A Generic Approach for Deploying and Upgrading Mutable Software Components

Sander van der Burg

Delft University of Technology, EEMCS, Department of Software Technology

June 3, 2012
Software deployment

A Generic Approach for Deploying and Upgrading Mutable Software Components

Sander van der Burg
Software deployment

Challenges

- Systems are becoming *bigger*, more distributed and more complicated
- Deployment takes a lot of *effort* and *time*
- Difficult to *reproduce* a configuration elsewhere
- *Upgrading* may break a system and may introduce significant downtimes
- Therefore, *automation* is needed
The Nix project is our solution to deployment complexity:

- Goal is to provide fully automated, reliable, reproducible deployment
- All tools are built around the Nix package manager
- Nix borrows concepts from purely functional languages, such as Haskell
The Nix project

Nix offers various *advantages* compared to conventional deployment tools:

- Ensures dependency *completeness*
- Ability to store multiple *variants* in isolation next to each other
- Strong guarantees of *reproducibility*
- *Atomic* upgrades and rollbacks
- Garbage collector capable of safely removing components no longer in use
Several tools extend Nix to a broader domain:

- **Disnix**, a Nix-based deployment tool for service-oriented systems
- **NixOS**, a Nix-based Linux distribution, managing complete system configurations
- **Hydra**, a Nix-based continuous build and integration server
- **Charon**, a NixOS-based cloud deployment tool
/nix/store/324pq1...-firefox-9.0.1

First part: 324pq1... is a hash code derived from all build-time dependencies:

- Libraries, build-scripts, source files, system architecture etc.
- A different build-time dependency yields a different hash code
- Components can be stored safely next to other variants
rec {
    stdenv = ...
    fetchurl = ...
    gtk = stdenv.mkDerivation {
      name = "gtk-2.24.1"; ...
    };

    firefox = stdenv.mkDerivation {
      name = "firefox-9.0.1";
      src = fetchurl {
        url = http://../firefox-9.0.1-source.tar.bz2;
        md5 = "195f6406b980dce9d60449b0074cdf92";
      };
      buildInputs = [ gtk ... ];
      buildCommand = ''
        tar xfvj $src
        cd firefox-
        ./configure
        make
        make install
      '';
    };
...
Building Nix packages

While building a package, many *side-effects* are removed:

- Environment variables are cleared and explicitly set, e.g. PATH.
- Various build tools are patched, e.g. gcc, ld to ignore global directories, such as `/usr/lib`.
- Optionally, builds are executed in a `chroot()` environment.
- Build results are made *immutable* by making the contents read-only.
A hash-code always identifies the same component, regardless on what machine it has been built:

- We can *download* a component with the same hash elsewhere, instead of building it ourselves
- We can *reproduce* entire configurations on a different machine in the network
- We can perform efficient upgrades. The Nix store is the *cache* for derivation function invocations
Disnix: Nix-based deployment for service-oriented systems

$ disnix-env -s services.nix -i infrastructure.nix -d distribution.nix
From the three Disnix models, a distributed deployment process is derived:

- **Build phase.** All services are built from source code using the Nix package manager.

- **Transfer phase.** All services and their dependencies are transferred to the target machines.

- **Activation phase.** Obsolete services are deactivated and new services are activated (end-user access may be blocked).
Mutable components

Nix and Nix applications can automatically reliably deploy static (*immutable*) parts of software systems.

What to do with *mutable* components, such as databases?
Mutable components have different properties:

- They have to change, so they must be *writable*
- Hosted inside a *container*: DBMS server, application server
- State is stored in an implementation/architecture specific format, which is not portable
- State files may contain *locks* and can be temporarily *inconsistent*
Containers provide tools capable of consistently capturing state in a portable format:

```bash
$ mysqldump --single-transaction mydatabase
```

- **Physical state.** The actual state stored on disk.
- **Logical representation.** Dump of state generation in a consistent portable format.
How can we deploy mutable software components with in conjunction with Disnix and other Nix related tools?
Dysnomia: A prototype tool supporting automated, reliable, reproducible deployment for *mutable* components
Managing mutable components

Provide an *interface* to each mutable component:

- **activate.** Activate mutable component in container
- **deactivate.** Deactivate mutable component in container
- **snapshot.** Snapshot current physical state into a portable format
- **incremental-snapshot.** Create an incremental snapshot in a portable format
- **restore.** Bring the mutable component in a specific state, by restoring a snapshot

Semantics differ among component types.
Identifying mutable components

We have to uniquely identify mutable component state generations:

/dysnomia/svn/default/disnix/34124

- Component type: svn (Subversion repository)
- Container identifier: default
- Component name: disnix
- Sequence number: 34124 (derived from Subversion revision number)
Extending Disnix: Basic method

- **Build phase.** All services are built from source code using the Nix package manager.

- **Transfer phase.** All services and their dependencies are transferred to the target machines.

- **Activation phase.** Obsolete services are deactivated and new services are activated (end-user access may be blocked).

- Deactivate obsolete mutable components and activate mutable components + state transfer (end-user access may be blocked)
Extending Disnix: Optimised method

From the three Disnix models, a distributed deployment process is derived:

- **Build phase.** All services are built from source code using the Nix package manager.
- **Transfer phase.** All services and their dependencies are transferred to the target machines.
- **State transfer (access NOT blocked)**
- **Activation phase.** Obsolete services are deactivated and new services are activated (end-user access may be blocked).
- **Deactivate obsolete mutable components and activate mutable components + incremental state transfer (end-user access may be blocked)**
Results

We have deployed and upgrades various types of systems in various environments:

- Industrial case study: SDS2
- Staff-Tracker toy systems (http://nixos.org/disnix)
- Open-source: ViewVC (http://viewvc.tigris.org)

All deployment actions yield the same result including the state of mutable components.
Various types of mutable components are supported:

- MySQL databases
- PostgreSQL databases
- Subversion repositories
- Ejabberd databases
- Web applications
- Processes
Conclusion

Is the world a better place now?
Limitations

Unfortunately... No. The costs may be high:

- File system is used as a common denominator.
- Snapshotting large data sets may take a lot of time
- A DMBS has more efficient replication features.
Related work

- Most conventional tools, deploy mutable components imperatively and in an ad-hoc manner.
- A different approach with Nix has been done, using Ext3COW
  - Is more efficient, but only captures physical state
  - Cannot address mutable components individually
Disnix (and other related Nix tooling) is available under Free and Open Source licenses:

- Disnix homepage: http://nixos.org/disnix
- Other Nix related software: http://nixos.org
Some discussion questions:

- How to "safely" upgrade components which are large in size?
- How can we efficiently capture generations of component states and how can we uniquely identify them?
- Can we upgrade state safely in a generic manner, without blocking external access?
- How can we efficiently transfer state generations to other machines in a generic manner?