sshr: ユーザに変更を要求せずにシステム変化に追従可能なSSHプロキシサーバ

鶴田 博文[†1,a)]　松本 亮介[†1,b)]

概要：Web サービスを支えるインフラは，ユーザからの多様な要求に応えるために，ユーザにシステムの構成情報やその変更を意識させることなく，迅速かつ柔軟にシステム構成を変更することが求められる．一方，サーバへのリモート接続サービスとして利用されている SSH では，ユーザが利用するサーバの IP アドレスまたはホスト名を指定して接続要求を送るため，サーバの IP アドレスまたはホスト名に変更があった場合，ユーザは変更後の情報を知る必要がある．この問題を解決するために，gcloud コマンドのようなクライアントツールがサーバごとの一意のラベル情報をもとに接続先の IP アドレス等を取得する手法があるが，この手法ではユーザに用いるツールの制限や変更を要求する．別の手法として，SSH Piper のようなプロキシサーバがユーザ名をもとに接続先の IP アドレス等取得する手法があるが，既存のプロキシサーバではその動作を変更するためにはソースコードを直接変更しなければならない．本論文では，ユーザに用いるクライアントツールの制限や変更を要求せず，システム管理者が組み込み可能なフック関数を用いてシステム変化に追従できる SSH プロキシサーバを提案する．提案手法は，組み込むフック関数のみの修正でプロキシサーバの動作を自由に変えられるため，システムの仕様変更に対して高い拡張性を有している．さらに実験から，提案手法を導入した場合の SSH セッション確立のオーバーヘッドは 20 ミリ秒程度であり，ユーザがサーバに SSH ログインする際に遅延を感じないほど短い時間であることを確認した．

sshr: An SSH Proxy Server for Responding to System Changes without Forcing Users to Change

Hiforumi Tsuruta[†1,a)]　Ryosuke Matsumoto[†1,b)]

Abstract: The infrastructure of Web services is required to change the system configuration quickly and flexibly without making the user aware of the system configuration in order to respond to various requests from users. On the other hand, since SSH used as a secure remote connection service to a server needs to send a connection request by specifying the IP address or host name of the server, the user needs to know the changed information when the IP address or host name is changed. In order to solve this problem, there is a method in which a client tool such as gcloud command obtains the IP address or host name of the destination based on the unique label information of each server. However, this method requires restrictions and changes to the tools used by client-side. Another method is to use a proxy server, such as SSH Piper, to obtains the IP address or host name of the connection destination based on the user name. However, in the existing SSH proxy server, the source code must be changed directly to change the behavior of a proxy server. In this paper, we propose an SSH proxy server that can respond to system changes using hook functions that can be incorporated by system administrators without requiring restrictions or changes to the clients. The proposed method has high extensibility for system changes because the behavior of proxy server can be easily changed by only modifying the hook function to be incorporated. Furthermore, when using the proposed method, it was confirmed that the overhead of establishing an SSH session is about 20 milliseconds which is a short time that the user does not feel a delay when logging into the server with SSH.

1. はじめに

Web サービスのようなインターネットを介して利用するサービスが世の中に普及し，その利用者数が増大している．その結果，システム管理者は，サービス利用者からの多様な要求に応えるために，管理するシステムの構成を状況に応じて迅速かつ柔軟に変化させることが求められている [1]．例えば，Web サービスの利用状況に基づき，サーバをスケールアウト・スケールイン [2] することが要求される．Web サービスへの突発的なアクセス集中に対してサーバをスケールアウトさせ，アクセスが減少するとスケールインすることで，サービスの機会損失の低減や運用コストの削減を行うことができる．また，サービスの機能拡張や規模拡大に伴い，新規サーバを導入 [3] していくことも要求される．市場変化をいち早く察知し，サービスそのものを変化させていくことは，サービスの競争力を維持・向上させることにつながる．こうしたサービスを取り巻く環境の変化やサービスそのものの規模の変化に対して，迅速かつ柔軟にシステム構成を変化させながら対応していくことは，サービス成長を加速させる上での重要な要素となる．

システム管理者が迅速に変化に対応するためには，システム内のサーバ等のリソースを使用するユーザに対して，システム変更が起きるたびに変更後の情報を提供することなく，自由にシステム構成を変更させられることが必要である．一方，サーバへの安全なリモート接続サービスとして広く利用されている SSH（Secure Shell）[4] は，ユーザが利用するサーバの IP アドレスまたはホスト名を指定して接続要求を送る必要があるため，ユーザは自分が利用したいサーバの IP アドレスまたはホスト名を把握しておかなければならない．そのため，もしサーバの IP アドレスやホスト名に変更があった場合，管理者は各ユーザへ変更後の情報を知らせる必要があり，ユーザはその変更に追従しなければならない．

現在，ユーザが接続するサーバの IP アドレス・ホスト名やその変更を意識することなく SSH 接続ができる手法がいくつか存在する．一つは，クライアントツールがサーバごとの一意のラベル情報をもとに接続先の IP アドレスまたはホスト名を取得する手法であり，例として，GCP（Google Cloud Platform）の gcloud コマンド [5] が挙げられる．この手法を用いると，ユーザはシステムの構成情報やその変更を意識することなく，サーバに紐づいたラベル情報のみを用いて透過的に目的のサーバに SSH 接続ができる．しかしながら，この手法では，ユーザに用いるクライアントツールを強制する，かつシステム側の仕様変更に伴いクライアントツールの動作にも変更が生じた場合に，全ユーザに変更を要求することになる．

他の手法として，クライアントとサーバの間に導入したプロキシサーバがリクエストの情報をもとに接続先の IP アドレスまたはホスト名を取得する手法がある．これを実現できる既存の SSH プロキシサーバとして，SSH Piper[6] というオープンソース実装が存在する．ユーザは，SSH Piper を介して SSH 接続することで，用いるクライアントツールの制限や変更を要求されない，かつシステムの構成情報やその変更を意識することなく，ユーザ名を用いて透過的に目的のサーバに SSH 接続ができる．しかしながら，SSH Piper は接続先サーバを決定するロジックをシステム管理者が自由に組み込むことや変更することができず，ロジックを変更するためにはソースコードに直接変更を加えなければならないため，システムの仕様変更に対する拡張性が低い．

本論文では，ユーザに用いるクライアントツールの制限や変更を要求せず，システム管理者が自由に組み込み可能なフック関数を用いてシステム変化に追従できる SSH プロキシサーバを提案する．提案する SSH プロキシサーバは sshr[7] と名付けた．sshr は，ユーザ名から接続先サーバを決定するためのフック関数をシステム管理者が自由に実装・導入可能である．これにより，ユーザ名を用いて透過的に目的のサーバに SSH 接続できるため，接続先サーバの IP アドレスやホスト名に変更があった場合でも，管理者は各ユーザへ変更後の情報を知らせる必要がなく，ユーザも変更に追従する必要がない．さらに，sshr は SSH のセッション確立に必要なユーザ認証の処理を拡張するための関数も備えている．例えば，sshr がユーザから公開鍵認証のリクエストを受けた際に，ユーザの公開鍵を検索する処理をフック関数として組み込み可能である．これにより，システム管理者がユーザの公開鍵をデータベースなどの自由なデータ形式で管理することができる．このように，sshr では SSH 特有の認証に対しても，プログラマブルに制御するための仕組みをとっている．

本稿の構成を述べる．2 章では，既存手法とその課題について言及する．3 章では，本論文での提案を述べる．4 章では，提案手法を用いたユースケースについて具体例を挙げて説明する．5 章では，提案手法の性能評価の結果を述べる．最後に，6 章でまとめとする．

2. 既存手法とその課題

サービスへの多様な要求に応じてシステム構成を迅速に変更していくことが求められる状況においては，システムの運用管理も変更に追従できる必要がある．一方，サーバへのリモート接続サービスとして用いられる SSH では，通常サーバの IP アドレスやホスト名に変更があった場合，管理者は各ユーザへ変更後の情報を知らせる必要があり，

†1 さくらインターネット株式会社 さくらインターネット研究所
SAKURA internet Research Center, SAKURA internet Inc., Akasaka, Chuo-ku, Fukuoka 810-0042 Japan
a) hi-tsuruta@sakura.ad.jp
b) r-matsumoto@sakura.ad.jp

ユーザはその変更に追従しなければならない．本章では，この SSH の課題を解決する既存手法を整理し，それらの特徴と課題について述べる．

2.1 クライアントツールによる接続先の決定

gcloud は GCP 上のリソースを操作する主要なコマンドラインツールである．「gcloud compute ssh」コマンド [8] にインスタンス名を指定することで，対象のインスタンスに SSH 接続が可能である．「gcloud compute ssh」は ssh(1) コマンド [9] のラッパーであり，SSH のリクエストを送る前にインスタンス名をキーに GCP が管理している構成情報から IP アドレスを取得する機能を有している．これにより，ユーザは自身で定義したサーバごとに一意のインスタンス名を知っていれば，対象サーバの IP アドレスの情報やその変更を意識することなく透過的に SSH 接続が可能である．

また，他の手法として，HashiCorp が開発するクラスタ管理ツールである Consul[10] を用いた手法が挙げられる．Consul はクラスタ内のメンバー管理機能と，そのメンバー情報を参照するための HTTP API を有している．これらを活用すると，ユーザが定義したサービス名やタグ情報をもとに特定サーバの IP アドレスを取得することができるため，この取得した情報をもとに SSH リクエストを送るようなクライアントツール [11] を作成可能である．

いずれの手法においても，ユーザはサーバごとにラベル情報を持たせられるため，同一ユーザで複数のサーバに対して，ラベル情報を用いた透過的な SSH 接続が可能である．もし，目的のサーバの IP アドレス等に変更があった場合でも，ユーザはその変更を意識することなく，クライアントツールが変更に追従する役割を担うことができる．一方，これらの手法は，ユーザ側に独自の SSH クライアントツールの使用を強制する．そのため，システム側の仕様変更などに伴いクライアントツールの動作にも変更が生じた場合に，全ユーザが使用しているツールに対してバージョンアップ等の変更を課す．したがって，ユーザがシステム側の仕様変更を意識する必要があることや，その変更に対して追従しなければならない点が課題として挙げられる．

2.2 プロキシサーバによる接続先の決定

SSH Piper[6] は，GitHub 上でオープンソースソフトウェアとして開発されている SSH のプロキシサーバである．SSH Piper の特徴は，SSH のリクエストを受けた際に，そのユーザ名から利用する接続先サーバを決定できる点である．SSH 接続に SSH Piper を介することで，ユーザはプロキシサーバの IP アドレスまたはホスト名さえ知っていれば，接続するサーバの IP アドレス・ホスト名やその変更を意識することなく，ユーザ名に紐づいたサーバに SSH 接続ができる．SSH Piper では接続先の決定の役割をユーザ側ではなく，プロキシサーバ側に持たせたことで，ユーザが用いるツールの制限や変更を課す必要がない．しかし，一方で，ユーザ名から接続先サーバを決定するロジックをシステム管理者が自由に開発・導入することができない．これにより，接続先の決定に用いるユーザとサーバの紐付け情報を管理するデータ形式を自由に設計できない．例えば，ユーザとサーバの紐付け情報をデータベースで管理したい場合，SSH Piper 側が指定したスキーマのテーブルを用意する必要がある．もし，このような接続先決定のロジックを既存の仕組みから拡張したい場合には，ソースコードに直接変更を加えなければならない．

また，SSH Piper のような接続先の決定が可能な SSH プロキシサーバを構築する手法の一つとして，sshd[12] を拡張する手法が挙げられる．sshd は，現在最も広く利用されている SSH サーバのデーモンプログラムである．プロキシサーバを一から実装する場合と比べて，sshd を拡張することで，認証の仕組みを独自に実装する必要がなく，sshd の信頼性の高い認証の仕組みを再利用できる点がメリットである．しかしながら，sshd はモジュールによる機能拡張の仕組みをサポートしていないため，その機能を拡張するためには，ソースコードに直接変更を加えなければならない．

いずれの手法においても，ユーザが用いるツールの制限や変更を課す必要がなく，ユーザ名を用いて目的のサーバに透過的な SSH 接続が可能である．もし，目的のサーバの IP アドレス等に変更があった場合でも，ユーザはその変更を意識することなく，プロキシサーバが変更に追従する役割を担うことができる．一方，接続先決定のためのラベル情報として用いられるものがユーザ名に限られるため，同一ユーザは 1 サーバにしか紐づかない点は，プロキシサーバを用いた場合の制約となる．また，既存のプロキシサーバを構築する手法は，機能拡張の仕組みをサポートしていないため，仕様変更に対する拡張性が低いことが課題として挙げられる．

3. 提案手法

2.1 節で述べた通り，システムの構成変更に追従する役割をクライアントツール側が担う場合は，ユーザに用いるクライアントツールの制限や変更を要求することが課題である．一方，2.2 節で述べた通り，変更に追従する役割をプロキシサーバが担う場合は，ユーザに制限や変更を要求しないが，仕様変更に対するプロキシサーバの機能の拡張性が低いことが課題である．これらの課題を解決するために，ユーザに用いるクライアントツールの制限や変更を要求せず，システム管理者が自由に組み込み可能なフック関数を用いてシステム変化に追従できる SSH プロキシサーバを提案する．提案手法では，システム側に仕様変更があった場合でも，ユーザにその変更を意識させることなく，プ

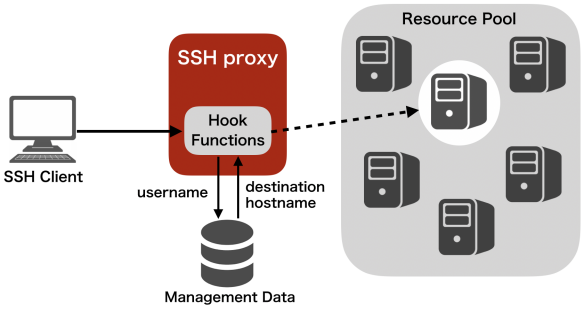

図 1: 提案手法の概要図

ロキシサーバに組み込むフック関数のみの修正でプロキシサーバの動作を拡張し，変更に対応可能である．

3.1 アーキテクチャの概要

提案手法では，SSH のプロキシサーバに対して，システム管理者が実行したい処理を自由に実装し，フック関数として組み込めるアーキテクチャをとった．図 1 は，提案手法の概要図である．SSH のプロキシサーバがクライアントからの接続要求を受け取り，接続するサーバを決定することで，ユーザはプロキシサーバの IP アドレスまたはホスト名さえ知っていれば，プロキシサーバより後方のシステムの構成やその変更を意識する必要がなくなる．システム管理者の視点では，ユーザに事前に提供するプロキシサーバの IP アドレスまたはホスト名の情報にさえ変更が起きなければ，ユーザに知らせることなく自由にシステムの構成変更を行うことが可能となる．そこで，プロキシサーバの前段に構築したロードバランサに仮想 IP アドレスを割り当て，ユーザからの SSH 接続を全て同じ IP アドレスで受け付けるようにすることが有効である．これにより，プロキシサーバの IP アドレスに変更が起きる場合や，プロキシサーバをスケールアウトしたい場合などに対してもユーザ側に変更を要求する必要がなくなる．

SSH でサーバに接続する場合，ユーザが目的のサーバを利用する権限をもっているかどうかを確認するために，ユーザ認証を行う必要がある [13]．提案手法では，認証方式としてパスワード認証および公開鍵認証をサポートしている．プロキシサーバを介してユーザ認証を行う場合にプロキシサーバがクライアント・サーバ間の認証をどのように仲介するかは用いる認証方式で異なる．パスワード認証の場合，プロキシサーバはクライアントから受けた認証リクエストを解釈することなく，SSH サーバにそのままリクエストパケットを送り，SSH サーバ側で認証を行うことができる．一方，公開鍵認証を用いた場合，プロキシサーバは認証を SSH サーバに任せることはできない．なぜなら，SSH セッションはセッション ID[13] と呼ばれるセッションごとに一意の識別子を有しており，公開鍵認証では，このセッション ID を含めたデータを秘密鍵で暗号し，認証リクエストを送るためである．プロキシサーバは，クライアント間とサーバ間の 2 つの SSH セッションを保持しており，当然ながらそれぞれ異なるセッション ID を有している．このため，プロキシサーバを介した公開鍵認証を行いたい場合，クライアント・プロキシサーバ間およびプロキシサーバ・SSH サーバ間の 2 段階で認証を行う必要がある．

```
import (
  "github.com/tsurubee/sshr/sshr"
  "log"
)

func main() {
   confFile := "/etc/sshr/example.toml"
    s, _ := sshr.NewSSHServer(confFile)
    // 独自に実装した接続先サーバ決定のための関数を渡す
   s.ProxyConfig.FindUpstreamHook = findUpstreamHook
   if err := s.Run(); err != nil {
      log.Fatal(err)
   }
}
```

図 2: 接続先の決定処理を組み込むためのサンプルコード

提案手法では，SSH プロキシサーバに対して，特定の役割をもつ複数のフック関数を組み込み可能であり，それらの関数が SSH リクエストの処理の過程で順次実行される仕組みをとっている．組み込み可能なフック関数として，SSH のリクエストを受けた際にユーザ名をもとに接続先サーバの IP アドレスまたはホスト名を取得する関数がある．その他に，決定した接続先サーバとクライアント間の公開鍵認証を拡張するためのフック関数も備えている．具体的には，クライアント・プロキシサーバ間の認証において，クライアントの公開鍵を検索する関数，およびプロキシサーバ・SSH サーバ間の認証に用いる秘密鍵を検索する関数がある．これらを用いると，SSH プロキシサーバにおける接続先の決定および公開鍵認証をプログラマブルに拡張でき，システム管理者はユーザとサーバの紐付け情報や，認証に用いる鍵をデータベースなど自由なデータ形式で管理することが可能となる．

3.2 アーキテクチャの実装

提案するアーキテクチャを実現するために，Go 言語 [14] を用いて sshr[7] という SSH プロキシサーバを実装する．sshr を実装するための要件を整理すると，まず sshr は複数のクライアントから同時に受けたリクエストを並行で処理できる必要がある．また，sshr が SSH のリクエストを処理するためには，SSH 特有の認証の仕組みをサポートしなければならない．Go 言語は，並行処理の記述が容易であるという特徴を有するため，1 つ目の要件を満たす．さらに，Go 言語は，豊富な機能をもつ SSH パッケージ [15] が存在するため，SSH プロキシサーバを実装する上で必要な認証等の多くの機能の実装に，SSH パッケージに備わって

いる機能を利用できる．これらの実装面の要件に加えて，運用面の要件も存在する．sshr はシステムの多様な変更に合わせて動作を変更することが想定されるため，その変更から本番環境への適用が容易に行える必要がある．Go 言語ではソースコードをビルドした結果の実行ファイルとして，シングルバイナリが作成される．これにより，サーバへのデプロイが容易であることや，デプロイ先のサーバに特別な実行環境を必要としないなどのメリットがある．そのため，例えばコード管理を Git で行い，リモートリポジトリへのプッシュを起点に，ビルド，デプロイ，実行までの一連の処理を CI ツールで容易に自動化できる．このようにプログラムの修正から実行までを開発プロセスに組み込むことで，保守性や開発効率を向上させることが可能である．これらの実装面および運用面の理由から sshr の実装に Go 言語を採用した．

sshr は，システム管理者が Go 言語で実装した関数を組み込んでビルドすることで，プロキシサーバの機能を拡張できる仕組みをとっている．図 2 は，接続先の決定処理を組み込むためのサンプルコードである．図中の sshr パッケージの FindUpstreamHook に対してシステム管理者側で独自に実装した関数を渡すことでフック関数として組み込むことができる．このような仕組みを採用することで，システム管理者が実装したフック関数も含めてシングルバイナリが作成できるため，前述のようなデプロイの容易性や開発の効率化などのメリットを享受できる．sshr では，以下の三つのフック関数をサポートしている．

(1) FindUpstreamHook
(2) FetchAuthorizedKeysHook
(3) FetchPrivateKeyHook

（1）は，ユーザ名をもとに接続先サーバの IP アドレスまたはホスト名を取得するための関数である．(2)・(3) は公開鍵認証を用いる場合に必要なフック関数である．(2) は，ユーザの公開鍵を検索するための関数である．(3) は，sshr から接続先サーバに対して認証を行うために利用する秘密鍵を検索するための関数である．(2)・(3) の用途については後述する．

SSH のプロキシサーバを実装する上での重要な課題として，プロキシサーバを介したクライアント・サーバ間の認証をいかにして行うかという点が挙げられる．3.1 節で述べた通り，公開鍵認証を用いた場合，クライアント・sshr サーバ間および sshr サーバ・SSH サーバ間の 2 段階で認証を行う必要がある．まず，クライアント・sshr サーバ間において，sshr サーバはクライアントの公開鍵認証を行うために，クライアントの公開鍵を検索するためのフック関数である FetchAuthorizedKeysHook を備えている．これは，sshd の AuthorizedKeysCommand[16] に相当する機能である．これを用いることで，システム管理者はユーザの公開鍵をデータベースなどの自由なデータ形式で一元的に管理することが可能となる．次に，sshr サーバ・SSH サーバ間において，sshr サーバはクライアントの秘密鍵を有していないため，別の秘密鍵を用いる必要がある．クライアントが目的のサーバに SSH 接続をする妥当性については，一段階目のクライアント・sshr サーバ間で認証を行っているため，sshr サーバ・SSH サーバ間については，sshr サーバが有している特定の秘密鍵を SSH サーバで全て許可するようにすることで，2 段階目の認証を行っている．この際に，sshr サーバが SSH サーバへの認証に用いる秘密鍵を検索するためのフック関数として FetchPrivateKeyHook を備えている．このような認証方式を用いることによるセキュリティリスクについては次節で言及する．

プロキシサーバを介した認証に成功し，SSH セッションを確立した後の sshr の動作について説明する．sshr で実行できるフック関数は，全て SSH セッション確立前に実行されるものである．セッション確立後は，sshr はクライアントおよびサーバから受け取るパケットを解釈することはなく，パケットをそのまま転送するのみである．このため，sshr を用いた場合，セッション確立時にフック関数の実行によるオーバーヘッドが発生する．この点ついては，5 章で評価する．

3.3 セキュリティリスクとその対策

sshr を用いた場合における第三者から不正アクセスを受けるリスクについて議論する．認証方式としてパスワード認証を用いた場合，sshr サーバはパスワードを読み取ることなく，SSH サーバにパケットを転送する．すなわち，ユーザのパスワードを知っていて，認証を行うことができるのは SSH サーバのみであるため，sshr サーバを用いた場合でも認証を突破されるリスクは変わらない．

公開鍵認証を用いた場合，クライアント・プロキシサーバ間およびプロキシサーバ・SSH サーバ間のセッションごとに 2 段階で認証を行う必要がある．sshr の認証の仕組みは，3.2 章で述べた通り，sshr サーバから SSH サーバへの公開鍵認証に特定の秘密鍵を全て受け入れるような設定を SSH サーバに行っている．このため，sshr サーバのセキュリティの問題や sshr デーモンの脆弱性などが原因で，sshr サーバが有している秘密鍵が流出した場合や，sshr サーバから秘密鍵を利用した任意のコマンドが実行される場合，SSH サーバへの全ての SSH 接続を許してしまうこととなる．

これらのリスクを最小化するための対策を sshr サーバ側と SSH サーバ側のそれぞれについて述べる．まず，sshr サーバでは，保有している秘密鍵の流出を防ぐためにサーバのセキュリティを堅牢にする必要がある．例えば，sshr サーバではログイン可能なユーザ数や権限を最小限に留めることや，アクセスが可能な IP を組織内のネットワークのみ絞ることが有効である．次に，sshr を経由して任意の

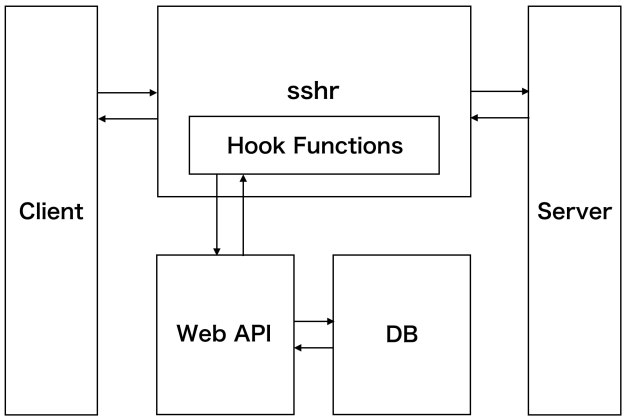

図 3: システム構成の例

username	hostname	publickey
userA	server01.sshr.sample.jp	ssh-rsa xxxxxxx
userB	server02.sshr.sample.jp	ssh-rsa yyyyyyy

図 4: テーブルスキーマの例

コマンドが実行される場合に備えて，sshr デーモンは root 権限を持たないユーザで実行しておく．この対策は，任意のコマンドが実行された場合に sshr サーバに対する操作を制限することには有効であるが，コマンドが sshr デーモンと同じ権限で実行されているため，SSH サーバに対する操作は可能となる．

次に，SSH サーバにおけるセキュリティ対策について述べる．まず，SSH サーバでは root でリモートログインを許可しない設定をすることが必要である．例えば，sshd では sshd_config の PermitRootLogin[17] を使って root でのリモートログインを制限することが可能である．これにより，SSH サーバに対する操作の権限を制限することができる．また，sshr サーバが有する秘密鍵が外部に流出した場合に備えて，接続元 IP の制限を行うことが有効である．例えば，sshd の authorized_keys ファイルでは公開鍵ごとにオプションを記述することができる．このオプションの中の接続元 IP の制限機能 [16] を用いて，sshr サーバの IP アドレスからの接続のみを許可することで，秘密鍵が外部に流出した際の被害を防ぐことができる．

4. 提案手法の適用例

図 3 は提案手法を採用したシステム構成の例である．それぞれのロールについて説明する．Client ロールは，用いるツールの制限がないため，OpenSSH に備わっている ssh(1) コマンド [9] や，Tera Term[18] など GUI ベースの SSH クライアントが想定される．Server ロールについても用いるツールの制限がないため，代表的な SSH サーバである sshd[12] などが想定される．DB ロールは，sshr を利用するユーザやサーバが増えてきた場合，それらの管理を行うデータベースサーバである．図 4 は，テーブルスキーマの例である．提案手法は，ユーザ名に紐づいたサーバに対して SSH 接続できる仕組みであるため，データベースではユーザ名とサーバのホスト名の紐付けを管理する．また，sshr にてユーザの公開鍵認証を行うために，ユーザ名およびホスト名に併せて公開鍵をデータベースで管理することが有効である．このテーブルの情報を sshr から直接参照することも可能であるが，Web API を介してデータベースに格納されている情報を参照することも可能である．このようにデータの取得方法を自由に選択できることは，sshr が実行するフック関数をプログラマブルに拡張できる仕組みを採用していることのメリットの一つである．

```
type Response struct {
  Username  string ‘json:"username"‘
  Hostname  string ‘json:"hostname"‘
  PublicKey string ‘json:"publickey"‘
}

func findUpstreamHook(username string) (string, error) {
  requestURL := "http://sshr.jp/upstream/username"
  resp, _ := http.Get(requestURL)
  defer resp.Body.Close()
  b, _ := ioutil.ReadAll(resp.Body)
  var decoded = new(Response)
  if err := json.Unmarshal(b, &decoded); err != nil {
    return "", err
  }
  return decodedBody.Hostname, nil
}
```

図 5: Web API 経由での接続先取得のサンプルコード

最後に，sshr ロールでは実行可能なフック関数を 3 つ備えているため，それぞれについて想定される処理を解説する．まず，接続先の決定に用いる FindUpstreamHook は，HTTP クライアントとして Web API から情報を取得する処理を行う．図 5 は，Web API 経由での接続先取得のためのサンプルコードである．Web API から取得した JSON 形式のデータを構造体にデコードし，ホスト名を取り出している．次に，ユーザの公開鍵検索に用いる FetchAuthorizedKeysHook は，前述の通りユーザの公開鍵をデータベースで管理するため，FindUpstreamHook と同様に HTTP クライアントとして Web API から情報を取得する処理を行う．最後に sshr から接続先サーバに対して認証を行うために利用する秘密鍵を検索する FetchPrivateKeyHook は，秘密鍵をデータベースで管理してネットワーク経由で取得することは，流出のリスクを高めるため，sshr サーバのローカルストレージに保管されている秘密鍵を参照する処理を行うことなどが想定される．

このようなシステムを構築した場合，システムの変更に

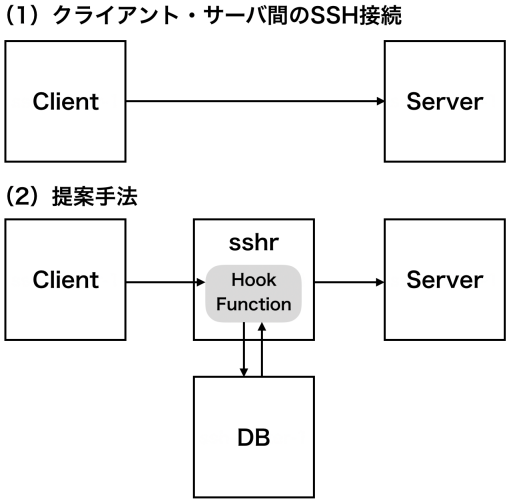

図 6: 実験環境の概要図

対してどのような方法で追従可能であるかを例を挙げて説明する．想定されるシステムの変更として，以下の 3 つを例に挙げる．

(1) ユーザが用いるサーバを変更したい
(2) DB ロールのテーブル構成を変更したい
(3) Web API の仕様を変更したい

（1）の場合，データベースの情報を更新するだけで良い．sshr はリクエストのたびにユーザとそのユーザが用いるサーバの紐付け情報をデータベースから参照するため，データベースの情報を更新するだけで，ユーザの接続先を切り替えることが可能である．（2）の場合，Web API の処理を変更するだけで良い．このようにビジネスロジックを Web API 側に持たせることで，sshr の変更を行う必要がなくなる．（3）の場合，sshr のフック関数に変更を加える必要がある．システム管理者が Go 言語で実装したフック関数を Web API の仕様変更に合わせて修正し，再度ビルドすることで得られたシングルバイナリをデプロイする．デプロイは CI ツールで行うなど，開発プロセスに組み込むことで開発者の手を介さず自動で行うことが有効である．また，Server::Starter[19] を用いることで，実行中のプロキシサーバを停止させることなく，デプロイした変更後のバイナリを適用することができる．

プログラマブルに組み込み可能なフック関数を用いて柔軟にプロキシサーバの動作を制御可能にしたことで，システム管理者がユーザやサーバの情報を自由なデータ形式で管理できることや，そのデータ取得方法を自由に選択できる．また，提案手法は想定される様々なシステムの変更に対して，プロキシサーバ自体に変更を要しない，もしくは関数の修正のみで無停止で動作を切り替えられるためシステムの変更に対して高い拡張性を有する．

表 1: 実験環境

	項目	仕様
Client	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 1core
	Memory	1 GBytes
	SSH Client	OpenSSH 7.9
Server	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 1core
	Memory	1 GBytes
	SSH Server	OpenSSH 7.9
sshr	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 1core
	Memory	1 GBytes
	sshr	sshr v0.1.6
DB	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 1core
	Memory	1 GBytes
	Database	MySQL 5.7.27

```
# クライアント・サーバのみの場合
time ssh <username>@<SSH server IP> uname

# 提案手法（sshr）の場合
time ssh <username>@<sshr server IP> uname
```

図 7: 実行時間測定のためのコマンド例

5. 性能評価

これまで示したように，クライアント・サーバ間の SSH 接続に対して sshr が介入することで接続先の決定処理や認証処理をプログラマブルに拡張でき，その結果，システムの変更に対して，ユーザに変更を要求することなく追従可能となる．一方，セッション確立時に実行されるフック関数により，通信のオーバーヘッドが発生する．このオーバーヘッドを測定することで，提案する sshr の導入が SSH 接続の処理時間に与える影響を評価し，sshr が想定されうる実用的な環境に耐えうるか議論する．

図 6 に用いた実験環境の概要図を示す．実験は，sshr を介さないクライアントとサーバの通常の SSH 接続と，提案手法である sshr を介した SSH 接続で比較を行う．表 1 は，各ロールの性能と用いたソフトウェアのバージョンを示している．各ロールの OS は全て CentOS 7.6.1810 Kernel 3.10.0 である．評価においては，公開鍵認証を通してクライアント・サーバ間のセッションを確立し，uname コマンド [20] の実行結果が返ってくるまでの時間を time コマンド [21] で測定する．図 7 に実行時間を測定するためのコマンド例を示す．それぞれ 100 回測定を行い，平均値を算出した．sshr が実行するフック関数は，接続先の決定処理（FindUpstreamHook）およびユーザの公開鍵検索の処理（FetchAuthorizedKeysHook）のいずれにおいても，別サーバ上のデータベースから参照するものとする．データベースには目的のレコード以外を含まないものとする．

表 2 は，実験結果を示している．クライアント・サーバ間の SSH 接続に対して sshr を介した場合のオーバヘッド

表 2: 実験結果

環境	実行時間/msec（100 回平均）
クライアント・サーバのみ	448
提案手法（sshr）	470

は 22 ミリ秒であることが分かった．これは，SSH セッション確立時にユーザが遅延を感じないほど短い時間 [22] である．そのため，複数のユーザが複数のサーバ群の中から特定のサーバに対して SSH ログインし，サーバの操作を行うようなシステム管理を想定した場合，sshr のオーバヘッドは実用に十分に耐えうるほど小さい．一方，SSH 経由で大量のファイルを転送する場合など，サーバに対して大量にセッションを確立する必要があるケースでは，約 20 ミリ秒のオーバーヘッドが積み重なって，許容できない遅延時間になる可能性がある．また，sshr サーバはデータベースサーバに対して 2 回クエリを実行しているため，クエリの応答時間が sshr を介した場合のオーバヘッドに大きく影響を与える．そのため，参照するテーブルのレコード数が膨大な場合などは，データベース側にクエリの応答時間を短くする対策を行う必要がある．

6. まとめ

本論文では，ユーザに用いるクライアントツールの制限や変更を要求せず，システム管理者が自由に組み込み可能なフック関数を用いてシステム変化に追従できる sshr という SSH プロキシサーバを提案した．システム管理者は，sshr に対してユーザ名から接続先サーバを決定するためのフック関数を自由に実装・導入可能である．これにより，ユーザ名を用いて透過的に目的のサーバに SSH 接続できるため，接続先サーバの IP アドレスやホスト名に変更があった場合でも，管理者は各ユーザへ変更後の情報を知らせる必要がなく，ユーザも変更に追従する必要がない．さらに，sshr は，公開鍵認証の際にユーザの公開鍵を検索するためのフック関数を組み込めるなど，SSH 特有のユーザ認証もプログラマブルに拡張できる仕組みを採用しているため，システム管理者がユーザの公開鍵をデータベースなどの自由なデータ形式で管理することが可能である．さらに実験から，クライアント・サーバ間に sshr を導入した場合の SSH セッション確立のオーバーヘッドは 20 ミリ秒程度であり，ユーザが特定のサーバに SSH ログインする際に遅延を感じないほど短い時間 [22] であることを示した．

今後の展望として，まず sshr により汎用的な拡張の仕組みを実装することを考えている．現在の sshr では，システム管理者が Go 言語で実装した関数を組み込み，ビルドすることで，sshr の機能を拡張できる．これにより，拡張コードも含めてシングルバイナリが作成できるメリットがあるが，Go 言語での実装を強制することや，実装した拡張コードを配布・再利用できないなどの課題がある．この課題を解決するために，sshr にプラグイン機構の導入を検討していく予定である．また，今後は sshr を利用した実用的なシステムの構築を通して，sshr の有用性を評価していきたい．sshr の接続先決定のフック関数を用いて，ユーザが利用可能な複数のサーバ群から，周囲の状況に応じて最も適したサーバを選択し，提供することが有効ではないかと考える．例えば，機械学習などを用いてサーバの負荷状況を精緻に把握し，最も負荷が低いサーバをユーザに提供するといったことも可能である．

参考文献

[1] G. Galante and L. C. E. d. Bona: A Survey on Cloud Computing Elasticity, *2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pp. 263–270 2012.
[2] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano: A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments, *Journal of Grid Computing*, Vol. 12, No. 4, pp. 559–592 2014.
[3] B. Furht and A. Escalante: *Handbook of Cloud Computing*, chapter Service Scalability Over the Cloud, Springer 2010.
[4] T. Ylonen: The Secure Shell (SSH) Protocol Architecture, Technical report, RFC 4251 2006.
[5] Google Cloud Platform: gcloud command-line tool overview, https://cloud.google.com/sdk/gcloud/.
[6] Boshi Lian: SSH Piper, https://github.com/tg123/sshpiper/.
[7] H. Tsuruta: sshr, https://github.com/tsurubee/sshr.
[8] Google Cloud Platform: gcloud compute ssh, https://cloud.google.com/sdk/gcloud/reference/compute/ssh.
[9] The OpenBSD Foundation: ssh – OpenSSH SSH client, https://man.openbsd.org/ssh.1.
[10] HashiCorp: Consul, https://github.com/hashicorp/consul.
[11] Outbrain: Consult, https://github.com/outbrain/consult.
[12] The OpenBSD Foundation: sshd – OpenSSH SSH daemon, https://man.openbsd.org/sshd.8.
[13] T. Ylonen: The Secure Shell (SSH) Authentication Protocol, Technical report, RFC 4252 2006.
[14] Golang.org: The Go Programming Language, http://golang.org.
[15] golang: crypto/ssh, https://github.com/golang/crypto/tree/master/ssh.
[16] 川本安武：OpenSSH［実践］入門，技術評論社 2014.
[17] The OpenBSD Foundation: sshd_config – OpenSSH SSH daemon configuration file, https://man.openbsd.org/sshd_config.
[18] TeraTerm Project: Tera Term, https://ttssh2.osdn.jp/index.html.en.
[19] K. Oku: Server::Starter, https://github.com/kazuho/p5-Server-Starter.
[20] uname command – print Unix system information, https://www.unixtutorial.org/commands/uname.
[21] time – system resource usage for running a Unix command, https://www.unixtutorial.org/commands/time.
[22] J. Nielsen: *Usability Engineering*, chapter Response Times: The 3 Important Limits, Morgan Kaufmann Publishers 1993.