実践で学ぶ

一歩進んだサーバー構築・運用術

written by 仙石 浩明

第11回 ssh(後編)

1月号に引き続いてssh(セキュア・シェル)の応用方法を解説します。sshにはクライアント側やサーバー側のポートを反対側へ転送するポート・フォワード機能があります。この機能を使って,例えば持ち歩いているノートPCと社内LAN上のPCとの間で任意のプロトコルを使って安全に通信できます。

明けましておめでとうございます。いよいよ21世紀ですね。残念ながら「2001年宇宙の旅」は実現しそうにありませんが,20世紀中にインターネットが全世界的にここまで普及したことは驚くべきことと言えるでしょう。多くの国で比較的低コスト*1で手軽にインターネットに接続できるようになりました。日本に国際電話をかけて,劣悪な回線状況と戦いながらパソコン通信などでメールを読み書きしていたころ*2とは雲泥の差があります。

こうなってくるとますます重要になるのが,自分のサーバーをインターネット上に持つことです。インターネットに接続できても接続先のサーバーが無ければWebくらいしか使い道が無く,ひまつぶしの域を出ません。自分のサーバーがあれば,サーバーに多岐にわたるデータを蓄えておき,必要に応じて参照できます。

さらに,そのサーバーを経由して職場の内部LANにアクセスすることができれば,職場にいるのと同等の環境で仕事を行えます。それも通信コストをほとんどかけずにです。海外出張先のホテルの部屋で,職場内LANへつなぎっぱなしにすることも十分可能です*3。

そして,前回,前々回で解説してきたように,ssh(セキュア・シェル)を使えば出先からインターネット経由で社内LANにつなぐことができます。しかもsshは通信路を暗号化しますから,盗聴される心配せずに社内LAN上の機密情報を取り扱うことが可能です。

ポート・フォワード

前回少しだけ紹介したように,sshにはクライアント側あるいはサーバー側のポートを反対側へ転送するポート・フォワード機能があります。この機能を使えば任意のプロトコルを暗号化できるので重宝します。ポート・フォワード機能で真価を発揮するのは,Proxy Command*4を設定してファイアウオールを越えてssh接続を行っているときです。例えば出張に持参したノートPCと社内LAN上のPCとの間で任意のプロトコルを使って盗聴の心配なく通信できます。

ローカルからリモートへ

sshクライアントを実行したローカル・ホストのポートをsshサーバーが動いている内部LAN上のリモート・ホストに転送すれば,ローカル・ホスト上で実行する任意のクライアントから内部LAN上

*1 国によっては市内通話料金が格安であるために,日本にいるより低コストでインターネットに接続できます。
*2 わずか10年前の話です。
*3 実際,私は米国へ行ったときはつなぎっぱなしにしています。何時間つないでも1ドル以下の電話代で済みますから。
*4 sshでは,直接TCP/IP接続できないホストに接続するために,ProxyCommandをperlスクリプトで作成できます。ファイアウオール越えを行うProxyCommandを作成すれば,ファイアウオールの向こう側にあるホストを,同じLAN上にあるホストと同様に扱うことができます。連載第10回「ssh(中編)」参照。

のサーバーに接続することができます。例えば，klab.orgの内部LAN上のホストkamiyaへssh接続する際は，**図1**のように実行して，ローカル・ホストの10080番ポートを内部LAN上のWebプロキシproxy.klab.org*5の8080番ポートへ転送します。

図1中の「-L 10080:proxy.klab.org:8080」がローカル・ホストのポート（10080番）をリモート・ホスト（proxy.klab.orgの8080番ポート）へ転送するオプションです。コマンドライン・オプションで指定する代りに，~/.ssh/configで**図2**のように指定することもできます。

するとローカル・ホスト上のWebブラウザの設定で，Webプロキシをlocalhostの10080番ポートに設定すれば，klab.orgの内部LAN上の任意のWebサーバーにアクセスできるようになります（**図3**）。

ただし，内部LAN上のWebプロキシproxy.klab.orgが，内部LAN上のWebサーバーへのアクセスを許可していることが必要です。LANによっては，LAN上のWebサーバーにアクセスするときは，Webブラウザから直接接続させるために，Webプロキシ経由の接続を禁止しているかもしれません。

そのような場合は，LAN上のWebサーバーへアクセスするためのWebプロキシを新たに設置する必要があります。LAN上のWebサーバーへのアクセスだけならキャッシュする必要が無いので，stone*6の簡易httpプロキシ機能を使うこともできます。例えば，**図4**のようにstoneを動かします。

さて，これで出先のノートPCから内部LAN上の任意のWebサーバーにアクセスできるようになりました。ただしこのままでは，インターネット上のWebサーバーにアクセスするときにも内部LAN上のWebプロキシ経由でアクセスすることになり，少々非効率です。

Webプロキシの自動設定

NetscapeなどのWebブラウザであれば，アクセスするURLごとにWebプロキシを自動設定することができます。例えば，**図5**のような内容のファイル「/home/sengoku/proxy.pac」を作成し，Netscape Communicator 4.7の「Edit」メニューで「Preferences...」を選び，**写真1**のように「Automatic proxy configuration」を選択して「Configuration location（URL）」に

```
ssh -L 10080:proxy.klab.org:8080 kamiya
```

図1　ローカル・ホストのポートをリモート・ホストへ転送（その1）

```
Host kamiya
        ProxyCommand    /home/sengoku/bin/proxy-into %h %p
        LocalForward    10080 proxy.klab.org:8080
```

図2　ローカル・ホストのポートをリモート・ホストへ転送（その2）

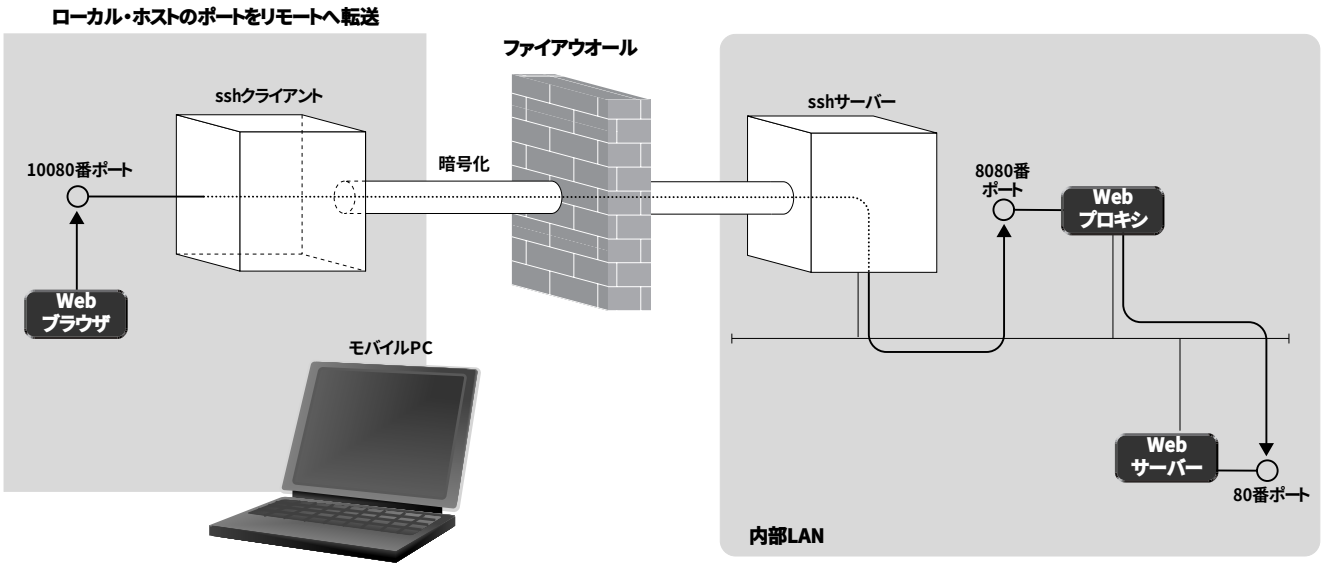

図3　インターネットから内部LAN上のWebサーバーへアクセス

「file:/home/sengoku/proxy.pac」を指定します。

図5のスクリプトは，Webブラウザがアクセスするurlごとにプロキシを自動的に設定するためのものです。このスクリプトはJavaScriptで記述します*7。

プロキシ自動設定スクリプトが設定されると，ブラウザは新しいURLへアクセスしようとするごとに，このスクリプトで定義された関数FindProxyForURLを，URLおよびホスト名を引数として呼び出します。例えば，「http://www.klab.org/」をアクセスする場合は，第一引数が「http://www.klab.org/」，第二引数が「www.klab.org」になります。

そしてFindProxyForURLの返り値に基づき，ブラウザはWebプロキシを自動設定します。例えば，返り値が「DIRECT」であればプロキシを設定せず，Webサーバーに直接アクセスしますし，返り値が「PROXY localhost:10080」であればlocalhostの10080番ポートのプロキシ経由でアクセスします。複数の設定を列挙することも可能で，例えば返り値「PROXY localhost:10080;DIRECT」は，まずlocalhostの10080番ポートのプロキシを使用し，もしこのプロキシが使用不可であれば，Webサーバーへ直接アクセス(「DIRECT」)します。

図5中，「dnsDomainIs(host, ".klab.org")」はホスト名のドメイン名部分が「klab.org」であれば「true」を返す関数です。つまりこのスクリプトは，

ドメイン名がklab.orgならば，ホスト名がwww.klab.orgで無い限りlocalhostの10080番ポートのプロキシを使用する。ホスト名がwww.klab.orgである場合や，ドメイン名がklab.orgで無い場合は，プロキシを使用せず直接Webサーバーへアクセスする。

という意味になります。

proxy.pacは，ファイルとして読むこともできますが，Webサーバーから読み込むこともできます。Webサーバーから読み込む場合，Webブラウザ側は，

```
stone -l proxy 8080 &
```

図4　簡易httpプロキシとしてstoneを実行

```
function FindProxyForURL (url,host) {
    if (dnsDomainIs (host,".klab.org")) {
        if (host == "www.klab.org") {
            return "DIRECT";
        }
        return "PROXY localhost:10080; DIRECT";
    } else {
        return "DIRECT";
    }
}
```

図5　プロキシ自動設定スクリプト「proxy.pac」の中身

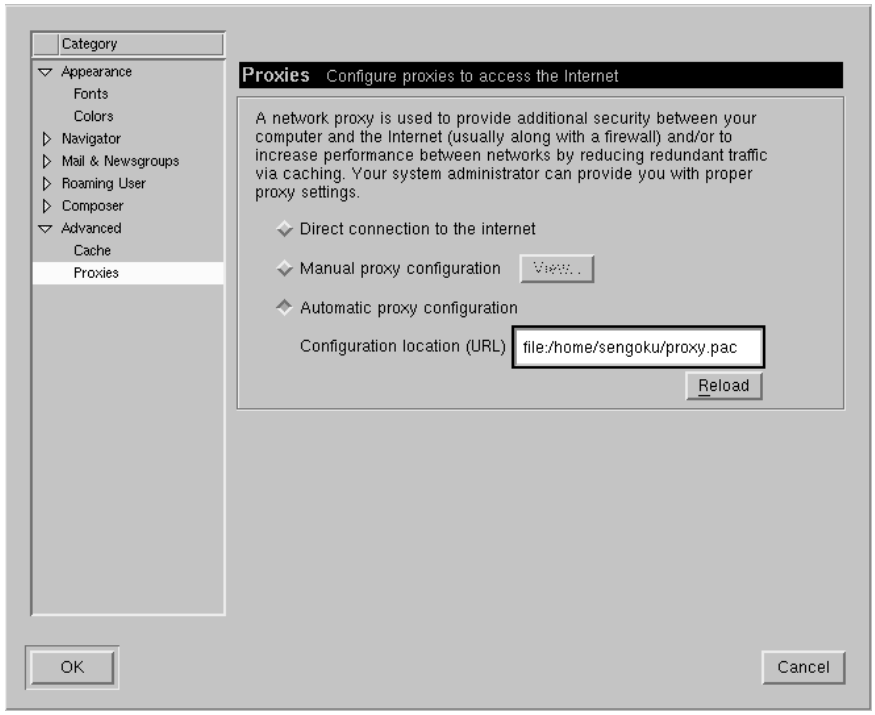

写真1　プロキシ自動設定スクリプトの指定

*5　proxy.klab.orgは，内部LANのみで有効なホスト名です。sshサーバーからproxy.klab.orgが到達可能であれば，転送可能です。
*6　stoneはTCPやUDPのパケットを中継するアプリケーション・レベルのパケット・リピータです。簡易httpプロキシ機能については，連載第5回「stone(前編)」を参照してください。
*7　記述方法の詳細については，http://home.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.htmlを参照してください。

写真1の「Configuration location (URL)」にproxy.pacがあるURLを入力するだけで良いのですが，Webサーバー側はproxy.pacを送信する際，「Content-Type」が「application/x-ns-proxy-autoconfig」になるよう設定しなければなりません。例えばApacheの場合であれば，設定ファイルhttpd.confに**図6**を加えるか，mime.typesに**図7**を加えます。

```
AddType application/x-ns-proxy-autoconfig .pac
```

図6　httpd.confでの設定

```
application/x-ns-proxy-autoconfig       pac
```

図7　mime.typesでの設定

```
ssh -R 10022:localhost:22 -R 10080:proxy.klab.org:8080 asao.gcd.org
```

図8　リモート・ホストのポートをローカル側へ転送

```
Host asao.gcd.org
        RemoteForward   10022:localhost:22
        RemoteForward   10080:proxy.klab.org:8080
```

図9　ローカル・ホストのポートをリモート・ホストへ転送

リモートからローカルへ

逆方向のポート・フォワード，すなわちsshサーバーが動いているリモート・ホストのポートを，sshクライアントを実行したローカル側のLAN上のホストへ転送することもできます。この機能を使えば，本来一方向のアクセスしか許可されていない環境下で，逆方向のアクセスが可能になります。例えば，内部LANからファイアウオールを越えてインターネットへアクセスするためのプロキシは用意されているけれども，インターネットから内部LANへアクセスする手段はセキュリティ上の理由などから一般ユーザーには提供されていないサイトが特に大企業などに多いのではないかと思います。あるいは，内部LANがプライベート・アドレス*で構築されていて，ファイアウォールのアドレス・ポート変換*機能により内部LANからインターネットへは任意の接続が可能だけれども，逆方向は接続不可能というケースは，特にケーブル・テレビ会社などが提供するインターネット接続サービスに多いでしょう。いずれの場合でも，内部LANからインターネット上のsshサーバーにssh接続できれば，逆方向のアクセスも可能になります。

例えばklab.orgの内部LAN上のホストkamiyaからインターネット上のホストasao.gcd.orgへssh 接続する際，**図8**のように実行すれば，ssh接続先

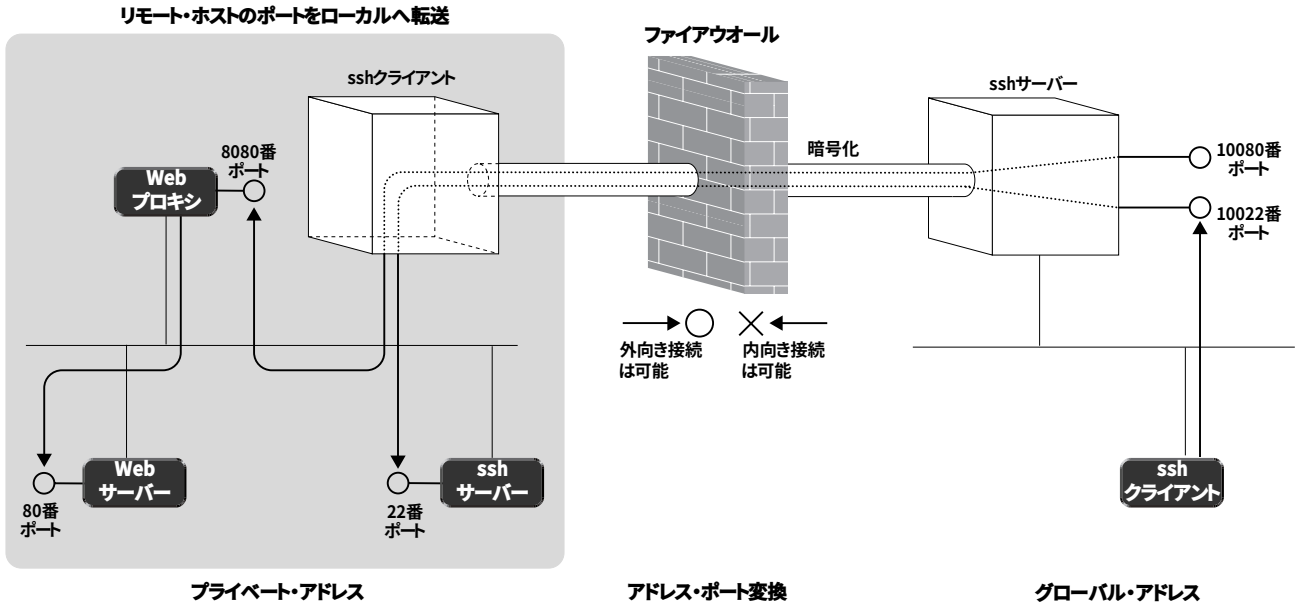

図10　プライベート・アドレス上にあるサーバーへのアクセス

(asao.gcd.org)の10022番ポートをローカル・ホストの22番ポートへ転送(「-R 10022:localhost:22」オプション)し, ssh接続先の10080番ポートを内部LAN上のWebプロキシproxy.klab.orgの8080番ポートへ転送(「-R 10080:proxy.klab.org:8080」オプション)します。

「-L」オプションと同様,「-R」オプションも, コマンドライン・オプションで指定する代りに, ~/.ssh/configに指定することもできます(**図9**)。するとリモート側でasao.gcd.orgの10022番ポートへssh接続すれば, 内部LAN上のホストkamiyaへssh接続できるようになりますし, リモート側で実行するWebブラウザの設定でasao.gcd.orgの10080番ポートをWebプロキシに設定すれば, 内部LAN上のWebサーバーへアクセスすることができるようになります(**図10**)。図10では, sshサーバーはインターネット上にありますが, 別のサイトの内部LAN上のsshサーバーであっても, ssh接続できる限り同様のことが可能です。

ただし, sshサーバー側から内部LANへアクセスできるのは, 図8のssh接続が続いている限りにおいてです。sshクライアントあるいはサーバーのいずれかのマシンをリブートすれば当然ssh接続は切れますし, リブートしなくてもネットワークの不調などでパケットが届かない状態が続けばssh接続が切れてしまうこ

```
#!/usr/bin/perl
$Interval = 60;
$opt{'asao.gcd.org'} =
    "-R 10022:localhost:22 ".
    "-R 10080:proxy.klab.org:8080";
$ENV{'PATH'} = "/bin:/usr/bin:/usr/ucb:/usr/local/bin";
$Child = 0;

sub alarm() {
    kill 1, $Child if $Child > 0;
    exit 0;
}

sub link() {
    my($host) = @_;
    $Child = open(SSH,"-|");
    if (!$Child) {
        exec "ssh", split(' ',$opt{$host}),
                    $host, "sh -c 'while :;do sleep 30;echo 1;done'";
    }
    $SIG{'ALRM'} = 'alarm';
    alarm($Interval * 2);
    while(<SSH>) {
        alarm $Interval;
    }
    close(SSH);
    sleep $Interval;
    exit 1;
}

for $host (keys %opt) {
    $pid = fork;
    if (!$pid) {
        &link($host);
    }
    $host{$pid} = $host;
}
while(($pid=wait) >= 0) {
    $host = $host{$pid};
    undef $host{$pid};
    $pid = fork;
    if (!$pid) {
        &link($host);
    }
    $host{$pid} = $host;
}
exit 1;
```

図11 rc.sshスクリプト
図8のssh接続を行い, 接続が切れたら再接続を行う。

```
su - nobody -c "/etc/rc.d/rc.ssh &"
```

図12 rc.ssh(図11)をブート時に実行

【プライベート・アドレス】 インターネットから直接アクセスする必要が無いサイト内部のネットワークで自由に利用できるIPアドレス。RFC1918で規定されている。インターネットから直接アクセスできる, インターネット上でユニークなIPアドレスは, グローバル・アドレスと呼ばれる。
【アドレス・ポート変換】 LinuxのIPマスカレード機能など。TCP/IPパケットの送信元アドレスをグローバル・アドレスへ変換することにより, プライベート・アドレスを割り当てられたホストからインターネットへのTCP/IP接続を可能にする仕掛け。

とはあります。そのたびに内部LAN に(物理的に)出向いて行って, 図8を実行し直すというのは面倒です*8。

常時ssh接続

そこで, ssh接続が切れたら再接続を行うスクリプトを書いて, ブート時に自動実行する方法を考えます。例えば, **図11**に示すrc.sshスクリプトを/etc/rc.d/rc.sshに置き, /etc/rc.d/rc.localに**図12**の行を挿入してブート時に実行します。セキュリティ上の理由から, rc.sshスクリプトを実行するユーザーはなるべく権限の無いユーザーにすべきです。ここではnobody権限で実行しています(図12)が, rc.sshを実行する専用のユーザーを作る方が望ましいでしょう。

rc.sshスクリプトは, リモート・ホスト上で**図13**のshスクリプトを実行します。つまり30秒に1回,「1」を標準出力へ出力し続けるプログラムです。そしてrc.sshは「1」が送られてくるかを監視します。もし60秒待っても「1」が送られてこなければssh接続が切れてしまったものと見なし, sshクライアントにHUPシグナルを送った後, 再接続を行います。

```
#!/bin/sh
while :
do      sleep 30
        echo 1
done
```

図13 リモート・ホスト上で実行するコマンド

なお, このrc.sshは複数のリモート・ホストに対してssh接続を維持できるように書いてあります。すなわちリモート・ホスト名をキーとし, そのリモート・ホストにssh接続する際のオプションを値とするopt連想配列に, リモート・ホストの数だけ要素を追加すればOKです。sshのオプションは何でも指定可能なので, ローカルからリモートへのポート・フォワードを行うことも可能です。

さて, これでポート・フォワードを常に維持し続けることが可能になったわけですが, 一点注意すべき点があります。それはパスフレーズの問題です。本連載第9回「ssh(前編)」で説明したように, sshのかぎを作成する際は必ずパスフレーズを設定すべきです。ところがパスフレーズを設定すると, ssh接続する際にパスフレーズを入力する必要があり, rc.sshスクリプトのように自動的に接続を行いたい場合に困ってしまいます。

ssh-agentを走らせておけば, 毎回パスフレーズを入力する必要は無くなりますが, リブートするとssh-agentを立ち上げ直してパスフレーズを入力する必要があり, rc.sshのようにブート時に実行されるスクリプトの場合はssh-agent は使えません。

従って常時ポート・フォワードを行う場合は, ポート・フォワード専用のsshかぎをパスフレーズ無しで作ることになります。万一かぎファイルを盗まれても影響を最小限に抑えるために, 極力権限の無いユーザーのかぎを作るべきです。そして次の2つの制約を設定して, かぎが使える状況を限定します。

(1) ssh接続元ホストを限定する

常時ポート・フォワードの場合, sshクライアントを実行するホストは常に同じですから, sshサーバーは特定のホストからの接続のみを受け付ければ十分です。ただし図10に示すように, sshクライアントとサーバーとの間でアドレス変換などを行うので, 接続元アドレスはsshクライアントを実行するホストにはなりません。アドレス変換を行うホストのアドレスになるでしょう。もちろん, 接続元アドレスはIPスプーフィングによって偽造可能ですから, この対策は万全ではないのですが, 安全性を高める方法としては有効でしょう。

(2) 実行するコマンドを限定する

rc.sshスクリプトの場合, sshサーバーで実行すべきコマンドは図13の内容に限られます。通常のssh接続のように任意のコマンドを実行する必要はありません。図13は, 30秒に1回,「1」を標準

```
from="napt.klab.org",command="sh -c 'while :;do sleep 30;echo 1;done'" 1024 35 23...2334 nobody@kamiya.klab.org
```

図14 ~/.ssh/authorized_keys
公開かぎ本体は長いので, 途中を省略しています。

```
tar cf - dir | (cd /some/where/.; tar xpf -)
```

図15 tarコマンドを使ってディレクトリごとコピー

出力へ出力し続けるだけのプログラムですから，このプログラムのみ実行できるように設定しておけば，万一かぎファイルを盗まれて，実行されてもさほど困ることはありません。

もちろん，特定のコマンドしか実行できなくても，ポート・フォワードを任意に設定されてしまうと安全がおびやかされますので，かぎは厳重に管理すべきです。

(1)，(2)共に~/.ssh/authorized_keysファイルにおいて，クライアントのかぎに対応する公開かぎの前にオプションを付けることにより，設定可能です。オプションは「,」で区切ることで複数指定できます。オプションに続く公開かぎを認証に用いた接続時のみ，そのオプションが適用されます。オプションとして以下の項目が指定できます。

●from="パターン"

接続を許可する接続元ホスト名のパターンを指定します。「*」と「?」をワイルド・カードとして使うことができ，「,」で区切ることにより複数のパターンを指定できます。パターンの前に「!」を付けると，パターンにマッチしないホスト名からの接続のみを受け付けます。

●command="コマンド"

ssh接続を受け付けたとき実行するコマンドを指定します。sshクライアントから送られてきたコマンドは無視します。

●environment="環境変数名=値"

sshクライアントから送られてきたコマンドを実行する前に，環境変数を設定します。

●no-port-forwarding

ポート・フォワードを禁止します。

●no-X11-forwarding

Xプロトコルの転送*9を禁止します。

●no-agent-forwarding

ssh-agentプロトコルの転送*10を禁止します。

従って，rc.sshを受け付けるサーバーasao.gcd.orgの~nobody/.ssh/authorized_keysファイルの設定は，「from="パターン"」と「command="コマンド" 」のオプションを使って**図14**のように記述すると良いでしょう。図14中のnapt.klab.orgは，アドレス変換を行うホスト名*11です。

他のコマンドとsshの組み合わせ

sshはrshと置き換え可能ですから，rshと組み合わせて使うコマンドは当然sshと組み合わせて使うこともできます。sshを/usr/bin/rshとは別にインストールして，sshとrshを使い分けることもできますが，ssh接続に失敗したとき自動的にrshへ切り替える機能*12をsshは持っていますから，sshを/usr/bin/rshとしてインストールして完全に置き換えてしまう方がお勧めです。

こうすると，内部でrshを呼び出すrdistやrsyncコマンドなどでも，意識せずにsshを利用できます。

tar

昔，cpコマンドには，-aオプションがありませんでした。シンボリック・リンクなどを含むディレクトリをそのままコピーするには，cpコマンドではなくtarコマンドなど*13を使って，**図15**などのようにコピーするのが普通でした*14。

tarコマンドを使うこの方法の良いところは，これをそのままリモート・ホストへのディレクトリ・コピーで応用できる点です(**図16**)。

sshを/usr/bin/rshにインストールしている場合は，図16の手順のままでsshを使ったリモート・ホストへディレクトリをコピーできます。sshをrshとは別にインストールしている場合でも，図16の「rsh」を「ssh」に置き換えるだけ

```
tar cf - dir | rsh remotehost "cd /some/where/.; tar xpf -"
```

図16　tarコマンドを使ってリモート・ホストへディレクトリをコピー

＊8　内部LAN上に電話回線経由でアクセスできるサーバーを用意しておいて，切れたらダイヤルアップ接続して図8の手順を実行し直すようにすれば，出向く必要は無いかも知れません。しかしインターネットにはアクセスできるけれどモデムは無い場所にいるときとか，海外にいるときは困ってしまいます。
＊9　リモート・ホストのポートをローカル側へ転送する機能のX版です。リモートで実行したXクライアントをローカル側のXサーバーに表示できます。
＊10　リモート・ホストのポートをローカル側へ転送する機能のssh-agent版です。リモートで，ローカル・ホストのssh-agentを利用できます。
＊11　実在はしません。
＊12　rsh，rlogin，rcpコマンドを/usr/libexecなどに移して，sshをmakeするときconfigureに，--with-rsh=/usr/libexec/rshオプションを指定します。
＊13　tarの代りにcpioコマンドを使う方法もあります。
＊14　cpに-aオプションがサポートされた現在でも，指が覚えていてついtarコマンドを使ってしまう，という人が多いのではないでしょうか。

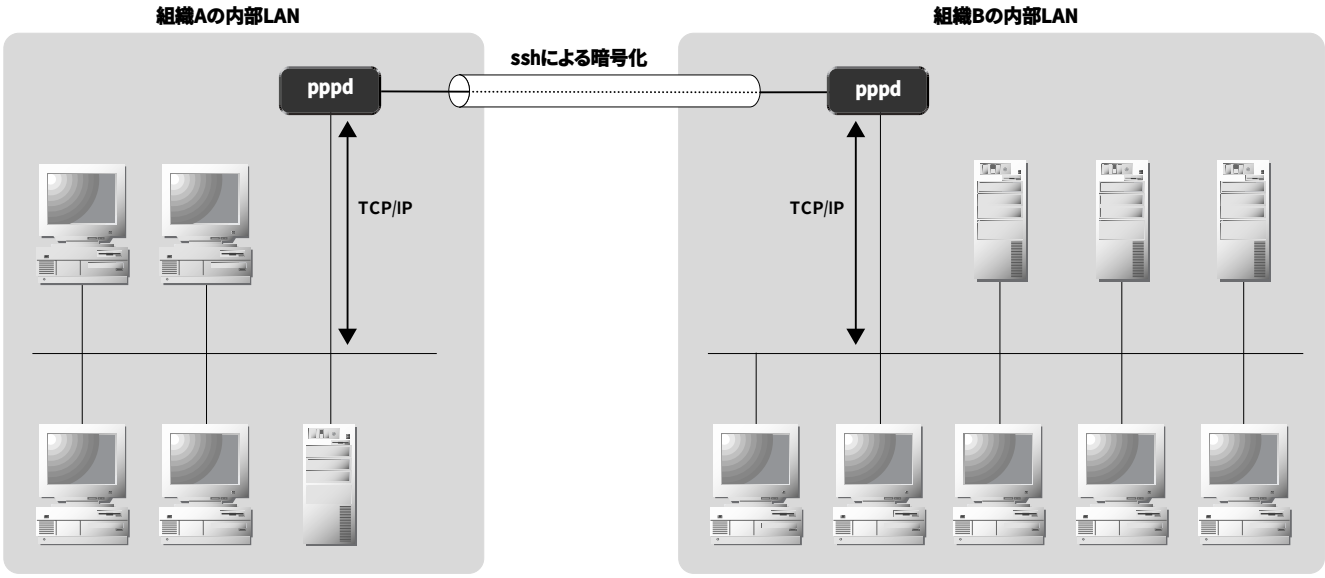

図17　pppdとsshを用いたVPN

```
#!/usr/bin/perl
$user = getlogin || (getpwuid($<))[0] || die;
open(PS,"ps auxw |") || die;
while(<PS>) {
    if (/^$user\s+(\d+).*[\s\(\[]ssh-agent([\s\)\]]|$)/) {
        $pid = $1;
        last;
    }
}
close(PS);
if (! $pid) {
    if (@ARGV == 1 && $ARGV[0] eq "-csh") {
        exec "ssh-agent";
    } else {
        die "Can't find ssh-agent.\n";
    }
}
loop: for ($i=1; ; $i++) {
    die "Can't find socket.\n" if $i > 20;
    $j = $pid - $i;
    $sock = "/tmp/ssh-$user/ssh-$j-agent";
    last if -S $sock;
    $sock = "/tmp/ssh-$user/agent-socket-$j";
    last if -S $sock;
    while (</tmp/ssh-*/agent.$j>) {
        $sock = $_;
        last loop if -S $sock;
    }
}

while (@ARGV > 0) {
    if ($ARGV[0] eq "-sh") {
        shift;
        print "SSH_AUTH_SOCK=$sock;\nSSH_AGENT_PID=$pid;\n";
```

図18　ssh-envスクリプト(その1)

で済みます。

rdist

rdistコマンドもディレクトリをリモート・ホストへコピーするためのコマンドですが，リモートとローカルのディレクトリを比較して，異なる部分だけを転送する点がtarと異なります。2つのマシンの特定のディレクトリの内容を同一に保ちたいときなど，毎回ディレクトリ全体をコピーするのでなく，変更された部分だけをコピーできるので便利です。

昔のバージョンのrdistはrshの機能を内蔵していたのですが，1994年2月以降（Version 6.1beta.2以降），rdistは内部でrshを呼び出すように変更されました。従って，sshをrshと置き換えてインストールしている場合は，何の変更もなくrdistでsshが使えます。rshとは別にsshをインストールしている場合は，rdist実行時に「-P /usr/bin/ssh」などとsshを使用するためのオプションを指定する必要があります。

どちらの場合も，環境変数をたくさん使っている場合，その個数に注意する必要があります。rdistはrsh（ssh）を呼び出す際に環境変数を引き継ぐのですが，その個数には上限があります。rdistはVersion 6.1.4では，src/setargs.cのMAXUSERENVIRONで上限が設定されています。デフォルトではこれが40になっているようです。

MAXUSERENVIRONで設定した以上の個数の環境変数を使っている場合，ssh-agentの実行で設定した環境変数SSH_AGENT_PID，およびSSH_AUTH_SOCKがsshに引き継がれなくなってしまいます。私は環境変数を100個近く使っている＊15ため，MAXUSERENVIRONの値を200に設定して，rdistをmakeし直しました。

rsync

rsyncはrdist同様，2つのマシンの特定のディレクトリの内容を同一に保つことができます。rdistがクライアントからサーバーへのコピー専用であるのに対し，rsyncはサーバーからクライアントへコピーするときも使えます。

rsyncも内部でrshを呼び出すので，sshをrshと置き換えてインストールしている場合は，何の変更もなくrsyncでsshが使えます。rshとは別にsshをインストールしている場合は，rsyncの実行時に「-e /usr/bin/ssh」などとsshを使用する

＊15 1989年に作った~/.cshrcをいまだに使い続けていますから，10年以上の歴史がある私独自の環境変数があったりします。

```
        exit 0;
    }
    if ($ARGV[0] eq "-csh") {
        shift;
        print "setenv SSH_AUTH_SOCK $sock;\nsetenv SSH_AGENT_PID $pid;\n";
        exit 0;
    }
    if ($ARGV[0] eq "-tty") {
        shift;
        $tty = 1;
        next;
    }
    if ($ARGV[0] eq "-d") {
        shift;
        $debug++;
        next;
    }
    last;
}
if (@ARGV == 0) {
    print "Agent pid $pid\n";
    exit 0;
}

$ENV{'SSH_AGENT_PID'} = $pid;
$ENV{'SSH_AUTH_SOCK'} = $sock;
#$ENV{'SSH_AUTHENTICATION_SOCKET'} = $sock;

if ($tty) {
LOOP:
    foreach $p ("p", "q", "r", "s") {
        foreach $q (0..9, "a".."f") {
            if (open(PTY,"+>/dev/pty$p$q")) {
                $tty = "/dev/tty$p$q";
                $success = 1;
                last LOOP;
            }
        }
    }
    die "Can't open pty.\n" if ! $success;
    print "tty: $tty  command: ".join(" ",@ARGV)."\n" if $debug;
    open(TTY,"+>$tty") || die;
    if (!fork) {
        close(PTY);
        open(STDIN,"<&TTY");
        open(STDOUT,">&TTY");
        open(STDERR,">&TTY");
        exec @ARGV;
    }
    close(TTY);
    while(<PTY>) {
        print;
    }
    wait;
    exit $?;
}
exec @ARGV;
```

図18 ssh-envスクリプト（その2）

ためのオプションを指定する必要があります。

pppd

pppdコマンドは，IP的に直接繋がっていない2つのホストの間に，IPパケットを流すことができるトンネルを設定するコマンドです。プロバイダへダイヤルアップしてPPP接続するときに使われる＊16ので，ダイヤルアップ専用のコマンドと思っている人も多いかも知れませんが，電話回線だけでなくssh接続を含む任意の通信路にIPパケットを流すことができます。

従ってpppdとsshdを組み合わせて使えば，ssh接続のクライアント側とサーバー側の内部LAN同士で直接IPパケットを送受信することが可能になります（**図17**）＊17。いわゆるVPN（仮想プライベート・ネットワーク）ですが，セキュリティ的にはあまり望ましい形態ではありません。片方のLANに侵入されれば他方のLANにまで影響がおよびます。もともとsshにはクライアント側とサーバー側が一蓮托生になる性質を持っているのですが，VPNはその究極の姿と言えるでしょう。

セキュリティの基本は，必要最小限のアクセスのみを許可し，それ以外のアクセスを排除することによって侵入のリスクを最小化することにあります。必要の無いパケットまで相互に流通させてしまうVPNは，本当に必要なのかを十分に検討した上で使うべきか否か判断すべきでしょう。もしsshのポート・フォワードだけで用が足りるのであれば，VPNは使うべきではありません。

VPNは暗号化を行うから安全，というイメージを抱く人が多いのですが，VPNを構成するLANの全てが完璧に守られていてはじめてVPNも安全と言えるのです。だれもがLANからインターネットへWebブラウザなどで手軽にアクセスすることが当たり前になってしまった現在では，LANの規模が大きくなればなるほど安全ではなくなります。安全でないLAN同士をVPNで結べば，よりいっそう安全でなくなるのは当然と言えるでしょう。

ssh-agentの再利用

連載3回にわたってsshを解説しましたが，いかがだったでしょうか。まだ書き足らないところがあるほど奥の深いsshですが＊18，とりあえず今回はこれでおしまいということにして，最後に1つ便利なスクリプトを紹介します。

ssh-agentはパスフレーズを覚えてくれている便利なデーモンですが，環境変数を設定しないと利用できないのは不便です。もちろん環境変数ですから，おおもとのプロセスで設定すれば子プロセスに引き継がれていくのですが，ログアウトしてしまった後でもう一度使いたい，と思うことはよくあることです。もちろん使わないときは，ssh-agentデーモンを終了させるべきなのですが，毎日使うマシンであればssh-agentデーモンを動かし続けたいと思うのが人情でしょう。

そんなとき，コマンド一発でssh-agentを利用するための環境変数を設定することができたら，と思って作ったのが**図18**に示すssh-envスクリプトです。sshの複数のバージョンに対応しているため冗長になっていますが，psコマンドを使ってssh-agentのプロセスIDを探し，それに対応するUNIXドメイン・ソケット探して環境変数を設定するためのコマンドを出力します。

~/.cshrc に，**図19**に示すalias設定を入れておけば，コマンド・プロンプトで「ssh-env」と入力するだけで，既に動いているssh-agentデーモンを再利用できます。もし，ssh-agentデーモンが動いていなければ新たに立ち上げてくれます。

また，このスクリプトを使えば，cronなどから呼び出すプログラムからssh-agentを利用することも可能になります。例えばcrontabで**図20**のように設定します。ssh-envスクリプトが環境変数を設定した上でrdistコマンドを呼び出すので，rdistが呼び出すsshにもその環境変数が引き継がれ，パスフレーズを入力する必要無くssh接続できます。

```
alias   ssh-env 'eval `perl $HOME/bin/ssh-env -csh`'
```

図19　ssh-envスクリプトを使うためのalias設定

```
33 4 * * *      $HOME/bin/ssh-env rdist azabu
```

図20　cronでsshを使う

＊16　RedHat系のLinuxでしたら，ネットワーク設定ツールnetcfgを使ってダイヤルアップ接続の設定を行うと，自動的にpppdコマンドが使われる設定になります。
＊17　具体的な方法は，VPN HOWTO（http://www.linux.or.jp/JF/JFdocs/VPN-HOWTO.htmlなど）を参照してください。
＊18　書き尽くそうと思えば，軽く本1冊くらいの分量になるでしょう。