Extensions: Extension Definitions, RFC 6066, 2011.
[4] 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.
[5] Ilya Grigorik, Pierre Far, Google I/O 2014 - HTTPS Everywhere, https://www.youtube.com/watch?v=cBhZ6S0PFCY.
[6] Internet Security Research Group (ISRG), Let's Encrypt - Free SSL/TLS Certificates, https://letsencrypt.org/.
[7] 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.
[8] Kegel D, The C10K problem, http://www.kegel.com/c10k.html.
[9] Mozilla Project, mozilla wiki Security/Server Side TLS, https://wiki.mozilla.org/Security/Server_Side_TLS.
[10] 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.
[11] 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.
[12] Nginx, Nginx, http://nginx.org/ja/.
[13] OpenSSL Software Foundation, OpenSSL, https://www.openssl.org/.
[14] 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.
[15] 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.
[16] Ryosuke M, ngx_mruby: Support ssl_handshake handler and dynamic certificate change, https://github.com/matsumotory/ngx_mruby/pull/145.
[17] Sanfilippo S, Noordhuis P, Redis, https://redis.io/.
[18] strace - linux syscall tracer, https://strace.io/.
[19] Takahiro Okumura, Dynamic certificate internals with ngx_mruby, https://speakerdeck.com/hfm/dynamic-certificate-internals-with-ngx-mruby-number-nagoyark03.
[20] The Apache Software Foundation, Apache HTTP Server, http://httpd.apache.org/.
[21] The Apache Software Foundation, ab - Apache HTTP server benchmarking tool, https://httpd.apache.org/docs/2.4/programs/ab.html.
[22] The Apache Software Foundation, Apache HTTP Server Version 2.4 Apache Module mod_ssl, http://httpd.apache.org/docs/current/mod/mod_ssl.html.
[23] The Apache Software Foundation, Apache Tutorial: Dynamic Content with CGI, http://httpd.apache.org/docs/2.2/en/howto/cgi.html.
[24] Will Glozer, wrk - a HTTP benchmarking tool, https://github.com/wg/wrk.
[25] アスタミューゼ株式会社, nginx + ngx_mruby で SSL 証明書の動的読み込みを実現して、作業がとても楽になった話, http://lab.astamuse.co.jp/entry/2016/11/30/114500.
[26] NPO 法人軽量 Ruby フォーラム, http://forum.mruby.org/.
[27] BASE 株式会社, BASE 開発チームブログ 独自ドメインのショップで https でアクセスできるようになりました, http://devblog.thebase.in/entry/2017/03/22/111015.
[28] 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
[29] 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.
[30] 松本亮介, ngx_mruby で最初の HTTPS アクセス時に自動で証明書を設定可能にする FastCertificate の提案と PoC, http://hb.matsumoto-r.jp/entry/2017/03/23/173236.