OpenSSH: SSH, SCP, and SFTP

1 Introduction

The OpenSSH suite consists of sshd, the SSH server, and a set of client programs: ssh, scp, and sftp. It is one of the most important programs for remote access and administration of Linux/UNIX systems. OpenSSH is included with virtually all Linux distributions, and the server is often installed and run by default. One of the key features of SSH is that it *encrypts all communications*, so it provides secure communications over the Internet. SSH can replace all of the following (insecure) alternatives: telnet, rsh, rlogin, rcp, ftp. These programs and their servers should *never* be used anymore (running an *anonymous ftp server* to serve files to anyone is OK).

The SSH clients can authenticate users via a standard *password* approach, or they can use a *public key method*. The public key method can be used to allow remote logins without having to supply any kind of password/passphrase. Another important feature of SSH is that it *authenticates the SSH server*. This can prevent what are called *"man in the middle"* attacks, where a machine other than the intended target server captures all packets and pretends to be that machine. By securely pre-fetching the server's public key, SSH clients can ensure they are connecting to the intended server.

The main OpenSSH client programs:

- ssh – For logging into a remote machine, and getting a shell.
- scp – For copying files between hosts on a network.
- sftp – An interactive file transfer program.

2 Basic Client Usage

2.1 ssh

If a remote machine has an SSH server running, the basic command to connect to that server is:

```
ssh username@hostname
```

(if you do not supply a username, your current username will be used)

This command will attempt to connect to machine `hostname` and log you in. By default, you will be prompted to enter the password for `username` (this is the password on the machine `hostname`). Once that is properly supplied, you will get a shell prompt and be able to run commands on the machine `hostname` as `username`.

2.1.1 ssh and remote commands

What makes `ssh` particularly powerful for remote administration is that it can also be used to *execute a single command* on a remote machine, without the need to start an interactive shell session. This command can also be part of a *pipeline* on the local machine.

The basic syntax to run a command on a remote machine is:

```
ssh username@hostname 'COMMAND_LINE'
```

`COMMAND_LINE` can be any legal Bash command line, which may contain multiple simple commands separated by ";"'s or pipelines, etc. Note that unless `COMMAND_LINE` is a simple command, it is good to *quote it* as shown.

For example, suppose you want to count the number of files in `username`'s `Documents` directory on machine `hostname`. This can be done as:

```
ssh username@hostname 'ls Documents | wc -l'
```

This causes the `ls-wc` pipeline to be run on `hostname`, but the output shown on the local machine. This is simpler than starting an interactive shell session, running the above command, and then having to exit the shell session. Furthermore, we will see how such remote commands can easily be incorporated into shell scripts to automate remote operations.

One of the most versatile aspects of the remote command capability is that you can use it in conjunction with *piping* and *redirection*. Consider the following alternative for counting files in `username`'s `Documents` directory on machine `hostname`:

```
ssh username@hostname 'ls Documents' | wc -l
```

The difference with the earlier command is that the first command does the counting remotely (`wc` runs on `hostname`), while this second command does the counting locally (`ls` is run on `hostname` and its output piped back to the local machine to count).

The two file counting examples also make clear the importance of properly *quoting* `COMMAND_LINE`. If quotes were not used around the entire pipeline in the first example, it would be interpreted like the second example (where the `wc` command will be executed on the local machine). While the results here will be the same, they generally will not be, so one needs to be vigilant about proper quoting.

The ability to run remote commands that involve piping and redirection makes it possible to implement many useful command patterns. Here are some examples:

- Run `ls` on the remote machine and store output to a remote file:
  ```
  ssh username@hostname 'ls -l > ~/temp/remote-ls'
  ```
- Run `ls` on the remote machine and append output to a remote file:
  ```
  ssh username@hostname 'ls -l Documents >> ~/temp/remote-ls'
  ```
- Run `ls` on the remote machine but store output to a local file:
  ```
  ssh username@hostname 'ls -l' > ~/temp/remote-ls
  ```
  (note the placement of the quotes here!)
- Print the local file `test` on a remote machine:
  ```
  cat test | ssh username@hostname 'cat | lpr'
  ```
- Another way to do the same thing:
  ```
  ssh username@hostname 'cat | lpr' < test
  ```
- Append the local file `test` onto the remote file `junk`:
  ```
  cat test | ssh username@hostname 'cat >> temp/junk'
  ```

- Another way to do the same thing:
  `ssh username@hostname 'cat >> temp/junk' < test`
- Create a local tar file backup of a remote directory (`~/downloads`):
  `ssh username@hostname 'tar czf - downloads' > hostname-downloads.tgz`
- Create a remote tar file backup of a local directory (`~/mail`):
  `tar czf - mail | ssh username@hostname 'cat > backup-mail.tgz'`
- Use `kwrite` to edit a file on a remote machine:
  `ssh -f -X username@hostname kwrite temp/test.text`
  (f option backgrounds ssh to get shell prompt back, X option is to display on local machine, but note that these options may not be required with most distro's default client configuration)

2.2 scp

If remote machine(s) have an SSH server running, the basic command to copy a file *to* that machine is:
`scp file-pathname1 username@hostname:file-pathname2`
(if username is ommitted, the current username will be assumed, if the filename in file-pathname2 is ommitted, the filename will be unchanged)

This command will attempt to copy the file on the localhost to the file/directory on hostname. By default, you will be prompted to enter the password for username on the machine hostname. Once these are properly supplied, the file(s) will be copied.

If remote machine(s) have an SSH server running, the basic command to copy a file *from* that machine is:
`scp username@hostname:file-pathname1 file-pathname2`
(if username is ommitted, the current username will be assumed, if the filename in file-pathname2 is ommitted, the filename will be unchanged)

This command will attempt to copy the file on hostname to the file/directory on the local machine. By default, you will be prompted to enter the password for username on the machine hostname. Once these are properly supplied, the file(s) will be copied.

If you want to copy a file from one remote machine to another, you can specify hostnames for each file. For this to work, however, you must have passwordless access to hostname1 (see Section **??**).

Multiple files may also be specified, to be copied to a directory, just as with `cp`, including via use of filename metacharacters like `*`. The `-r` option can be used to recursively copy a directory.

scp does have trouble dealing with pathnames/filenames that contain spaces on the remote machine, so to copy such a file from the remote machine you have to "double escape" the spaces, using both quotes and escape chars, e.g.,:
`scp username@hostname:temp/'file\ with\ spaces\ in\ name' temp`

2.3 sftp

If a remote machine has an SSH server, the basic command to start an SFTP session on that server is:
`sftp username@hostname`
(if you do not supply a username, the current username will be used)

This command will attempt to start up the `sftp-server` program on `hostname`. `sftp-server` is a separate program/subsystem from `sshd`, which is run as a subprocess by `sshd` (after it completes authentication).

By default, you will be prompted to enter the password for username (on the machine `hostname`). Once that is properly supplied, you will get an `sftp` prompt and be able to run commands to get or put files.

The SFTP commands are basically identical to those for FTP:
`cd`, `lcd`, `get`, `put`, etc.
(Note that `get` and `put` can take wildcards, like FTP's `mget` and `mput`.)

3 Machine Authentication and Keys

Every SSH server has a key pair that identifies the server machine (actually, there may be multiple key pairs, using RSA, DSA, etc.). Each key pair consists of a public key meant to be distributed to clients and a private key that must be kept secret for security. When a client attempts to connect to an SSH server, it looks in the file `~/.ssh/known_hosts` to find the public key of the server. This key will be used to authenticate the server—i.e., to prove that the server that is responding is actually the correct server.

If a user connects to a server and does not have the server's public key in the `known_hosts` file, the ssh client displays a hash of the key and asks if the user wants to connect and accept the key. If you say yes, the key will be stored iun the `known_hosts` file for future connections. A more secure approach is to obtain the server's public key by some other, secure means (e.g., carry it on a flash drive), and store it in the `known_hosts` file prior to ever attempting to connect to the server. A server's keys are found in `/etc/ssh`, and have names `ssh_host_*_key*`.

If a server fails the key test, the client will print a warning messages and fail to connect. Usually, this happens because the OS (and SSH server) have been reinstalled. If a key failure occurs, the user needs to verify that this is due to an OS (or server) change, and delete the existing key from the `known_hosts` file.

4 sshd (Server) Configuration

The default configuration file for sshd on most distributions is `/etc/ssh/sshd_config`. Key settings that one will want to check and possibly change:

- `ChallengeResponseAuthentication` – This option controls whether the *"keyboard-interactive"* authentication scheme (defined in RFC-4256) is allowed or not. This authentication scheme can use several possible mechanisms, but is typically set up (via PAM) to prompt for the user's (system password). Security can be increased by having this option set to `no`, and using key-based authentication (see Section **??**).
- `MaxAuthTries` – Determines how many authentication failures are allowed before the connection closes. For more security, you can set this to 2 or even 1. Note, though, that while this can slow down password attacks, it cannot prevent them because it simply closes the current connection, and an attacker can then simply re-connect.
- `PasswordAuthentication` – This option controls whether standard *"password"* authentication (defined in RFC-4252) is allowed or not. Security can be increased by having this option set to `no`, and using key-based authentication (see Section **??**). Note that newer versions of OpenSSH required that `ChallengeResponseAuthentication` also be set to no to ensure password-based authentication is completely disabled.
- `PermitRootLogin` – Determines whether someone can attempt to login directly as root (administrator). For highest security, you should set this to: "no". While this means that root will have to login as a

normal user and then use su to become root, it can help prevent brute-force password attacks. This does make it impossible to run `scp` and `sftp` directory as root, which can be annoying. It is certainly more secure, though. An alternative is to set it to "without-password" which will allow direct root login using public key authentication (see Section **??**).

- `Protocol` – Determines which ssh protocol (1 and/or 2) clients are allowed to connect with. For security, you should make sure that this is just "2", not "1,2".
- `PubkeyAuthentication` – Specifies whether *"publickey"* authentication is allowed (see Section **??**). Security can be increased by having this option set to `yes`, and disabling password-based authentication (see Section **??**).
- `UseDNS` – Specifies whether to check if resolved host name for the remote IP address maps back to the very same IP address. The default is yes. Results in slight delay making connections, and can cause considerable delay on LANs should DNS connectivity be lost.
- `X11Forwarding` – Specifies whether X11 forwarding is permitted. Usually yes in modern distributions. See Section **??** for further information.

5 ssh (Client) Configuration

The default configuration file for sshd on most distributions is `/etc/ssh/ssh_config`. Key settings that one will want to check and possibly change:

- ForwardAgent – Specifies whether the connection to the authentication agent (if any) will be forwarded to the remote machine. See Section **??** for further information.
- ForwardX11 – Specifies whether X11 connections will be automatically redirected over the secure channel. "yes" means that when one starts a GUI/X11 program from an ssh session, the program will run on the remote machine, but will display back on the machine the ssh client was started on. All traffic to the GUI program is encrypted, basically being carried in an ssh tunnel (see Section **??**).
- Protocol – Determines which protocol clients can connect with. Should be only "2".

6 Securing SSH

While SSH authenticates servers and users, and encrypts network traffic, default installs often leave SSH servers *vulnerable to some network attacks*. The primary security concern for SSH servers is *brute-force password guessing attacks*. There are various tools that can easily be found on the Internet that allow an attacker to automate the process of repeatedly trying to login to an SSH server with different passwords and usernames. *This is one the most used types of attacks on Linux systems!* In part this is because many people mistakenly believe that since SSH is "secure," it must be immune/resistant to such (basic) attacks, so they do not bother to do anything to secure their SSH servers. Unfortunately, OpenSSH does not itself include mechanisms designed specifically to protect against password guessing attacks,

Luckily, there are a number of steps one can take to secure an SSH server against such attacks:

- Disable direct *root logins* or allow only *key-based* root login by changing the `PermitRootLogin` setting in `sshd_config`.
- Force users to have only very strong passwords, e.g., by setting strength checking for the `passwd` program with PAM's `pam_cracklib` module, and periodically auditing user passwords (by attempting to crack them).
- Disable password-based authentication for all users, forcing the use of *key-based authentication* (see Section **??**): change the `PasswordAuthentication` and `ChallengeResponseAuthentication` settings to no, and the `PubkeyAuthentication` setting to yes, in `sshd_config`.

- Limit IP addresses that can connect to the server, using the standard **hosts access files** (see more on this below).
- Limit IP addresses that can connect to server, using the firewall.
- Block attackers using a program such as Fail2ban, DenyHosts, SSHblock, etc., which look at login failures and block IP addresses after several failures.
- Limit the *rate* of server connections via the firewall.

The most common user target for brute-force password-guessing attacks is of course root, since logging in as root gives one complete control and since every Linux/UNIX system has a root user (so username guessing is not required). Disallowing direct root logins completely neutralizes password guess against root (the attacker cannot tell that root logins have been disabled, so will guess fruitlessly). Remotely logging in as root will then require first loggin in as a normal user and using `sudo` or `su` to execute commands as `root`.

Password-guessing attacks typically use "dictionaries" of common passwords, dictionary words, proper names, etc. Obviously, if users' passwords are long, random character strings, password-guessing will be much more difficult. PAM can be setup to check user passwords when they are changed, and accept only "strong" ones. A sysadmin can also periodically run a program like John the Ripper against user passwords to check strength.

Password-guessing attacks rely on SSH servers using password-based authentication for users. Clearly, if password-based authentication is disabled, the server is virtually completely secure against such attacks. How does one then login? Using *key-based authentication*—see Section **??**.

Few SSH servers truly require that any IP address in the world be allowed access. Limiting the IP addresses that are allowed to connect to a server can have a significant effect on preventing brute-force password attacks (or other attacks) against the server. One approach for limiting access is via the *host access control files* (what were originally called *"TCPWrappers"*). `sshd` is usually configured to observe the access specifications in the standard server access control files `/etc/hosts.deny` and `/etc/hosts.allow`. The best security is provided by having `/etc/hosts.deny` contain the line: `ALL:ALL` (to deny all addresses by default), and then having an `sshd` line in `/etc/hosts.allow` that specifies the IP ranges from which SSH connections are allowed. E.g.,: "`sshd: 131.230.133.*`". This would permit logins from only cs.siu.edu IP addresses. Note that you can also use *hostnames*, but if DNS becomes unreachable, your SSh server may then disallow all access, so using IP addresses is preferred. See "`man 5 hosts_access`" for further information on the syntax for the access files.

If you absolutely must allow password-based authentication and access from IP addresses ranges you cannot trust, then it is critical to use other approaches to secure an SSH server, or it *will* be subject to brute-force attacks. Many people are suprised that `sshd` does not include any mechanism for automatically blocking IP addresses after a certain number of failed login attempts (`MaxAuthTries` does *not* do this!). However, there are many add-on programs that can be used to do this. Generally, they monitor log files for SSH authentication failures, and block IP addresses either by adding them to the `hosts.deny` file or by adding a firewall rule. Most Linux distros will include at least one of the above listed monitoring-blocking programs.

7 Key-Based Authentication

An alternative to password-based authentication for users connecting to a remote ssh server is to use *public key authentication*. With this approach, a user creates a public-private key pair on their local machine, and then distributes the public key to all the machines he wants to connect to. When an SSH client attempts to connect to one of these machines, the key pair will be used to authenticate the user, instead of the user being prompted for his password.

Distributing the public key is accomplished by appending it to the `~/.ssh/authorized_keys` file in the user's directory on each remote machine. This can be accomplished (via password authentication) using this ssh command:

```
cat id_rsa.pub | ssh username@hostname 'umask 077; cat >> .ssh/authorized_keys'
```

(your key may be in the file `id_dsa.pub` instead). Note that the `umask 077` is in the command because SSH requires that this file be readable only by the owner.

A key pair is generated with the `ssh-keygen` command. One of the decisions to make when creating a key pair is whether it will have a *passphrase* or not. If you create a key with a passphrase, you will be prompted for this passphrase every time you attempt to connect to the server you copied the key to. The `ssh-keygen` man page says this about passphrases: "Good passphrases are 10-30 characters long, are not simple sentences or otherwise easily guessable, and contain a mix of upper and lowercase letters, numbers, and non-alphanumeric characters. The passphrase can be changed later by using the -p option. There is no way to recover a lost passphrase."

Having a key with a passphrase will not appear much different from password authentication, but it can allow you to use the same passphrase to login to multiple remote machines and the `ssh-agent` program can help by automatically providing passphrases when required (see below). If you choose to create a key without a passphrase (just hit return at the passphrase prompt), you will be able to be logged in to a remote server without having to type a password or passphrase. This provides extreme ease of use, plus the ability to use SSH in automated scripts. However, it also means that *should anyone gain access to your private key, they would be able to login to your account on any machine that you set the public key up on.* Note also that since root can access all files on most Linux systems, any administrator on your machine could potentially gain access to your private key and thus your remote files!

The `ssh-agent` program can help ease the burden of providing passphrases, as it can automatically supply your passphrase when required. Once `ssh-agent` is running, use the `ssh-add` program to add your key(s). It is at this point that you are prompted for your passphrase(s). From this point on in the session, whenever an SSH client would need a passphrase, the agent supplies it. You will want `ssh-agent` to start automatically on login. Different distributions can have different ways of doing this. Many use software called `keychain` to automatically start both `ssh-agent` and `gpg-agent`. There are other alternatives as well, such as `envoy`.

SSH even understands how to forward passphrase requests back through a chain of connections, to `ssh-agent` running on the ultimate originating machine (i.e., if you ssh from host1 to host2 and from there ssh to host3, the passphrase request can be sent back to host1). This is known as *agent forwarding*, and it must either be enabled in the client configuration files on the machines (see Section **??**), or the `-A` option to `ssh` must be used when each connection is started.

8 Secure Tunnels and Port Forwarding

SSH can be used to forward TCP traffic through an SSH connection. When an SSH connection is used in this way, it is known as a *tunnel*. All traffic through the tunnel is *encrypted*, allowing an insecure protocol like POP to be secured over the Internet, much as with a VPN. An SSH tunnel is used along with SSH's *port forwarding* capablities. Port forwarding means that an SSH server will automatically take any data sent to a particular port and foward it to (a specified port on) another machine.

There are two types of port forwarding: *local port forwarding* and *remote port forwarding*. With local port forwarding, data sent to some port on your local machine is automatically forwarded/tunneled to a port on a remote host. With remote port forwarding, data sent to some port on a remote host is automatically forwarded/tunneled to a port on the local machine.

There are two basic use cases for tunnels and port fowarding:

1. We want to encrypt network traffic for an insecure protocol. E.g., to securely connect to a POP server over the Internet.
2. We want to connect to a remote machine that is not directly reachable, but we can connect via another machine.

A direct connection to a server may not be possible because of a *firewall*. For example, most servers on SIUC machines are not accessible through SIUC's border firewall. However, because the CS Department's `pc00.cs.siu.edu` SSH server is accessible, it can potentially be used to reach other internal machines. Direct connections might also be prevented by other security mechanisms. For example, a home SSH server might have *hosts access* settings that prevent connections from all but a limited number of IP addresses. If one is on a machine that is not allowed to connect, it might still be possible to *"chain"* connections through another SSH server.

8.1 Local Port Forwarding

The basic command to setup local port fowarding is:

```
ssh -fN -L [bind-address:]localport:remotehost:remoteport ssh-server
```

with these parameters:

- `bind-address` – the local machine addresses to be listened on, typically one would put "localhost" to limit access to this machine only or use * to listen to all local machine IP addresses;
- `localport` – the local machine port one connects to for fowarding;
- `remotehost` – the hostname/ip address of the machine to foward traffic to (this is from the perspective of `ssh-server`, so use "`localhost`" if the destination is `ssh-server`);
- `remoteport` – the port on `remotehost` that traffic is fowarded to;
- `ssh-server` – the hostname/ip address of the remote ssh server.

(Note that if `localport` is less than or equal to 1024, root must run the command in order to create the desired local port.)

Once such an `ssh` command has been executed, processes can connect to the the remote machine `remotehost` at port `remoteport`, by connecting to port `localport` on the local machine. All traffic from the local machine to `ssh-server` will be encrypted. But note that if `remotehost` is not `ssh-server`, then traffic between `ssh-server` and `remotehost` will not be. Thus, that hop should trusted if security is the reason for the tunnel.

Suppose we needed our email client to connect to the POP mail server on remote machine `csmail.cs.siu.edu`, but since POP sends data in clear-text, we need to tunnel/encrypt the connection. If we are able to SSH into `csmail`, we could setup a tunnel to the POP server on `csmail.cs.siu.edu` with this command:

```
ssh -fN -L localhost:110:localhost:110 csmail.cs.siu.edu
```

To get a secure connection to the POP server on `csmail`, we would then set our email client to connect to port 110 *on the local machine* (`localhost`), rather than connecting directly to `csmail`.

Being allowed to SSH into a mail server not likely to be possible, as it is generally not a good idea to allow users SSH access to server machines. In that case, we can instead setup a tunnel to a machine whose connection to `csmail` we trust, and still have our Internet traffic encrypted. A tunnel to the POP server on `csmail.cs.siu.edu` via the machine `pc00.cs.siu.edu` can be setup with this command:

```
ssh -fN -L localhost:110:csmail:110 pc00.cs.siu.edu
```

Once again we would set our email client to connect to port 110 *on the local machine* (`localhost`), rather than connecting directly to `csmail`. In this case, the tunnel would connect to `pc00.cs.siu.edu`, which would automatically forward all traffic to `csmail`. While traffic between `pc00` and `csmail` would *not* be encrypted, if we trusted that the connection between them was not being sniffed, this would still be much better than transmitting our POP credentials in clear text over the Internet.

Suppose that we want to `ssh/scp/sftp` to remote machine `userpc.cs.siu.edu`, that we cannot do it directly, but that we can connect to `userpc` by going via `pc00.cs.siu.edu`. A tunnel to `userpc.cs.siu.edu` via `pc00.cs.siu.edu` can be setup with:

```
ssh -fN -L localhost:2200:userpc:22 pc00.cs.siu.edu
```

Once the tunnel has been created, we can `ssh/scp/sftp` to `userpc` with these commands:

```
ssh -p2200 localhost
scp -P2200 LOCALFILE localhost:REMOTEDIR  (put file)
scp -P2200 localhost:REMOTEFILE LOCALDIR  (get file)
sftp -P2200 localhost
```

8.2 Predefining Tunnels

Tunnels that are used frequently can be defined in the configuration file `~/.ssh/config`. For example, the tunnel to `userpc.cs.siu.edu` via `pc00.cs.siu.edu` could be defined like:

```
Host userpctun
    HostName pc00.cs.siu.edu
    LocalForward localhost:2200 userpc:22
```

The tunnel can then be setup with the call:

```
ssh -fN userpctun
```

8.3 Remote Port Forwarding

Port forwarding can also be used to forward traffic from a remote host to your local host, by using the -R option instead of the -L option. An example command would be:

```
ssh -fN -R 8080:localhost:80 remoteserver
```

This command would will forward all traffic from port 8080 (alternate http port) on the host remoteserver, to port 80 (http/webserver) on the local machine. (Note that you need to be a privileged user on the remote host to open privileged ports.)

8.4 SOCKS: Dynamic Port Forwarding

SOCKS is an Internet protocol that allows a client to connect to a server via another machine called a *proxy server*. OpenSSH is able to perform as a SOCKS proxy server, while again tunneling TCP (or UDP) traffic. This has a similar effect to the port forwarding discussed above, but has the advantage that a SOCKS-compliant client program can dynamically select the ultimate target machine.

To set up a SOCKS proxy, one issues a command like:
ssh -D 1080 sshproxy.example.org!

This creates a secure tunnel to `sshproxy.example.org` and opens port 1080 on the local machine for use as a SOCKS proxy (1080 is the default SOCKS port).

A SOCKS-compliant client program can now use the local machine as a SOCKS proxy, with all traffic securely tunneled to `sshproxy.example.org`, and from there to the true target.

As we discussed earlier, forwarding can be used to get through a local firewall that prevents a direct connection to the target or if the SSH server is local to the target it could provide much improved security.

It is even possible to use non-SOCKS-compliant programs with the aid of a "socksifier program" such as `tsocks`:

```
tsocks thunderbird
```

SOCKS5 can even tunnel DNS requests (UDP).

8.5 SSH Chains

As noted above, it is sometimes necessary to go through a *chain* of SSH servers in order to reach the one you need to do work on. This can occur because of firewall restrictions or access restrictions. For example, while logged into host1, you want to do work on host3, but you have to go through host2 to reach host3. Obviously, you can do this manually by `ssh`'ing from host1 to host2, getting a shell session on host2, then `ssh`'ing to host3. This is a bit tedious if you must do it frequently, and will not work with `scp`/`sftp` (to allow you to move files from host1 to host3).

We saw in Section **??** how to do this with tunnels and local port forwarding, but that approach requires creating the tunnels first. Luckily, there are some methods that can make "SSH chaining" easier to do frequently (and can support chained `scp`/`sftp`). These approaches make forwarding through intervening SSH servers relatively transparent.

If one needs to simply `ssh` to host3 via host2, this can be accomplished with a single command line that takes advantage of `ssh` accepting commands. The command line can be as simple as:

```
ssh -t host2 ssh host3
```

If the usernames are different on each host, use:

```
ssh -t user2@host2 ssh user3@host3
```

The `-t` option is required to create a *"pseudo-tty"* on host2, so that an interactive shell session is started on host3. The approach can be extended to longer chains: simply include `-t` with all but the final `ssh` command.

These commands will work if host2 and host3 can be logged into with key-based authentication, and the required key is stored on the previous host in the chain. If the same username is used on all hosts and the key(s) to login to host2 and host3 are stored only on host1, the `-A` option must be used to enable forwarding of the SSH authentication agent:

```
ssh -tA host2 ssh -A host3
```

If you frequently want to connect from host1 to host3 this way, it would obviously be easy to define an *alias* to reduce typing:

```
alias sshhost3="ssh -t host2 ssh host3"
```

While quite simple, the multiple-ssh-command method does not support `scp`/`sftp`. Luckily, there is another method that allows chaining of `ssh`/`scp`/`sftp`. This alternative approach makes use of SSH's `ProxyCommand`

option and I/O forwarding option:

- `-o ProxyCommand` – Specifies the command to use to connect to the server.
- `-W host:port` – Standard input and output on the client are to be forwarded to host on port.

The basic commands to connect to host3 via host2 are:

```
ssh -o "ProxyCommand ssh -W %h:%p host2" host3
scp -o "ProxyCommand ssh -W %h:%p host2" LOCALFILE host3:REMOTEDIR
scp -o "ProxyCommand ssh -W %h:%p host2" host3:REMOTEFILE LOCALDIR
sftp -o "ProxyCommand ssh -W %h:%p host2" host3
```

While these may look like fairly ugly command lines to remember/type, aliases or functions can be used to simplify what must be typed. In addition, it is possible to define frequently used chains in the `~/.ssh/config`, requiring only simple commands. The host1 to host2 to host3 chain can be defined as:

```
Host h3
  HostName host3  (or can use IP address)
  User USERNAME   (if username is different)
  ProxyCommand ssh -W %h:%p host2
```

Once `h3` has been defined, we simply need do:

```
ssh h3
sftp h3
```

etc.

The `ProxyCommand` option can potentially be used even with an intermediary that isn't an SSH server. This will require the availability of *Netcat* (`nc`). For example, the following directive would use Netcat's proxy support to connect to host3 via an HTTP proxy at host2:

```
ssh -o "ProxyCommand nc -X connect -x host2:8080 %h %p" host3
```

9 Other Uses of SSH

9.1 SSHFS

SSHFS (SSH FileSystem) allows directories on remote SSH server machines to be *mounted* on the local machine. SSHFS uses the SFTP protocol to transmit files between the machines, so all information is kept secure. This is in contrast to other file sharing protocols such as SMB/CIFS or NFS, where files are transmitted in plain text, making these protocols unsuitable for file sharing across the Internet (SMB 3 supports encryption). SSHFS is available for Linux, Mac OS X, and even Windows (via Dokan).

9.2 rsync over SSH

`rsync` is a key tool for remote backups on Linux/UNIX systems. It can efficiently synchronize directories on different machines because it transmits only the differences between files. While `rsync` transmits information as clear text, if the remote machine is running an SSH server, `rsync` can tunnel its connection. Modern versions of `rsync` will try to use SSH by default (so don't require remote machine be running `rsyncd`). Automating backups via `rsync` will require that key-based user authentication be enabled.