Efficient Upgrading in a Purely Functional Component Deployment Model

CBSE 2005

Eelco Dolstra
eelco@cs.uu.nl

Universiteit Utrecht, Faculty of Science,
Department of Information and Computing Sciences

May 15, 2005

Overview

Nix deployment system

ICSE'04 Imposing a Memory Management Discipline on Software Deployment

LISA'04 Nix: A Safe and Policy-Free System for Software Deployment

⇒ `http://www.cs.uu.nl/groups/ST/Trace/Nix`

Contribution of this paper

- A purely functional deployment model is not incompatible with efficient upgrading.
- A generic method for computing patches in the presence of arbitrary renames, moves, etc.
- Patching can be made transparent between any set of components.
- Heuristics to select patch bases efficiently.

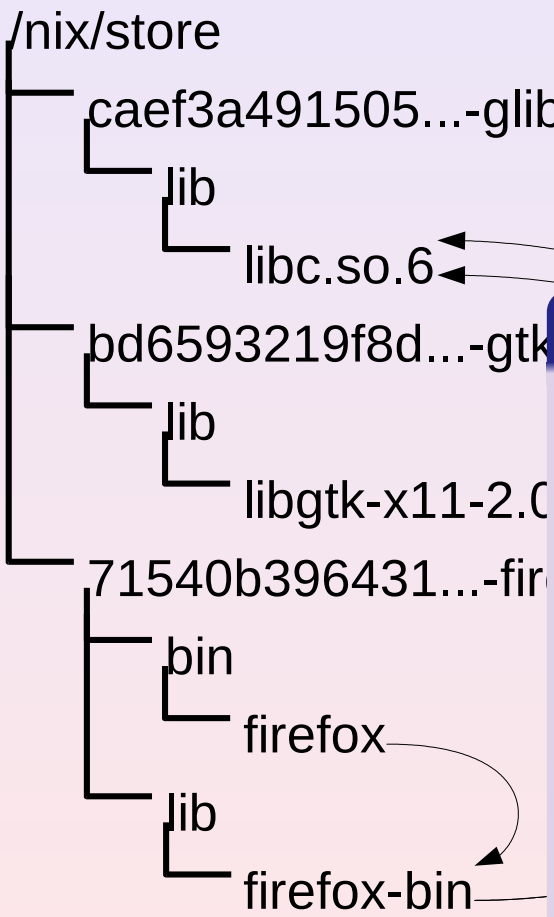
Software Deployment in Nix

- Components reside in the *Nix store*.
- Component names are *cryptographic hashes* of all build inputs.
- Components statically reference each other.
- This gains us:
  - Isolation ($\Rightarrow$ safe upgrades)
  - Side-by-side versioning
  - Variability for free
  - Reliable dependency info
- Components *never* change after they have been built.
- $\Rightarrow$ Purely functional model.

Software Deployment in Nix

- Components reside in the *Nix store*.
- Component names are *cryptographic hashes* of all build inputs.
- Components statically reference each other.
- This gains us:
  - Isolation ($\Rightarrow$ safe upgrades)
  - Side-by-side versioning
  - Variability for free
  - Reliable dependency info
- Components *never* change after they have been built.
- $\Rightarrow$ Purely functional model.

Software Deployment in Nix

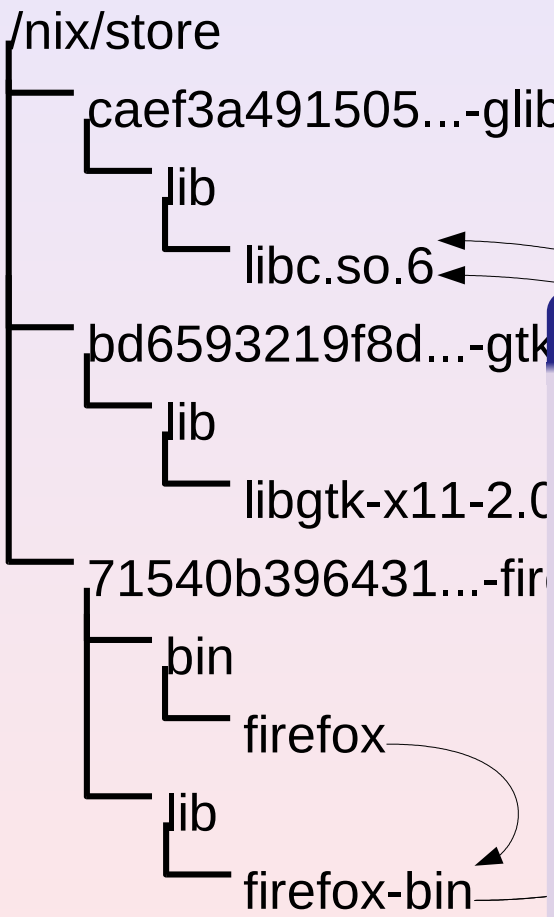


- Components reside in the *Nix store*.
- Component names are *cryptographic hashes* of all build inputs.
- Components statically reference each other.

Contents of **firefox-bin**

```
...
2e 36 00 6c 69 62 73 74  64 63 2b 2b 2e 73 6f 2e  |.6.libstdc++.so.|
36 00 6c 69 62 67 63 63  5f 73 2e 73 6f 2e 31 00  |6.libgcc_s.so.1.|
6c 69 62 70 74 68 72 65  61 64 2e 73 6f 2e 30 00  |libpthread.so.0.|
6c 69 62 63 2e 73 6f 2e  36 00 5f 5f 63 78 61 5f  |libc.so.6.__cxa_|
61 74 65 78 69 74 00 5f  65 64 61 74 61 00 5f 5f  |atexit._edata.__|
62 73 73 5f 73 74 61 72  74 00 2f 6e 69 78 2f 73  |bss_start./nix/s|
74 6f 72 65 2f 62 64 36  35 39 33 32 31 39 66 38  |tore/bd6593219f8|
64 63 62 36 33 30 61 34  35 35 62 31 61 35 37 66  |dcb630a455b1a57f|
36 34 36 33 33 2d 67 74  6b 2b 2d 32 2e 32 2e 34  |64633-gtk+-2.2.4|
2f 6c 69 62 3a 2f 6e 69  78 2f 73 74 6f 72 65 2f  |/lib:/nix/store/|
62 37 65 62 34 37 36 64  36 32 62 61 65 38 62 63  |b7eb476d62bae8bc|
...
```

Software Deployment in Nix

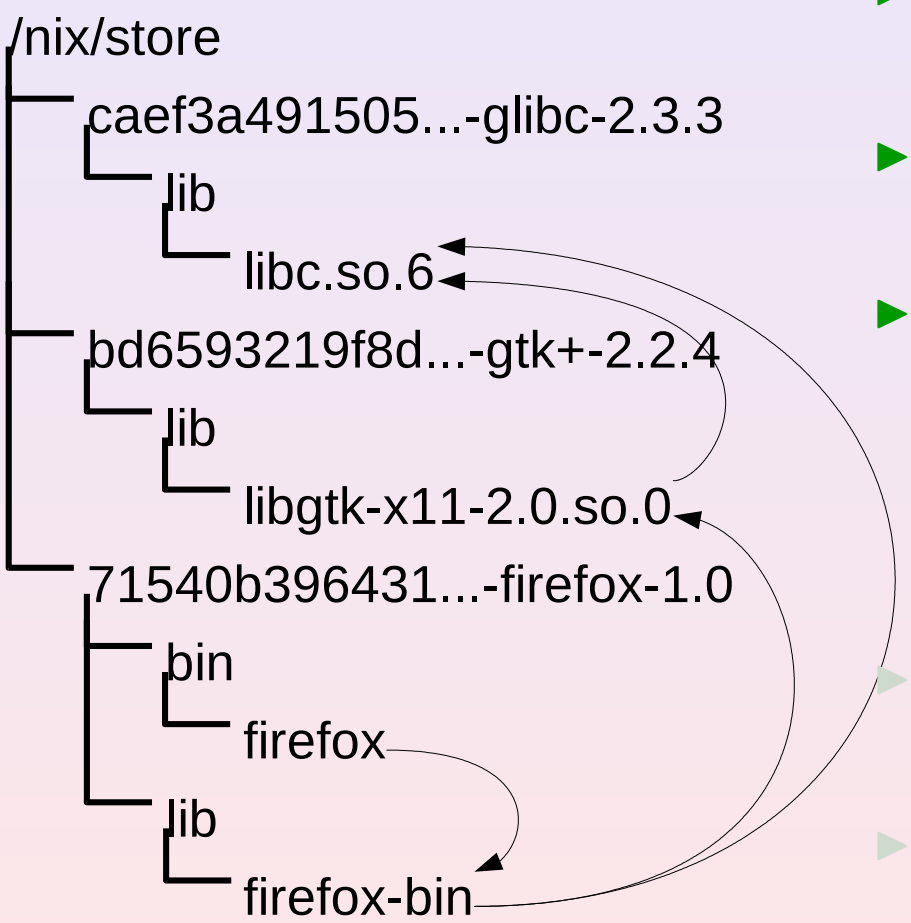


- Components reside in the *Nix store*.
- Component names are *cryptographic hashes* of all build inputs.
- Components statically reference each other.

Contents of **firefox-bin**

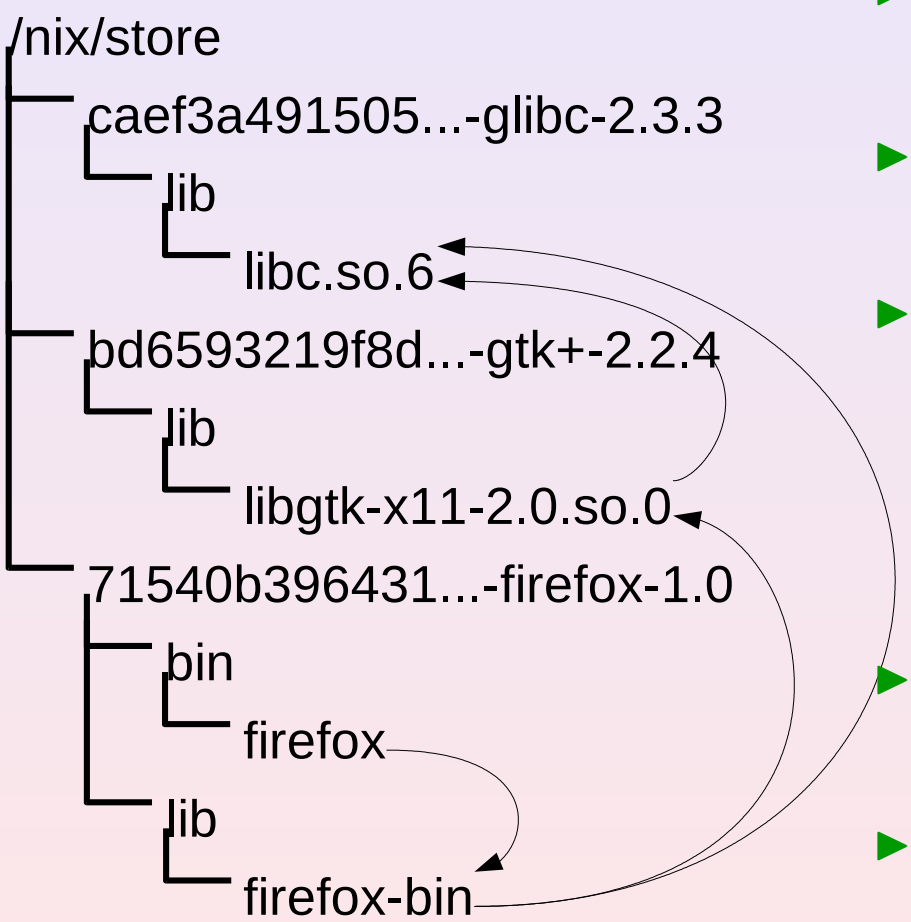
```
...
2e 36 00 6c 69 62 73 74  64 63 2b 2b 2e 73 6f 2e  |.6.libstdc++.so.|
36 00 6c 69 62 67 63 63  5f 73 2e 73 6f 2e 31 00  |6.libgcc_s.so.1.|
6c 69 62 70 74 68 72 65  61 64 2e 73 6f 2e 30 00  |libpthread.so.0.|
6c 69 62 63 2e 73 6f 2e  36 00 5f 5f 63 78 61 5f  |libc.so.6.__cxa_|
61 74 65 78 69 74 00 5f  65 64 61 74 61 00 5f 5f  |atexit._edata.__|
62 73 73 5f 73 74 61 72  74 00 2f 6e 69 78 2f 73  |bss_start./nix/s|
74 6f 72 65 2f 62 64 36  35 39 33 32 31 39 66 38  |tore/bd6593219f8|
64 63 62 36 33 30 61 34  35 35 62 31 61 35 37 66  |dcb630a455b1a57f|
36 34 36 33 33 2d 67 74  6b 2b 2d 32 2e 32 2e 34  |64633-gtk+-2.2.4|
2f 6c 69 62 3a 2f 6e 69  78 2f 73 74 6f 72 65 2f  |/lib:/nix/store/|
62 37 65 62 34 37 36 64  36 32 62 61 65 38 62 63  |b7eb476d62bae8bc|
...
```

Software Deployment in Nix



- Components reside in the *Nix store*.
- Component names are *cryptographic hashes* of all build inputs.
- Components statically reference each other.
- This gains us:
  - Isolation ($\Rightarrow$ safe upgrades)
  - Side-by-side versioning
  - Variability for free
  - Reliable dependency info
- Components *never* change after they have been built.
- $\Rightarrow$ Purely functional model.

Software Deployment in Nix



- Components reside in the *Nix store*.
- Component names are *cryptographic hashes* of all build inputs.
- Components statically reference each other.
- This gains us:
  - Isolation ($\Rightarrow$ safe upgrades)
  - Side-by-side versioning
  - Variability for free
  - Reliable dependency info
- Components *never* change after they have been built.
- $\Rightarrow$ Purely functional model.

Transparent source/binary deployment

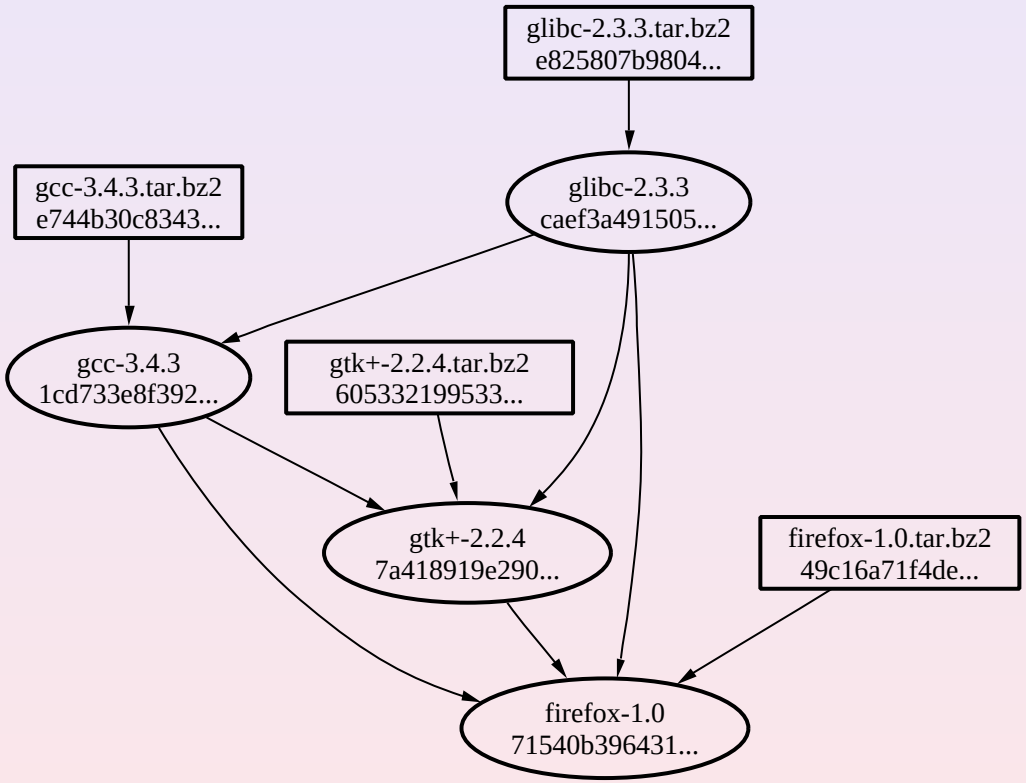
- Components are built from *Nix expressions*.
- To build a component, Nix computes hash, then checks if a *substitute* is available.

```
{ StorePath: /nix/store/075931820cae...-firefox-1.0
  URL: http://.../075931820cae...-firefox-1.0.nar.bz2
  Size: 11480169 }
```

  If so, the substitute is downloaded and unpacked.
- If not, the component is built normally.

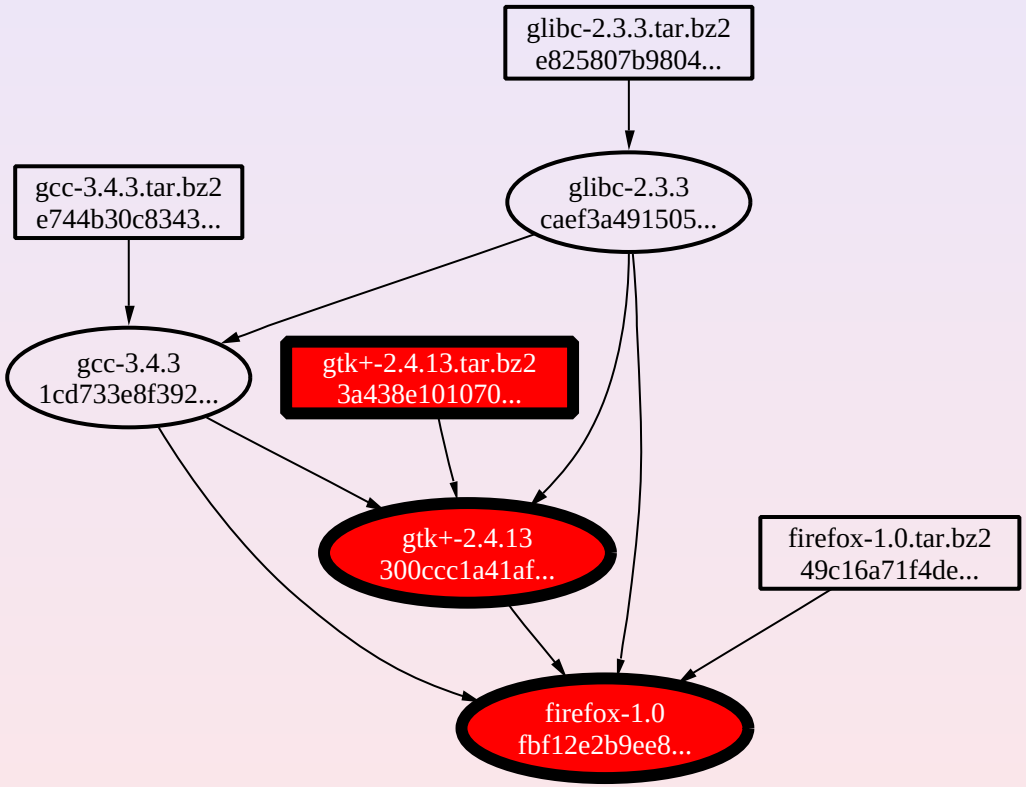
Upgrading

- If we change a fundamental component...
- ... we have to rebuild all components depending on it.
- Why not just do a dynamic library override (e.g., **LD_LIBRARY_PATH**)?
  - Static linking
  - Inlining, whole-program optimisations
  - Tool changes (e.g., compiler fixes)



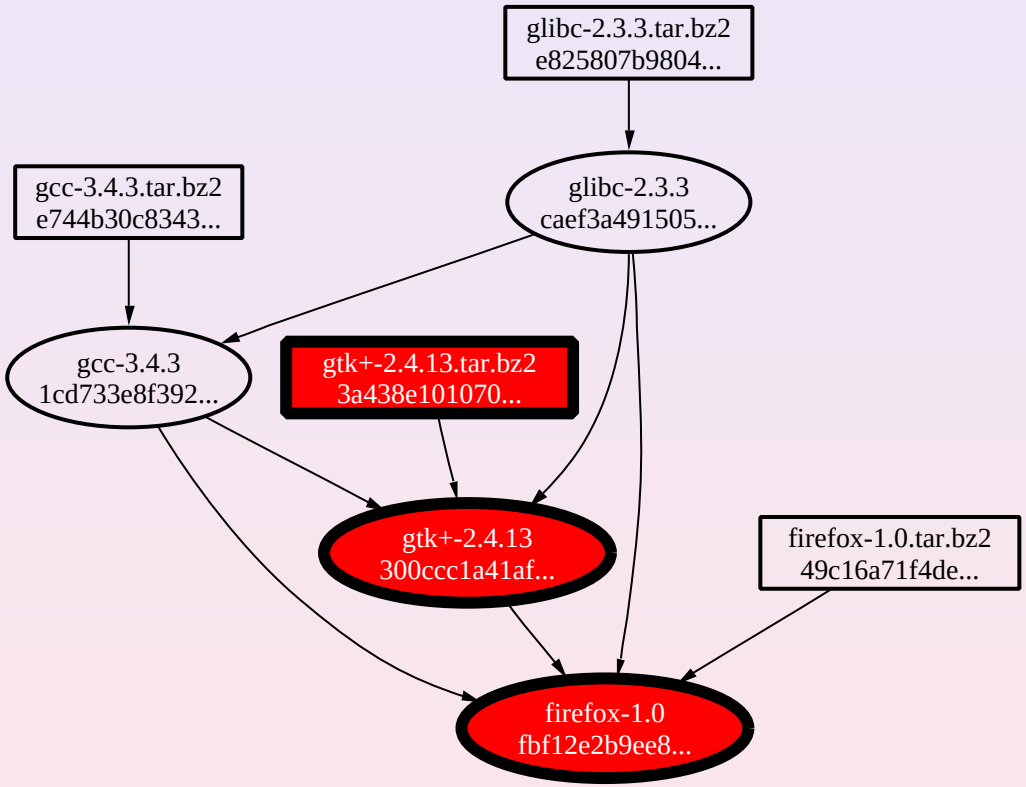
Upgrading

- If we change a fundamental component...
- ... we have to rebuild all components depending on it.
- Why not just do a dynamic library override (e.g., **LD_LIBRARY_PATH**)?
  - Static linking
  - Inlining, whole-program optimisations
  - Tool changes (e.g., compiler fixes)

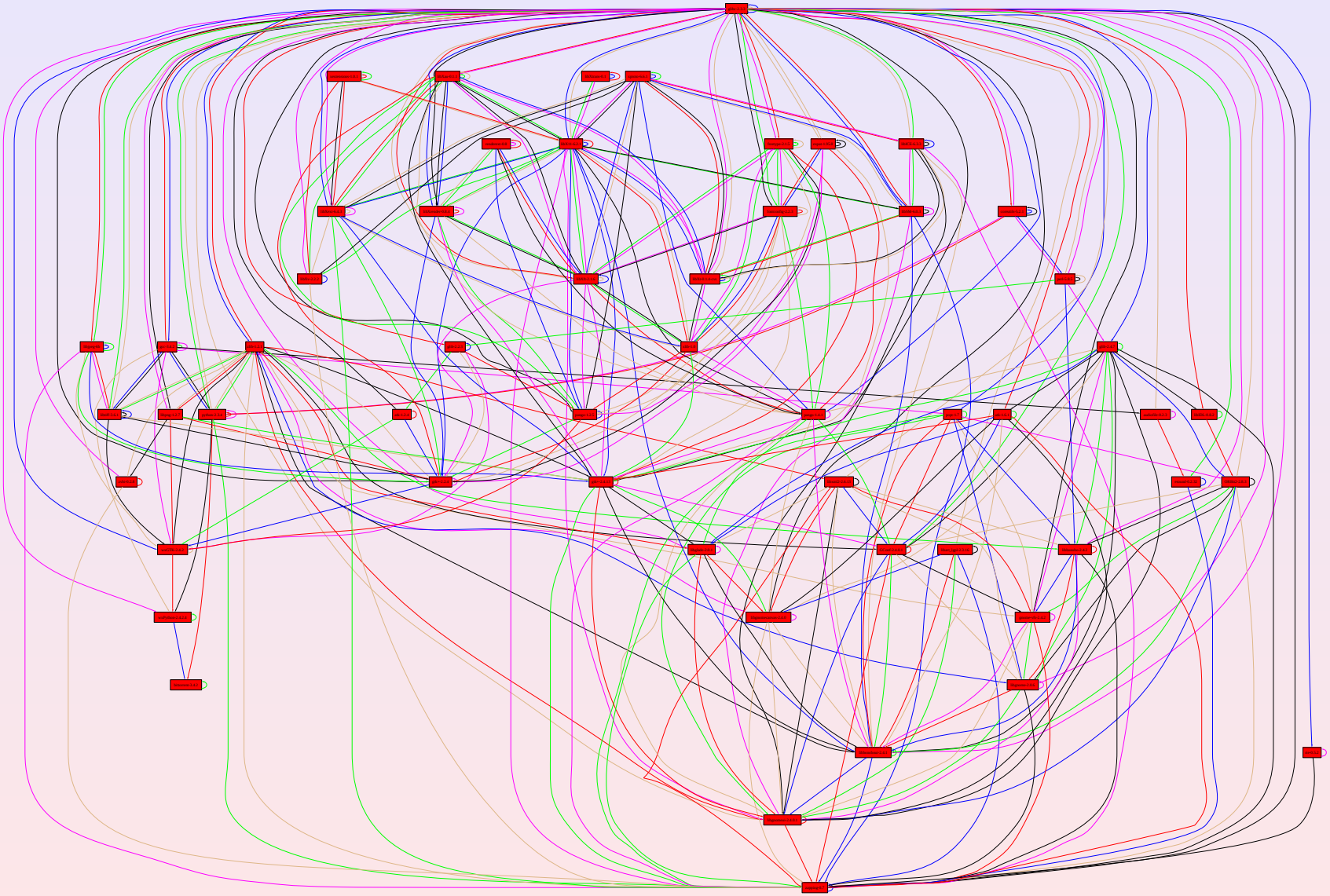


Upgrading

- If we change a fundamental component...
- ... we have to rebuild all components depending on it.
- Why not just do a dynamic library override (e.g., **LD_LIBRARY_PATH**)?
  - Static linking
  - Inlining, whole-program optimisations
  - Tool changes (e.g., compiler fixes)



Upgrading



The problem

- Purely functional model makes deploying fundamental upgrades (e.g., to **glibc**, **gcc**, **gtk**) inefficient.
- Must *rebuild* everything; that's developer/deployer-side, so it's okay. Only needs to be done once.
- Must *redeploy* everything, to every machine. Expensive/slow in terms of network bandwidth.

Binary patch deployment

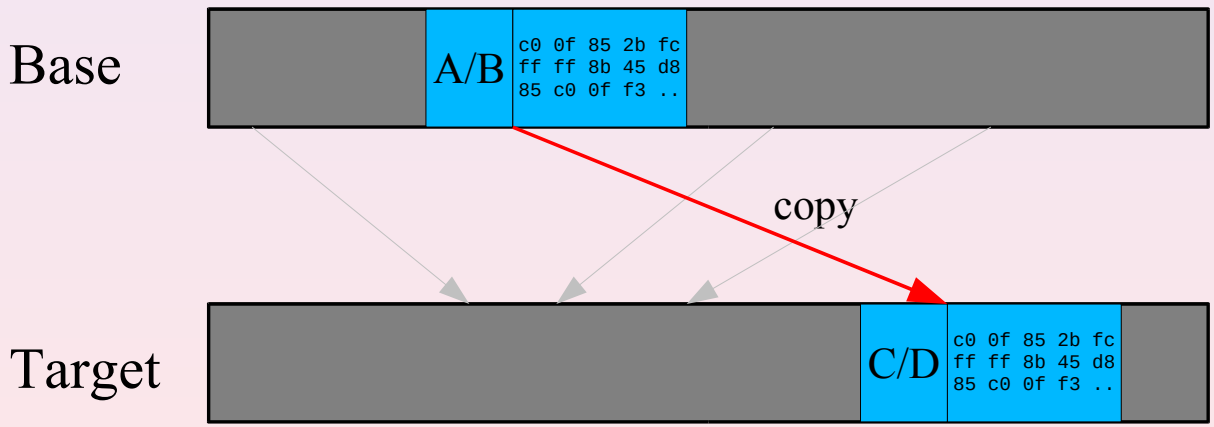
- Solution: deploy *binary patches* between store objects.
- Extend substitute downloader to download and apply patches:

```
patch {
  StorePath: /nix/store/5bfd71c253db...-firefox-1.0
  URL: http://.../52c036147222...-firefox-0.9-to-1.0
  Size: 357
  BasePath: /nix/store/075931820cae...-firefox-0.9
}
```

- I.e.,
  - If we *need* **/nix/store/5bfd71c253db...-firefox-1.0**
  - And we *have* **/nix/store/075931820cae...-firefox-0.9**,
  - Then we can *download* **http://.../52c036147222...-firefox-0.9-to-1.0**,
  - *Copy* the base to the target,
  - And *apply* the patch to the target.

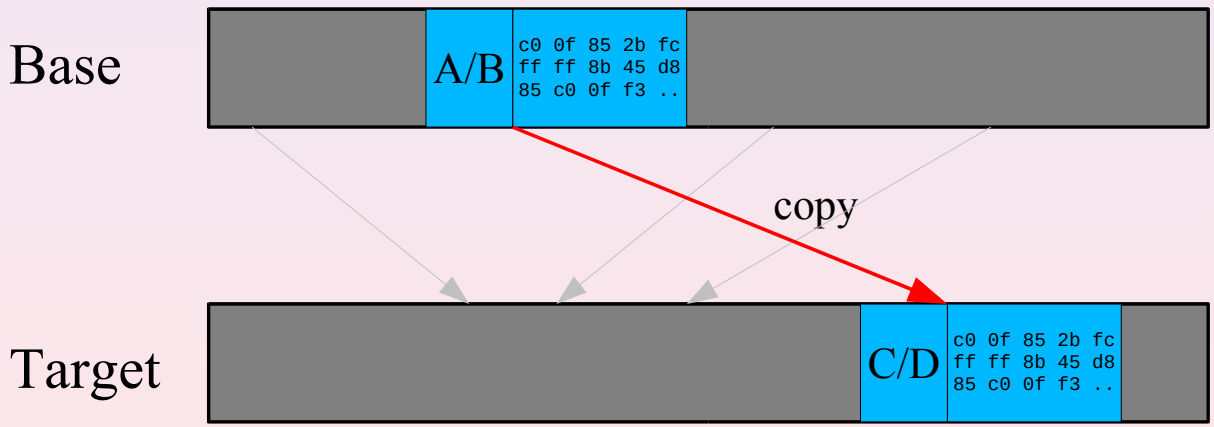
Computing patches

- Naive approach: compute file-by-file deltas (e.g., using **bsdiff**).
- How to deal with file renames, moves, deletions, etc.?
- Better approach: compute delta between *archive dumps* of store paths.
- The delta algorithm will deal with renames/moves/deletes automatically:



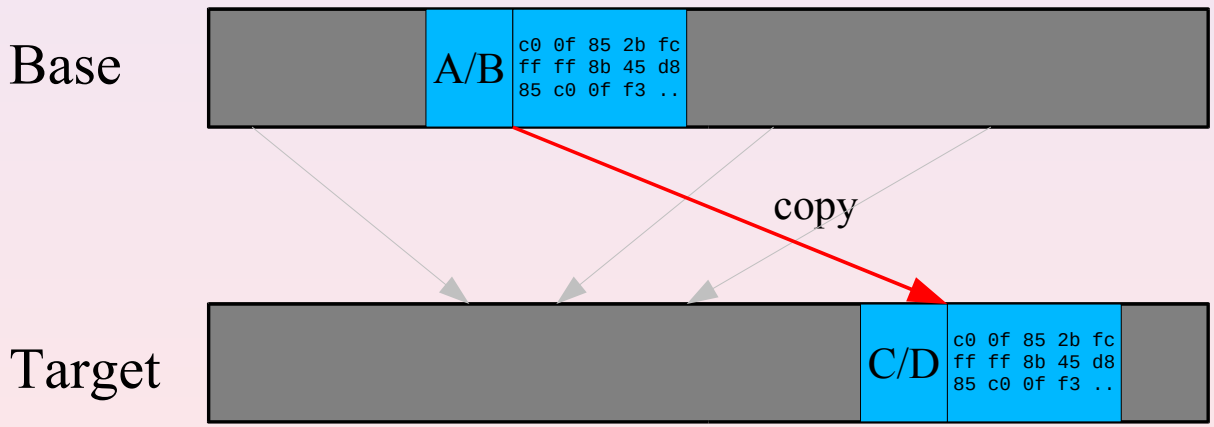
Computing patches

- Naive approach: compute file-by-file deltas (e.g., using **bsdiff**).
- How to deal with file renames, moves, deletions, etc.?
- Better approach: compute delta between *archive dumps* of store paths.
- The delta algorithm will deal with renames/moves/deletes automatically:



Computing patches

- Naive approach: compute file-by-file deltas (e.g., using **bsdiff**).
- How to deal with file renames, moves, deletions, etc.?
- Better approach: compute delta between *archive dumps* of store paths.
- The delta algorithm will deal with renames/moves/deletes automatically:



Patch chaining

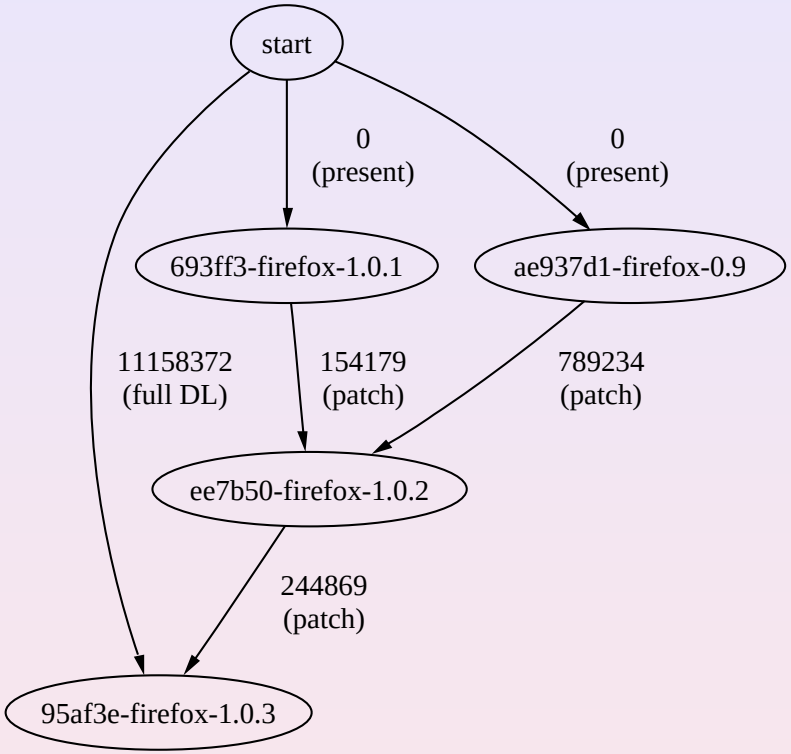
- Problem: if we have n releases, we do not want to produce $\Theta(n^2)$ patches.
- Solution: *patch chaining*.
- Find *sequence of patches* that transforms store object P to Q.
- Find shortest path in graph consisting of patches, available bases, and full downloads.

Patch chaining

- Problem: if we have n releases, we do not want to produce $\Theta(n^2)$ patches.
- Solution: *patch chaining*.
- Find *sequence of patches* that transforms store object P to Q.
- Find shortest path in graph consisting of patches, available bases, and full downloads.

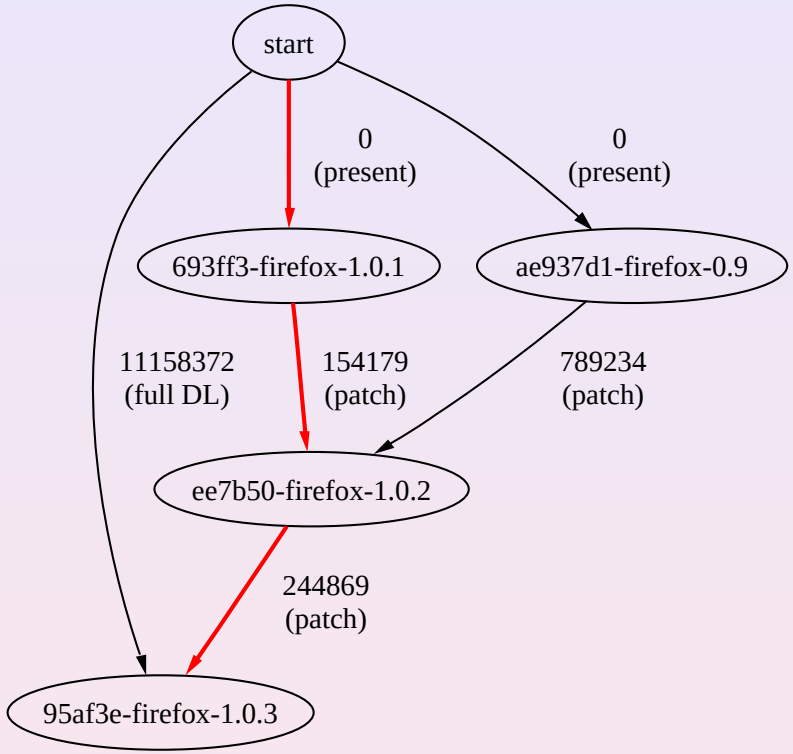
Patch chaining

- Problem: if we have n releases, we do not want to produce $\Theta(n^2)$ patches.
- Solution: *patch chaining*.
- Find *sequence of patches* that transforms store object P to Q.
- Find shortest path in graph consisting of patches, available bases, and full downloads.



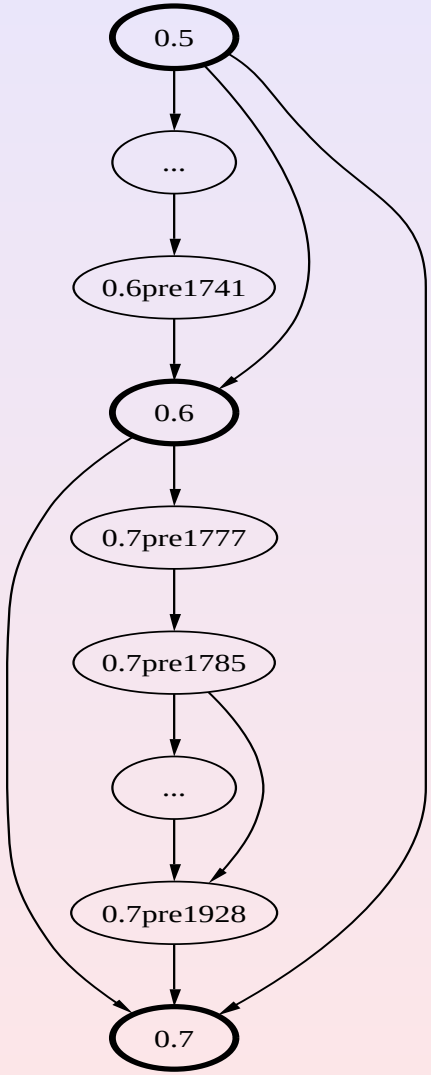
Patch chaining

- Problem: if we have n releases, we do not want to produce $\Theta(n^2)$ patches.
- Solution: *patch chaining*.
- Find *sequence of patches* that transforms store object P to Q.
- Find shortest path in graph consisting of patches, available bases, and full downloads.



Patch chaining (2)

- Chaining policies: between what releases do we produce patches?
- Example: for *Nixpkgs* we produce patches between
  - Directly succeeding pre-releases (several times per day, for developers subscribing to the **unstable** channel)
  - All succeeding releases (for developers subscribing to the **stable** channel)



Experience

- Applied to Nix Packages collection.
- Large set of Unix packages.
- Representative set of changes:
  - To "leaf" components: Firefox, Subversion, ...
  - To fundamental components **glibc**, **gcc**, (including ABI changes), ...

Release	Comps. changed	Full size	Total patch size	Savings	Remarks
0.7pre1931	1	1164K	45K	96.1%	Subversion 1.1.1 → 1.1.2
0.6pre1069	27	31.6M	162K	99.5%	X11 client libraries update
0.7pre1820	154	188.6M	598K	99.7%	Glibc loadlocale bug fix
0.6pre1489	147	180M	71M	60.5%	Glibc 2.3.2 → 2.3.3, GCC 3.3.3 → 3.4.2, etc.
0.7pre1980	154	197.2M	3748K	98.1%	GCC 3.4.2 → 3.4.3