15 – Networking and Package Management

CS 2043: Unix Tools and Scripting, Spring 2019 [1]

Matthew Milano
February 27, 2019

Cornell University
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>welcome back to THE INTERNET</td>
</tr>
<tr>
<td>2.</td>
<td>Package Management</td>
</tr>
<tr>
<td>3.</td>
<td>System Specific Package Managers</td>
</tr>
<tr>
<td>4.</td>
<td>Other Managers</td>
</tr>
</tbody>
</table>
welcome back to THE INTERNET
Command we forgot from last time

**ping** a packet off a remote host

**ping** [flags...] **<host>**

- Simple echo back-and-forth
- tests connections
- uses **ICMP** protocol – same as **traceroute**
- runs forever by default

```
$ ping -c 4 google.com
PING google.com (172.217.9.238) 56(84) bytes of data.
64 bytes from lga34s11-in-f14.1e100.net (172.217.9.238): icmp_seq=1 ttl=55 time=8.24 ms
64 bytes from lga34s11-in-f14.1e100.net (172.217.9.238): icmp_seq=2 ttl=55 time=8.51 ms
64 bytes from lga34s11-in-f14.1e100.net (172.217.9.238): icmp_seq=3 ttl=55 time=8.56 ms
64 bytes from lga34s11-in-f14.1e100.net (172.217.9.238): icmp_seq=4 ttl=55 time=8.56 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 8ms
rtt min/avg/max/mdev = 8.237/8.468/8.563/0.163 ms
```
Last time

- Computers communicate by sending **packets** through the network
- Packets are addressed to a local **MAC** and a potentially-remote **IP**
- **Switches** connect computers into a **local network** and forward packets by **MAC**
- **Routers** connect local networks into an **intranet** and forward packets by **IP**
Protocols from last time

- The **DHCP** protocol gives computers an IP address
- The **ARP** protocol associates an IP address with a MAC address
- The **DNS** protocol associates a domain name (google.com) with a MAC address
What is a protocol?

• an agreement on what sort of packets to exchange to achieve a particular goal
• Can be multi-step
• we distinguish between transport layer and application layer
More about protocols: transport layer

- *transport-layer* protocols correspond to different “kinds” of packets
  - examples: ARP, ICMP
- Operating system sees the different packets, handles them accordingly
- **operating system’s job** to handle transport-layer packets
More about protocols: application layer

• *application-layer* protocols use the *same* kind of packet
  • examples: DHCP, DNS, HTTPS, SSH, most others you know
• Operating system passes them to applications
• How do applications find their packets?
Introducing: TCP and UDP

• *transport-layer* protocols for communicating with applications
  • differentiate applications with “ports”
    • just a 16-bit integer
    • like apartment numbers
  • applications listen at a specific port
    • registers with the OS
    • OS only forwards port-destined traffic
• contains “return addresses” for easy reply to client
TCP

- Most popular transport protocol
  - examples: HTTP, SSH
- *connection-oriented* protocol
  - “connect” to a port on a remote stream
  - receive a private channel on which to keep communicating
  - like a phone call ... or SSH session
- Hides common failures
  - ensures packets are reasonably ordered
  - retransmits packets if they get lost
  - cool algorithm to avoid congestion
UDP

- Second-most popular transport protocol
  - examples: DHCP, DNS, VoIP, Steam (as in video games), internet radio
  - **not** Netflix
- **only gives you the port**
  - no connection: works like physical mail.
- All common failures exposed to application
  - packet order may vary
  - packets may not arrive
  - no indication whether transmitted packet got there
- Mostly used in either very-old, high-assurance or real-time applications
- more resilient to DOS attacks than TCP
Application protocols

- Still defines pattern of communication
- Specific messages expected at specific times
- Messages sent via (usually) TCP/UDP
- Example: HTTP, SSH, etc.
**netcat**: so much more than *cat* over the *network*

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nc [flags] [host]</td>
<td>Raw TCP protocol tool</td>
</tr>
<tr>
<td>nc -l -p &lt;port&gt;</td>
<td>sends stdin over the network</td>
</tr>
<tr>
<td>nc &lt;host&gt; &lt;port&gt;</td>
<td>receives stdout from the network</td>
</tr>
</tbody>
</table>

- nc -l “listens”, behaves like a server
- nc <host> “connects”, behaves like a client
HTTP: a protocol to explore

- HTTP messages are raw text!
- Strings sent via TCP to port 80
- GET request: access a page

```
GET /people/mpmilano/ HTTP/1.1
Host: cs.brown.edu
```

- Let’s send this via **netcat**! (demo)
- Can explore more protocols this way; try it!
Some common ports

• HTTP: TCP/80
• SSH: TCP/22
• FTP: TCP/20 and TCP/21
• HTTPS: TCP/443
• SMTP (mail): TCP/25
Firewalls

- In a perfect world, we wouldn’t need a firewall.
- Lives in the network, or in the kernel
- Inspects traffic *before* it reaches its destination
- Two primary uses: filter legitimate services, block unwanted ones
Firewalls: the good uses

• Legit: *Filters* certain ports to prevent regions of the internet from accessing them
  • Cornell firewall drops all traffic destined to on-campus servers originating from off-campus IPs
  • *wash* firewall does the same
  • mail relay firewall would only allow known senders to connect

• prevents server from being overloaded by random external griefers
• prevents aggressive server scans from the darkweb
  • which, by the way, exists. ask me later.
Firewalls: the lazy uses.

- Block insecure / old apps
- cover up for weird/bad OS/system design
  - Example: print server on a mac at port 631
  - Example: just a lot of windows
- Block all uninvited remote connections
  - if your laptop isn’t a server, shouldn’t have exposed ports
  - if it does have exposed ports, some application is doing a bad.
- Fundamentally lazy: right answer is to secure the applications, not hide them.
- lots of legacy apps (that we’re stuck with) can’t be fixed, so also fundamentally necessary
Package Management
• If I had to give only one reason why Unix systems are superior to Windows: Package Management.
• Can install almost anything with ease of from your terminal.
• Update to the latest version with one command.
  • No more download the latest installer nonsense!
• Various tools can be installed by installing a package.
  • A package contains the files and other instructions to setup a piece of software.
  • Many packages depend on each other.
  • High-level package managers download packages, figure out the dependencies for you, and deal with groups of packages.
  • Low-level managers unpack individual packages, run scripts, and get the software installed correctly.
• In general, these are “pre-compiled binaries”: no compilation necessary. It’s already packaged nice and neat just for you!
Package Managers in the Wild

- GNU/Linux:
  - Low-level: two general families of packages exist: deb, and rpm.
  - High-level package managers you are likely to encounter:
    - Debian/Ubuntu: **apt-get**.
    - Some claim that **aptitude** is superior, but I will only cover **apt-get**. They are roughly interchangeable.
    - SUSE/OpenSUSE: **zypper**.
    - Fedora: **dnf** (Fedora 22+).
    - **zypper** and **dnf** use SAT-based dependency solvers, which many argue is fundamentally superior. The dependency resolution phase is usually not the slowest part though...installing the packages is. See [3] for more info.
    - RHEL/CentOS: **yum** (until they adopt **dnf**).

- Mac OSX:
  - Others exist, but the only one you should ever use is **brew**.
  - Don’t user others (e.g. **port**), they are outdated / EOSL.
• Though the syntax for each package manager is different, the concepts are all the same.
  • This lecture will focus on **apt-get**, **dnf**, and **brew**.
  • The **dnf** commands are almost entirely interchangeable with **yum**, by design.
  • Note that **brew** is a “special snowflake”, more on this later.

• What does your package manager give you? The ability to
  • **install** new packages you do not have.
  • **remove** packages you have installed.
  • **update** installed packages.
  • update the lists to search for files / updates from.
  • **view dependencies** of a given package.
  • a whole lot more!!!
A Note on **update**

- The **update** command has importantly different meanings in different package managers.
- Some **do**, and some **do not** default to system (read linux kernel) updates.
  - Ubuntu: default is **no**.
  - Fedora: default is **yes**.
  - RHEL: default is **no**.
- It depends on your operating system, and package manager.
  - Know your operating system, and look up what the default behavior is.
- If your program needs a specific version of the linux kernel, you need to be very careful!
A Note on Names and their Meanings

• You may see packages of the form:
  • `<package>.i[^{3456}]86` (e.g. `.i386` or `.i686`):
    • These are the 32-bit packages.
  • `<package>.x86_64`: these are the 64-bit packages.
  • `<package>.noarch`: these are independent of the architecture.

• Development tools can have as many as three packages:
  • The header files are usually called something like:
    • `deb`: usually `<package>-dev`
    • `rpm`: usually `<package>-devel`
  • The library you will need to link against:
    • If applicable, `lib<package>` or something similar.
  • The binaries (executables), often provided by just `<package>`.
    • Most relevant for C and C++, but also Python and others.
  • Use the `search` functionality of your package manager.
Example Development Tool Installation

- If I needed to compile and link against **Xrandr** (X.Org X11 libXrandr runtime library) on Fedora, I would have to install
  - **libXrandr**: the library.
  - **libXrandr-devel**: the header files.
  - Not including `.x86_64` is OK / encouraged, your package manager knows which one to install.
  - Though in certain special cases you may need to get the **32-bit** library as well.
    - In this case, if I were compiling a program that links against **libXrandr**, but I want to release a pre-compiled 32bit library, it must be installed in order for me to link against it.

- The **deb** versions should be similarly named, but just use the **search** functionality of find the right names.
- This concept has no meaning for **brew**, since it compiles everything.
System Specific Package Managers
Debian / Ubuntu Package Management (apt-get)

- Installing and uninstalling:
  - Install a package:
    `apt-get install <pkg1> <pkg2> ... <pkgN>`
  - Remove a package:
    `apt-get remove <pkg1> <pkg2> ... <pkgN>`
  - Only one pkg required, but can specify many.
  - “Group” packages are available, but still the same command.

- Updating components:
  - Update lists of packages available: `apt-get update`.
    - No arguments, it updates the whole list (even if you give args).
  - Updating currently installed packages: `apt-get upgrade`.
    - Specify a package name to only update / upgrade that package.
  - Update core (incl. kernel): `apt-get dist-upgrade`.

- Searching for packages:
  - Different command: `apt-cache search <pkg>`
RHEL / Fedora Package Managers (**yum** and **dnf**)

- Installing and uninstalling:
  - Install a package:
    ```bash
    dnf install <pkg1> <pkg2> ... <pkgN>
    ```
  - Remove a package:
    ```bash
    dnf remove <pkg1> <pkg2> ... <pkgN>
    ```
  - Only one **pkg** required, but can specify many.
  - “Group” packages are available, but different command:
    ```bash
    dnf groupinstall 'Package Group Name'
    ```
- Updating components:
  - Update EVERYTHING: **dnf upgrade**.
  - **update** exists, but is essentially **upgrade**.
    - Specify a **package** name to only upgrade that package.
  - Updating repository lists: **dnf check-update**
- Searching for packages:
  - Same command: **dnf search <pkg>**
- **yum** and **dnf** (**Dandified Yum**) nearly interchangeable: [3].
**dnf: Cautionary Tales**

- **WARNING**: if you install package Y, which installs X as a dependency, and later **remove** Y
  - By default, X will be removed!
  - Generally, won’t know you needed to **mark** until it is too late.

- **Solution?**
  - Basically, **pay attention to your package manager**.
  - It gets removed because nothing *explicitly* depends on it.
  - So one day you may realize “OH NO! I’m missing package X”...
  - ...so just **dnf install X**.
    - So while **mark** is available, personally I don’t use it.
  - Sad face, I know. Just the way of the world.
OSX Package Management: Install **brew** on your own

- Sitting in class right now with a Mac?
- **DON’T DO THIS IN CLASS.** You will want to make sure you do not have to interrupt the process.
  - Make sure you have the “Command Line Tools” installed.
    - Instructions are on the First Things First Config Page
  - Visit [http://brew.sh/](http://brew.sh/)
    - Copy-paste the given instructions in the terminal as a regular user (**not root**).

- **VERY IMPORTANT:** READ WHAT THE OUTPUT IS!!!! It will tell you to do things, and you **have** to do them. Specifically
  
  You should run **'brew doctor'** BEFORE you install anything.
OSX Package Management (brew)

• Installing and uninstalling:
  • Install a formula:
    `brew install <fmla1> <fmla2> ... <fmla2>`
  • Remove a formula:
    `brew uninstall <fmla1> <fmla2> ... <fmlaN>`
  • Only one `fmla` required, but can specify many.
  • “Group” packages have no meaning in `brew`.

• Updating components:
  • Update `brew`, all `taps`, and installed formulae listings. This does not update the actual software you have installed with `brew`, just the definitions: `brew update`.
  • Update just installed formulae: `brew upgrade`.
    • Specify a `formula` name to only upgrade that formula.

• Searching for packages:
  • Same command: `brew search <formula>`
• Safe: confines itself (by default) in `/usr/local/Cellar`:
  • No `sudo`, plays nicely with OSX (e.g. Applications, `python3`).
  • Non-linking by default. If a conflict is detected, it will tell you.
  • **Really important to read what `brew` tells you!!!**

• `brew` is modular. Additional repositories (“`taps`”) available:
  • Essentially what a `.rpm` or `.deb` would give you in Linux.
  • These are 3rd party repos, not officially sanctioned by `brew`.

• Common taps people use:
  • `brew tap homebrew/science`
    • Various “scientific computing” tools, e.g. `opencv`.
  • `brew tap caskroom/cask`
    • Install `.app` applications! Safe: installs in the “Cellar”, symlinks to `~/Applications`, but **now these update with brew all on their own when you brew update!**
    • E.g. `brew cask install vlc`
OSX: One of These Kids is Not Like the Others (Part II)

- **brew** installs *formulas*.
  - A **ruby** script that provides rules for where to download something from / how to compile it.
- Sometimes the packager creates a “**Bottle**”:
  - If a bottle for your version of OSX exists, you don’t have to compile locally.
  - The bottle just gets *downloaded* and then “*poured***”.
- Otherwise, **brew** downloads the source and compiles locally.
- Though more time consuming, can be quite convenient!
  - **brew options opencv**
  - **brew install --with-cuda --c++11 opencv**
  - It really really really is magical. No need to understand the **opencv** build flags, because the authors of the **brew** formula are kind and wonderful people.
  - **brew reinstall --with-missed-option formula**
• Reiteration: **pay attention to brew** and what it says. Seriously.
• Example: after installing **opencv**, it tells me:

```bash
mkdir -p /Users/sven/.local/lib/python2.7/site-packages
echo 'import site; site.addsitedir( 
"/usr/local/lib/python2.7/site-packages")' >> 
/Users/sven/.local/lib/python2.7/site-packages/homebrew.pth
```

• **brew** gives copy-paste format, above is just so you can read.
• I want to use **opencv** in **Python**, so I do what **brew** tells me.
Sometimes when dependencies are installed behind the scenes, and you no longer need them, you will want to get rid of them.

- `apt-get autoremove`
- `dnf autoremove`
- `brew doctor`

View the list of repositories being checked:

- `apt-cache policy` (well, sort of… `apt` doesn’t have it)
- `dnf repolist [enabled|disabled|all]`

  - Some repositories for `dnf` are `disabled` by default (with good reason). Usually you want to just

    `dnf --enablerepo=<name> install <thing>`

    e.g. if you have `rawhide` (development branch for fedora).

- `brew tap`
Other Managers
Like What?

• There are so many package managers out there for different things, too many to list them all!
  • Ruby: gem
  • Anaconda Python: conda
  • Python: pip
  • Python: easy_install (but really, just use pip)
  • Python3: pip3
  • LaTeX: tlmgr (uses the CTAN database)
    • Must install TeX from source to get tlmgr
  • Perl: cpan
  • Sublime Text: Package Control
  • Many many others...
Some notes and warnings about Python package management.

Notes:

- If you want X in Python 2 and 3:
  - `pip install X and pip3 install X`
- OSX Specifically: advise only using `brew` or Anaconda Python. The system Python can get really damaged if you modify it, you are better off leaving it alone.
- So even if you want to use `python2` on Mac, I strongly encourage you to install it with `brew`.

Warnings:

- Don’t mix `easy_install` and `pip`. Choose one, stick with it.
  - But the internet told me if I want `pip` on Mac, I should `easy_install pip`.
  - NO! Because this `pip` will modify your system python, USE BREW.
- Don’t mix `pip` with `conda`. If you have Anaconda python, just stick to using `conda`. 
[1] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.
