Lecture 22: Dependency management

CS 5150, Spring 2022

Administrative reminders

- In-class test this Thursday (April 21)
  - If medical/religious conflict, must notify instructor *before* exam

- Final project delivery in 3 weeks

- Complete peer evaluations for session 4

Questions on old material?

Quick review

- What is the *critical path* in an activity graph?
- What distinguishes *incremental delivery* from *iterative refinement*?
- What are some properties of good requirements?
- How is a *virtual machine* different from a *container*?
- When would you employ the *Builder* pattern?

Lecture goals

- Manage application dependencies and associated risks

Dependencies

Internal vs. external dependencies

Internal

- Maintainers' goals are (hopefully) aligned
- Can audit for all uses of a library
- Can coordinate large-scale changes of all code using library (facilitated by monorepo)
- Can manage with source control tools, policies

External

- Cannot assume coordination between library and users
- Cannot enforce compatibility, maintenance policies
- Cannot control release schedule
- Danger of diamond dependency problem
- Domain of dependency management

Why depend on external code?

Pros

- **Increase productivity**
- Benefit from higher quality
  - External expertise
  - Incorporate experience from diverse users
- Outsource maintenance burden

Cons

- Dependence on code outside of your control
  - Do you have the resources to audit it?
- Potential for dependency bloat
- Potential for incompatibilities
- Supply chain vulnerabilities

Where to get dependencies from?

- Defer to users / distributors
  - E.g. List of Debian packages to install
  - Common for libraries, system software (C/C++); often used for "standard" dependencies
  - Build system should confirm that dependencies are satisfied
  - May assume elevated privileges, may mask portability
- "Vendoring"
  - Copy third party source code (or artifacts) into your repository

- Artifact repositories
  - Download binary artifacts and their transitive dependencies
  - E.g. Maven Central, Python Wheels, Debian packages
- Source code repositories
  - Download source code and compile locally
  - E.g. Cargo.io, BSD ports, npm

Repository mirrors

- Depending on public repositories is risky
    - What if their servers are not available?
    - What if packages are removed?
    - Do you trust that an artifact will never change?
    - Does your employer's firewall block binaries? Do they need to scan for viruses?

- Can point build tools to an internal repository mirror, rather than the public Internet
    - Tradeoff between maintenance and control

Dependency networks

- Dependencies have their own transitive dependencies
  - Demo: `sbt dependencyTree`

- Assignment (next week): analyze dependency tree for a real application

Diamond dependency problem

- Consider an application that uses a computer vision library and a GUI toolkit
- Suppose the CV library depends on libpng-1.4, but the GUI toolkit is linked against libpng-1.2. These versions are incompatible
- What version of libpng can your application link against?

- See *Software Engineering at Google*, Figure 21-1

Dependency management

- What versions of dependencies should you import?
- When should you upgrade dependency versions?

- SwE@Google book outlines four options:
  - Never upgrade
  - Semantic versioning
  - Bundled distributions
  - "Live at HEAD"

Dependency management tradeoffs

Never upgrade

- Predictable
  - Avoids failures due to changes outside of your control
- Natural when starting out, or for short-lived projects
  - Compatible with "vendoring"
- What happens when a dependency has a security vulnerability?
- What happens when a new dependency depends on newer versions of old dependencies?

Bundled distributions

- Defer dependency management to distribution maintainer
  - Responsible for maintaining compatibility while incorporating security updates
- Depend on the bundle and whatever dependency versions it provides
  - Common for commercial applications
- Limits (verified) portability
- Can't leverage latest features

Semantic versioning (SemVer)

- Dependency version numbers obey MAJOR.MINOR.PATCH format
  - Changes to PATCH should be fully compatible (bug fixes, security fixes)
  - Changes to MINOR may add functionality in a backwards-compatible manner
  - Changes to MAJOR indicate API changes
- Assumed by many build tools
  - Depend on a specific MAJOR version and a minimum MINOR version
- Challenges
  - Not all dependencies follow this scheme
  - Human maintainers make mistakes
  - Hyrum's Law: one person's "bug" is another's "feature"
  - Can be over-constraining (no solution to SAT problem)
    - Heuristics for relaxing some requirements

Which version to choose?

- For deterministic builds, choice shouldn't depend on when dependency resolution is performed
  - Lock files: capture results of dependency resolution
  - Newer dependencies will only be considered if locked versions do not satisfy constraints
  - Commit lock file to repository
    - It will be changed (and should be recommitted) when dependency resolution is run

- Go recommends choosing *minimum* (MINOR) version required by dependency network
- If MINOR versions are maintained as release branches, hopefully security fixes will be backported to them as PATCH releases

Compatibility

API

- Names of public functions and data types
- Recompilation should succeed
  - May be required to incorporate updates

ABI

- Function calling conventions
- Data structure layout
- Instructions, inlined system functions
- Dependent code does not need to be recompiled to incorporate updates

Compatibility

Backward compatibility

- Code that worked with an older version of a dependency will work with a newer version
  - Preserved across MINOR versions

- Implies that public types and functions cannot be removed
- For ABI compatibility, public data structures cannot change outside of "reserved" fields

Forward compatibility

- Code built with a newer version of a dependency will also work with an older version
  - Preserved across PATCH versions

- Implies that no new public types, fields, or functions may be added

"Live at HEAD"

- Analogous to trunk-based development in a monorepo
- Dependency maintainer responsible for not breaking all users
    - Effectively requires continuous integration for all software in the world
    - If compatibility cannot be maintained, maintainer will provide upgrade tool

- Some of this infrastructure already exists
    - "Rolling" Linux distributions (e.g. Gentoo) integrate tens of thousands of packages continuously
    - Programming languages (e.g. Scala, Rust) proactively test all changes against major libraries/applications

Dependency vulnerabilities

- NPM has a history of dependency-related disasters
  - left-pad unpublished
  - Bitcoin theft transitive dependency in event-stream
  - Ukraine war "protestware" in node-ipc

- Why was impact so large?
  - Tools depended on external repository services rather than internal mirror
  - Projects depended on floating instead of fixed versions
  - Projects were built "too continuously"
  - Fine-grained dependencies depended upon by many other libraries