Table of Contents

- Job control
  - Pausing and backgrounding processes
- Terminal Multiplexers
  - Sessions
  - Windows
  - Panes
  - References
- Aliases
- Dotfiles
- Remote machines
  - SSH Keys
  - Key generation
  - Key based authentication
  - Copying files over SSH
- Port Forwarding
- SSH Configuration
- Miscellaneous
- Job control
- Terminal multiplexer
- Aliases
- Dotfiles
- Remote Machines
- Job Control
-

Job control

- there are a variety of software interrupts or signals which can effect execution
- `Ctrl+c` sends `SIGINT` signal
- See `man signal` for reference
- `kill`: sends signals to a process; default is TERM
- `SIGINT`: `^C`; interrupt program; terminate process

- `SIGQUIT`: `^\`; quit program
- `SIGKILL`: terminate process; kill program; cannot be captured by process and will always terminate immediately
  - can result in orphaned child processes
- `SIGSTOP`: pause a process
  - `SIGTSTP`: `^Z`; terminal stop
- `SIGHUP`: terminal line hangup; terminate process
- `SIGTERM`: signal requesting graceful process exit
  - To send this signal: `kill -TERM <pid>`

Pausing and backgrounding processes

- `Ctrl-Z` to pause process
- `fg`/`bg` to continue the process in forground/background
- `jobs`: lists unfinished jobs associated with current terminal session
  - `pgrep`: look up processes based on name/attributes, gives you the process id which you can use to refer to a job
  - `%<job-number>` also allows you to refer to a job
  - `$!`: last backgrounded job
- & suffix: runs command in background, but still uses `STDOUT`
- To background an already running process: `^Z`, `bg`
- background processes are still children processes of your terminal
- if you close the terminal, `SIGHUP` will be sent and the process will die
- to stop this run the the program with `nohup` (a wrapper to ignore `SIGHUP`), or `disown` if the process has already started
- alternatively use a terminal multiplexer

Sample session

```
$ sleep 1000
^Z
[1]  + 18653 suspended  sleep 1000
```

```
4
5  $ nohup sleep 2000 &
6  [2] 18745
7  appending output to nohup.out
8
9  $ jobs
10 [1]  + suspended  sleep 1000
11 [2]  - running    nohup sleep 2000
12
13 $ bg %1
14 [1]  - 18653 continued  sleep 1000
15
16 $ jobs
17 [1]  - running    sleep 1000
18 [2]  + running    nohup sleep 2000
19
20 $ kill -STOP %1
21 [1]  + 18653 suspended (signal)  sleep 1000
22
23 $ jobs
24 [1]  + suspended (signal)  sleep 1000
25 [2]  - running    nohup sleep 2000
26
27 $ kill -SIGHUP %1
28 [1]  + 18653 hangup     sleep 1000
29
30 $ jobs
31 [2]  + running    nohup sleep 2000
32
33 $ kill -SIGHUP %2
34
35 $ jobs
36 [2]  + running    nohup sleep 2000
37
38 $ kill %2
39 [2]  + 18745 terminated  nohup sleep 2000
40
41 $ jobs
```

Terminal Multiplexers

- terminal multiplexers e.g. `tmux` let you run multiple things at once without having to open separate terminal windows
  - panes/tabs to interact with multiple shell sessions
  - you can detach from a current terminal session and reattach at a later time
  - means you don't have to use `nohup` etc.

- tmux keybindings have form <C-b> x i.e. press Ctrl+b release and press ‘x

tmux has the following hierarchy of sessions, windows, and objects

Sessions

- session: independent workspace with 1+ windows
- tmux: starts a new session
- tmux ls: lists current sessions
- tmux **new** -s NAME starts a new session with specified name
- <C-b> d detaches current session (from within tmux)
- tmux a attaches last session; -t flag specifies which session

Windows

- window: visually separate part of same session; equivalent to tabs in editors/browsers
- <C-b> c creates new window
- <C-d> terminate shell to close a window
- <C-b> N go to Nth window
- <C-b> p go to previous window
- <C-b> n go to next window
- <C-b> , rename current window
- <C-b> w list current windows

Panes

- pane: multiple shells in the same visual display; like vim splits
- <C-b> " split current pane horizontally |
- <C-b> % split current pane vertically –
- <C-b> <dirn> move to pane in specified direction (arrow key)
- <C-b> z toggle zoom for current pane
- <C-b> [start scrollback. press space to start selection, enter to copy the selection
- <C-b> <space> cycle through pane arrangements

References

- quick and easy guide to tmux

- terminal multiplexers
- screen comes installed in most UNIX systems

Aliases

- short form of another command that shell automatically replaces
- a `bash` alias has the following structure, defined in the .bashrc/.zshrc:

```
alias alias_name="command_to_alias arg1 arg"
```

Aliases

```
# Save a lot of typing for common commands
alias gs="git status"
alias gc="git commit"
alias v="vim"

# Save you from mistyping
alias sl=ls

# Overwrite existing commands for better defaults
alias mv="mv -i"           # -i prompts before overwrite
alias mkdir="mkdir -p"     # -p make parent dirs as needed
alias df="df -h"           # -h prints human readable format

# Alias can be composed
alias la="ls -A"
alias lla="la -l"

# To ignore an alias run it prepended with \
\ls
# Or disable an alias altogether with unalias
unalias la

# To get an alias definition just call it with alias
alias ll
# Will print ll='ls -lh'
```

Dotfiles

- shell startup scripts
- guide to dotfiles on github
- example popular dotfiles

Some other examples of tools that can be configured through dotfiles are: - bash - ~/.bashrc,

~/.bash_profile - git - ~/.gitconfig - vim - ~/.vimrc and the ~/.vim folder - ssh - ~/.ssh/config - tmux - ~/.tmux.conf

Common edit is to add a program to the path environment variable:

```
export PATH="$PATH:/path/to/program/bin"
```

Best way to manage your dotfiles is to store them in their own folder, symlink to them, and version control the folder. Benefits: - portability: work the same way on different machines - easy installation on new machines - synchronisation - change tracking

Use if statements to vary config for machine specific customisations

```
if [[ "$(uname)" == "Linux" ]]; then {do_something}; fi

# Check before using shell-specific features
if [[ "$SHELL" == "zsh" ]]; then {do_something}; fi

# You can also make it machine-specific
if [[ "$(hostname)" == "myServer" ]]; then {do_something}; fi
```

If the configuration file supports it, make use of includes to store machine specific settings which could be tracked in separate repositories. For example, a ~/.gitconfig can have a setting:

```
[include]
    path = ~/.gitconfig_local
```

if you want different programs to share some configurations. For instance, if you want both bash and zsh to share the same set of aliases you can write them under .aliases and have the following block in both:

```
# Test if ~/.aliases exists and source it
if [ -f ~/.aliases ]; then
    source ~/.aliases
fi
```

Remote machines

Secure shell (ssh)

To login as `foo` at `bar.mit.edu` server:

```
ssh foo@bar.mit.edu
```

- nb you can execute commands directly: `ssh foo@server ls` executes `ls` on home directory of `foo`. this could be piped into `grep` for example ### SSH Keys

- key-based authentication uses public-key cryptography to prove to the server the client owns the private key without revealing it.
  - means you do not need to re-enter password every time
  - private key (often ~/.ssh/id_rsa or ~/.ssh/id_ed25519) should be treated like a password

Key generation

- generate a pair by running ssh-keygen: bash ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
- choose a passphrase; use ssh-agent/gpg-agent so you don't have to type it each time
- configure pushing to github using ssh keys
- check if you have a passphrase & validate with ssh-keygen -y -f /path/to/key

Key based authentication

ssh will look into .ssh/authorized_keys to determine which clients it should let in. To copy a public key over you can use:

```
cat .ssh/id_ed25519.pub | ssh foobar@remote 'cat >> ~/.ssh/authorized_keys'
```

A simpler solution can be achieved with ssh-copy-id where available:

```
ssh-copy-id -i .ssh/id_ed25519.pub foobar@remote
```

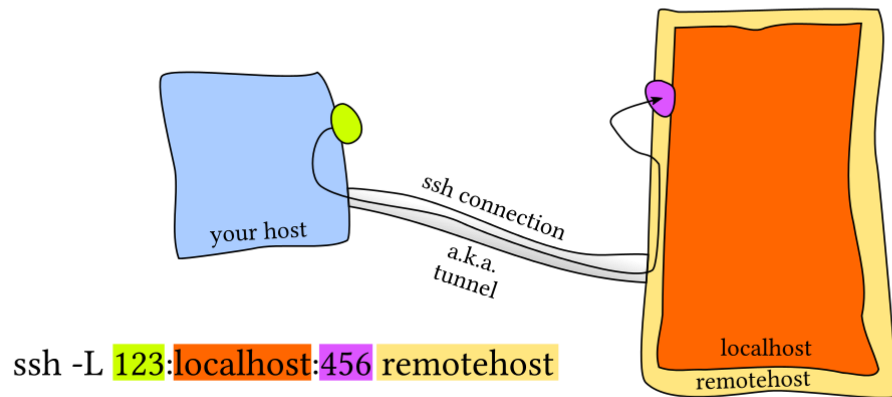
Copying files over SSH

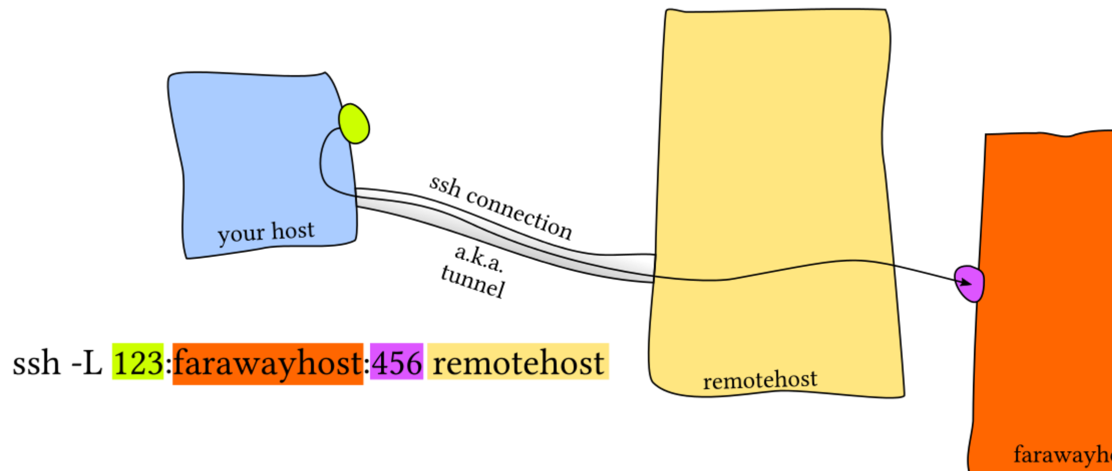
- ssh+tee, the simplest is to use ssh command execution and STDIN input by doing cat localfile | ssh remote_server tee serverfile. Recall that tee writes the output from STDIN into a file.
- scp when copying large amounts of files/directories, the secure copy scp command is more convenient since it can easily recurse over paths. The syntax is scp path/to/local_file remote_host:path/to/remote_file
- rsync improves upon scp by detecting identical files in local and remote, and preventing copying them again. It also provides more fine grained control over symlinks, permissions and has extra features like the --partial flag that can resume from a previously interrupted copy. rsync has a similar syntax to scp.

Port Forwarding

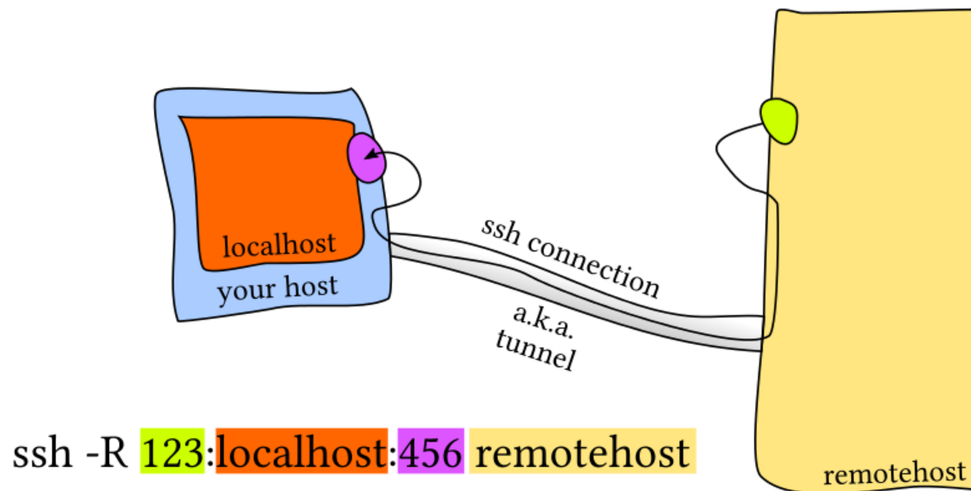
In many scenarios you will run into software that listens to specific ports in the machine. When this happens in your local machine you can type `localhost:PORT` or `127.0.0.1:PORT`, but what do you do with a remote server that does not have its ports directly available through the network/internet?.

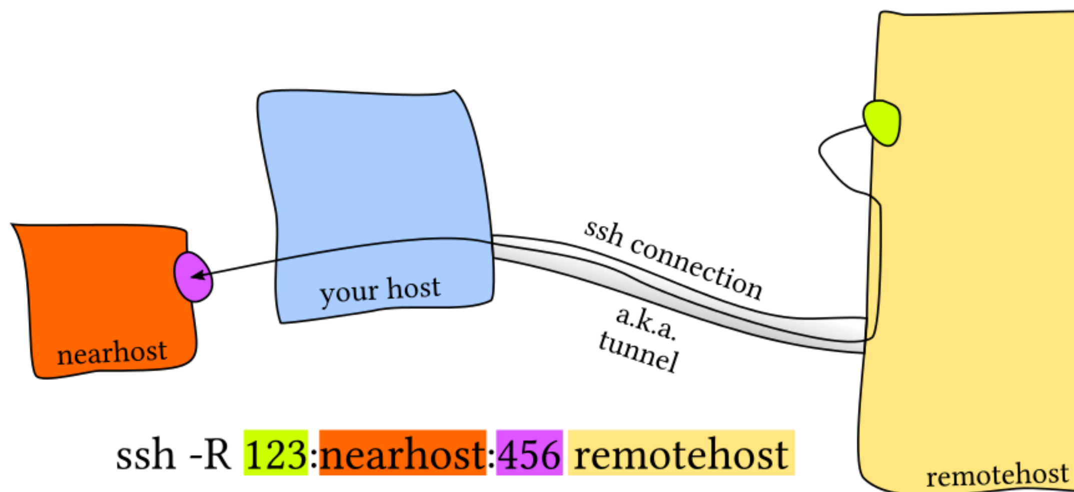
This is called *port forwarding* and it comes in two flavors: Local Port Forwarding and Remote Port Forwarding (see the pictures for more details, credit of the pictures from this StackOverflow post).





Local Port Forwarding





Remote Port Forwarding

The most common scenario is local port forwarding, where a service in the remote machine listens in a port and you want to link a port in your local machine to forward to the remote port. For example, if we execute `jupyter notebook` in the remote server that listens to the port `8888`. Thus, to forward that to the local port `9999`, we would do `ssh -L 9999:localhost:8888 foobar@remote_server` and then navigate to `locahost:9999` in our local machine.

SSH Configuration

We have covered many many arguments that we can pass. A tempting alternative is to create shell aliases that look like

```
alias my_server="ssh -i ~/.id_ed25519 --port 2222 - L 9999:localhost
    :8888 foobar@remote_server
```

However, there is a better alternative using `~/.ssh/config`.

```
Host vm
    User foobar
    HostName 172.16.174.141
    Port 2222
    IdentityFile ~/.ssh/id_ed25519
    RemoteForward 9999 localhost:8888

# Configs can also take wildcards
Host *.mit.edu
    User foobaz
```

An additional advantage of using the `~/.ssh/config` file over aliases is that other programs like `scp`, `rsync`, `mosh`, &c are able to read it as well and convert the settings into the corresponding flags.

Note that the `~/.ssh/config` file can be considered a dotfile, and in general it is fine for it to be included with the rest of your dotfiles. However, if you make it public, think about the information that you are potentially providing strangers on the internet: addresses of your servers, users, open ports, &c. This may facilitate some types of attacks so be thoughtful about sharing your SSH configuration.

Server side configuration is usually specified in `/etc/ssh/sshd_config`. Here you can make changes like disabling password authentication, changing ssh ports, enabling X11 forwarding, &c. You can specify config settings in a per user basis.

Miscellaneous

- remote server disconnects, bad connections are common pain points: Mosh, the mobile shell, improves upon ssh, allowing roaming connections, intermittent connectivity and providing intelligent local echo.
- mount a remote folder with sshfs

Shells & Frameworks

Alternatives to `bash`: e.g. the `zsh` shell is a superset of `bash` and provides many convenient features out of the box such as:

- Smarter globbing, `**`
- Inline globbing/wildcard expansion
- Spelling correction
- Better tab completion/selection
- Path expansion (`cd /u/lo/b` will expand as `/usr/local/bin`)

Frameworks can improve your shell as well. Some popular general frameworks are prezto or oh-my-zsh, and smaller ones that focus on specific features such as zsh-syntax-highlighting or zsh-history-substring-search. Shells like fish include many of these user-friendly features by default. Some of these features include:

- Right prompt
- Command syntax highlighting
- History substring search
- manpage based flag completions
- Smarter autocompletion
- Prompt themes

NB frameworks may slow down your shell, especially if the code they run is not properly optimized or it is too much code. If an issue: profile it and disable the features that you do not use often or value over speed.

Terminal Emulators

Along with customizing your shell, it is worth spending some time figuring out your choice of **terminal emulator** and its settings. There are many terminal emulators out there (here is a comparison).

Since you might be spending hundreds to thousands of hours in your terminal it pays off to look into its settings. Some of the aspects that you may want to modify in your terminal include:

- Font choice
- Color Scheme
- Keyboard shortcuts
- Tab/Pane support
- Scrollback configuration

- Performance (some newer terminals like Alacritty or kitty offer GPU acceleration).

TeX and pandoc

- `pandoc` converts between all sorts of file formats
- `fc-list : family | sort > font_list.txt` outputs list of files
- Customising pandoc
- Syntax highlighting
- Eisvogel notes template

Exercises

Job control

1. From what we have seen, we can use some `ps aux | grep` commands to get our jobs' pids and then kill them, but there are better ways to do it. Start a `sleep 10000` job in a terminal, background it with `Ctrl-Z` and continue its execution with `bg`. Now use `pgrep` to find its pid and `pkill` to kill it without ever typing the pid itself. (Hint: use the `-af` flags).

2. Say you don't want to start a process until another completes, how you would go about it? In this exercise our limiting process will always be `sleep 60 &`. One way to achieve this is to use the `wait` command. Try launching the sleep command and having an `ls` wait until the background process finishes.

   However, this strategy will fail if we start in a different bash session, since `wait` only works for child processes. One feature we did not discuss in the notes is that the `kill` command's exit status will be zero on success and nonzero otherwise. `kill -0` does not send a signal but will give a nonzero exit status if the process does not exist. Write a bash function called `pidwait` that takes a pid and waits until said process completes. You should use `sleep` to avoid wasting CPU unnecessarily.

Terminal multiplexer

1. Follow this `tmux` tutorial and then learn how to do some basic customizations following these steps.

Aliases

1. Create an alias `dc` that resolves to `cd` for when you type it wrongly.
2. Run `history | awk '{$1="";print substr($0,2)}' | sort | uniq -c | sort -n | tail -n 10` to get your top 10 most used commands and consider writing shorter aliases for them. Note: this works for Bash; if you're using ZSH, use `history 1` instead of just `history`.

Dotfiles

Let's get you up to speed with dotfiles. 1. Create a folder for your dotfiles and set up version control. 1. Add a configuration for at least one program, e.g. your shell, with some customization (to start off, it can be something as simple as customizing your shell prompt by setting `$PS1`). 1. Set up a method to install your dotfiles quickly (and without manual effort) on a new machine. This can be as simple as a shell script that calls `ln -s` for each file, or you could use a specialized utility. 1. Test your installation script on a fresh virtual machine. 1. Migrate all of your current tool configurations to your dotfiles repository. 1. Publish your dotfiles on GitHub.

Remote Machines

Install a Linux virtual machine (or use an already existing one) for this exercise. If you are not familiar with virtual machines check out this tutorial for installing one.

1. Go to `~/.ssh/` and check if you have a pair of SSH keys there. If not, generate them with `ssh-keygen -o -a 100 -t ed25519`. It is recommended that you use a password and use `ssh-agent` , more info here.
2. Edit `.ssh/config` to have an entry as follows

```
Host vm
    User username_goes_here
    HostName ip_goes_here
    IdentityFile ~/.ssh/id_ed25519
    RemoteForward 9999 localhost:8888
```

1. Use `ssh-copy-id vm` to copy your ssh key to the server.
2. Start a webserver in your VM by executing `python -m http.server 8888`. Access the VM webserver by navigating to `http://localhost:9999` in your machine.
3. Edit your SSH server config by doing `sudo vim /etc/ssh/sshd_config` and disable password authentication by editing the value of `PasswordAuthentication`. Disable root login by editing the value of `PermitRootLogin`. Restart the `ssh` service with `sudo service sshd restart`. Try sshing in again.

4. (Challenge) Install `mosh` in the VM and establish a connection. Then disconnect the network adapter of the server/VM. Can mosh properly recover from it?
5. (Challenge) Look into what the `-N` and `-f` flags do in `ssh` and figure out what a command to achieve background port forwarding.

Solutions

Job Control

1.

```
$ sleep 10000
^-Z
[1]  + 12388 suspended  sleep 10000
$ bg %1
[1]  + 12388 continued  sleep 10000
$ pgrep -af sleep
12388 sleep 10000
$ pkill -f sleep
[1]  + 12388 terminated  sleep 10000
```

1.

```
$ sleep 30 &
[1] 12606
$ pgrep sleep | wait; ls
```

Bash function `pidwait`:

```
pidwait() {
  wait $1
  return 0
}
$ sleep 30 &
[1] 12721
$ pgrep sleep | pidwait
[1]  + 12721 done       sleep 30
[2]  - 12736 done       pgrep sleep
```