Imposing a Memory Management Discipline on Software Deployment

Eelco Dolstra   Eelco Visser   Merijn de Jonge

Institute of Information & Computing Sciences
Utrecht University, The Netherlands

May 28, 2004
1. Why Does Software Deployment Fail?
2. Deriving a Solution
3. Practical Aspects
Outline

1. Why Does Software Deployment Fail?
   - Unresolved Component Dependencies
   - Component Interference
   - This Is a Big Problem

2. Deriving a Solution

3. Practical Aspects
Software deployment (the act of transferring software to another system) is surprisingly hard.

- It’s hard to ensure correctness (the software should work the same on the source and target systems).
- It’s too much work.
- Deployment systems tend to be inflexible.
When we deploy a component... we have to ensure that all its dependencies are present on the target system.
When we deploy a component... 

... we have to ensure that all its dependencies are present on the target system.
Operations on a component (install, upgrade, remove) often break other components (interference). E.g.:

- Upgrade of App2 breaks App1 due to upgrade of LibB to LibB’
- Removal of App3 breaks App1 due to removal of LibA
Component Interference

Operations on a component (install, upgrade, remove) often break other components (interference). E.g.:

- Upgrade of App2 breaks App1 due to upgrade of LibB to LibB'.
- Removal of App3 breaks App1 due to removal of LibA.
Operations on a component (install, upgrade, remove) often break other components (interference). E.g.:

- Upgrade of App2 breaks App1 due to upgrade of LibB to LibB'.
- Removal of App3 breaks App1 due to removal of LibA.
Note: these are runtime dependencies; there are still more build time dependencies.
1 Why Does Software Deployment Fail?

2 Deriving a Solution
- Deployment as Memory Management
- Deployment Requires Closures
- Pointer Discipline in PLs
- Imposing a Pointer Discipline on the FS
- Risks
- Preventing Interference

3 Practical Aspects
Deployment as Memory Management

memory ⇔ disk
objects (values) ⇔ components
addresses ⇔ path names
pointer dereference ⇔ I/O
pointer arithmetic ⇔ string operations
dangling pointer ⇔ reference to absent component
Deployment as Memory Management

\[
\begin{align*}
\text{memory} & \iff \text{disk} \\
\text{objects (values)} & \iff \text{components} \\
\text{addresses} & \iff \text{path names} \\
\text{pointer dereference} & \iff \text{I/O} \\
\text{pointer arithmetic} & \iff \text{string operations} \\
\text{dangling pointer} & \iff \text{reference to absent component}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Memory Management Concepts</th>
<th>Corresponding Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Disk</td>
</tr>
<tr>
<td>Objects (values)</td>
<td>Components</td>
</tr>
<tr>
<td>Addresses</td>
<td>Path names</td>
</tr>
<tr>
<td>Pointer dereference</td>
<td>I/O</td>
</tr>
<tr>
<td>Pointer arithmetic</td>
<td>String operations</td>
</tr>
<tr>
<td>Dangling pointer</td>
<td>Reference to absent component</td>
</tr>
</tbody>
</table>
memory ↔ disk
objects (values) ↔ components
addresses ↔ path names
pointer dereference ↔ I/O
pointer arithmetic ↔ string operations
dangling pointer ↔ reference to absent component
Deployment as Memory Management

memory ⇔ disk
objects (values) ⇔ components
addresses ⇔ path names
pointer dereference ⇔ I/O
pointer arithmetic ⇔ string operations
dangling pointer ⇔ reference to absent component
Correct deployment of component $c$ requires distributing the smallest set of components $C$ containing $c$ closed under the “has-a-pointer-to” relation.

So we have to discover the *pointer graph*.

This is exactly what garbage collectors for programming languages have to do.
Correct deployment of component $c$ requires distributing the smallest set of components $C$ containing $c$ closed under the “has-a-pointer-to” relation.

So we have to discover the *pointer graph*.

This is exactly what garbage collectors for programming languages have to do.
Correct deployment of component \( c \) requires distributing the smallest set of components \( C \) containing \( c \) closed under the “has-a-pointer-to” relation.

So we have to discover the *pointer graph*.

This is exactly what garbage collectors for programming languages have to do.
Correct deployment of component $c$ requires distributing the smallest set of components $C$ containing $c$ closed under the “has-a-pointer-to” relation.

So we have to discover the *pointer graph*.

This is exactly what garbage collectors for programming languages have to do.
GC requires a *pointer discipline*:

- Ideally, entire memory layout is known, and no arbitrary pointer formation (e.g., integer $\Leftrightarrow$ pointer casts).
- But even C/C++ has rules: pointer arithmetic is not allowed to move a pointer out of the object it points to.
- This is why *conservative GC* works: assume that everything that looks like a pointer *is* a pointer.

But software components do not have any pointer discipline.

- Any string can be a pointer.
- Pointer arithmetic and dereferencing directories can produce pointers to any object in the file system.
Each component should include in its path a unique identifying string.

Then we can apply conservative GC techniques to find pointers...

...which gives us the pointer graph!
Imposing a Pointer Discipline on the FS

/nix/store/eeeeaf...-subversion/bin/svn:

200000000200000000004000000 ............
04000000050e57464e0420100 ....P.td.B..
e0c20508e0c2050814000000 ............
140000000040000000004000000 ............
2f6e69782f73746f72652f38 /nix/store/8
643031336561383738643038 d013ea878d08
66323346164353462303131 f234ad54b011
313832313564662d676c6962 18215df-glib
c-2.3.2/lib/
6c642d6c696e75782e736f2e ld-linux.so.
320000000400000010000000 2............
01000000474e55000000000000000000 ...._.GNU.....
02000000000000000000000000000000 ............
83000000bb0000058000000 ............X...
ab000000ae000000a1000000 ............
000000006c0000000000000000000000 ...._.l......

### Each component should include in its path a unique identifying string.

### Then we can apply conservative GC techniques to find pointers...

### ...which gives us the pointer graph!
Imposing a Pointer Discipline on the FS

/nix/store/eeefafe...-subversion/bin/svn:

```
20000000020000000004000000 ............
0400000050e57464e0420100 ..........P.td.B..
e0c20508e0c2050814000000 ............
140000000040000000004000000 ............
2f6e69782f73746f72652f38 /nix/store/8
643031336561383738643038 d013ea878d08
6632334616435346230313f f234ad54b011
313832313564662d676c6962 18215df-glib
632d322e322e322f6c69622f c-2.3.2/lib/
6c642d6c696e7578e736f2e ld-linux.so.
320000000040000001000000 2............
01000000474e55000000000000000000 ....GNU.....
02000000000000000000000000000000000000000000000000
83000000bb00000058000000000000000000000000000000000
ab000000ae000000a100000000000000000000000000000000000
0000000006c00000000000000000000000000000000000000000
```

- Each component should include in its path a unique identifying string.
- Then we can apply conservative GC techniques to find pointers...
- ...which gives us the pointer graph!
Each component should include in its path a unique identifying string.

Then we can apply conservative GC techniques to find pointers...

...which gives us the pointer graph!
Each component should include in its path a unique identifying string.

Then we can apply conservative GC techniques to find pointers...

...which gives us the pointer graph!
As in all conservative GC approaches, there is a risk of pointer hiding.
- Compressed executables.
- UTF-16 encoded paths.

However, we haven’t observed this yet, despite Nixifying some 170 Unix packages.

I.e., this is a heuristic, but a reliable one.
The unique strings are cryptographic MD5 hashes of *all* inputs involved in building the component.

This prevents address collisions in the target address space (i.e., path name collisions in the target file system).
Outline

1. Why Does Software Deployment Fail?
2. Deriving a Solution
3. Practical Aspects
   - End Users
   - Developers
“I don’t want to type `/nix/-store/very-long-path/bin/svn all the time!”

Solution: synthesise a user environment of currently activated applications.

These are components themselves, so multiple environments can co-exist.

On Unix we can atomically switch between them.

These are roots of the garbage collector.
End Users

- “I don’t want to type `/nix/-store/very-long-path/bin/svn` all the time!”
- Solution: synthesise a user environment of currently activated applications.
- These are components themselves, so multiple environments can co-exist.
  - On Unix we can atomically switch between them.
  - These are roots of the garbage collector.
“I don’t want to type /nix/-store/very-long-path/bin/svn all the time!”

Solution: synthesise a user environment of currently activated applications.

These are components themselves, so multiple environments can co-exist.

On Unix we can atomically switch between them.

These are roots of the garbage collector.
“I don’t want to type `/nix/-store/very-long-path/bin/svn` all the time!”

Solution: synthesise a `user environment` of currently activated applications.

These are components themselves, so multiple environments can co-exist.

On Unix we can atomically switch between them.

These are roots of the `garbage collector`. 
“I don’t want to type `/nix/-store/very-long-path/bin/svn all the time!”

Solution: synthesise a user environment of currently activated applications.

These are components themselves, so multiple environments can co-exist.

On Unix we can atomically switch between them.

These are roots of the garbage collector.
“I don’t want to type `/nix/-store/very-long-path/bin/svn` all the time!”

Solution: synthesise a *user environment* of currently activated applications.

These are components themselves, so multiple environments can co-exist.

On Unix we can atomically switch between them.

These are roots of the *garbage collector.*
“I don’t want to type `/nix/-store/very-long-path/bin/svn all the time!”

Solution: synthesise a *user environment* of currently activated applications.

These are components themselves, so multiple environments can co-exist.

On Unix we can atomically switch between them.

These are roots of the *garbage collector*. 
“I don’t want to write
/nix/store/very-long-
path/... in my
Makefiles all the time!”

Solution: build actions
are generated from
high-level Nix
expressions.

Nix takes care of
computing hashes and
passes them to build
scripts.
“I don’t want to write /nix/store/very-long-path/... in my Makefiles all the time!”

Solution: build actions are generated from high-level Nix expressions.

Nix takes care of computing hashes and passes them to build scripts.

Nix expression for Subversion

```nix
{ localServer, stdenv, fetchurl , openssl ? null, db4 ? null, ... }
assert localServer -> db4 != null;
stdenv.mkDerivation {
  name = "subversion-1.0.3";
  builder = ./builder.sh;
  src = fetchurl {url=...};
  db4 = if localServer
    then db4 else null;
    then db4 else null;
  ...
}
```
Developers

“I don’t want to write /nix/store/very-long-path/… in my Makefiles all the time!”

Solution: build actions are generated from high-level Nix expressions.

Nix takes care of computing hashes and passes them to build scripts.

Build script for Subversion

tar xvfj $src
cd subversion-*
if test "$localServer"; then
    extraFlags=
        --with-berkeley-db=$db4
fi
./configure --prefix=$out \ $extraFlags
make
make install
Related Work

- Deployment / package managers: RPM, Gentoo, etc.
  - Unsafe — incomplete deployment, not atomic.
- Better build managers: Vesta, ClearCase.
  - Do not do deployment.
  - Cannot handle retained dependencies.
  - Not portable; rely on virtual file system.
- .NET / Java WebStart
  - Covers only executable resources.
  - “Unmanaged” file system.
  - Bound to a specific component technology.
Related Work

- Deployment / package managers: RPM, Gentoo, etc.
  - Unsafe — incomplete deployment, not atomic.
- Better build managers: Vesta, ClearCase.
  - Do not do deployment.
  - Cannot handle retained dependencies.
  - Not portable; rely on virtual file system.

- .NET / Java WebStart
  - Covers only executable resources.
  - “Unmanaged” file system.
  - Bound to a specific component technology.
Related Work

- Deployment / package managers: RPM, Gentoo, etc.
  - Unsafe — incomplete deployment, not atomic.
- Better build managers: Vesta, ClearCase.
  - Do not do deployment.
  - Cannot handle retained dependencies.
  - Not portable; rely on virtual file system.

.NET / Java WebStart
- Covers only executable resources.
- “Unmanaged” file system.
- Bound to a specific component technology.
Conclusion

- Paradigm: solving deployment problems by applying PL techniques.
- Safe deployment requires identification and deployment of closures.
- Closures can be identified using unique hashes.
- These also ensure non-interference between versions/variants.
- Multiple user environments.
- Safe garbage collection.

More information:
http://www.cs.uu.nl/groups/ST/Trace/Nix.
“How to handle security patches (e.g., in the C library)? There you do want destructive updates.”

- No you don’t. How to roll-back if the patch breaks things?
- Just deploy the new components; to the extent that there is sharing with old ones, no rebuilds / redownloads are necessary.
- In the case of dynamic libraries, wrapper packages can be used to prevent a mass rebuild.