Automated Deployment of a Heterogeneous Service-Oriented System

Sander van der Burg  Eelco Dolstra

Delft University of Technology, EEMCS, Department of Software Technology

September 3, 2010
The Service Oriented Computing (SOC) paradigm is very popular to build distributed applications.
- Philips has developed a service oriented architecture for Asset Tracking & Utilisation services in a hospital environment.
- Service Development Support System (SDS2)
- Motivating example throughout the paper
Service Development Support System (SDS2)
A hospital contains a wide range of medical devices
Each produce status and event logs in their own format
Difficult to perform analysis on data
How can we transform these implicit datasets into something useful?
SDS2: Utilisation Service

Sander van der Burg, Eelco Dolstra

Automated Deployment
MySQL: data storage

Ejabberd: messaging system for transmitting status and location events

Java: implementation language

Apache Axis: for implementing web services

Google Web Toolkit: for implementing web application front-ends
SDS2 must be eventually *deployed*, i.e. made available for use
Create a global configuration file with service locations, and:

**Databases:**
- Transfer schema to target machine
- Install database on MySQL server

**Web services:**
- Compile Java web service code
- Package web application archive
- Activate web service on target machine

**Web applications:**
- Compile Java code to JavaScript code
- Compile Java web application code
- Package web application archive
- Activate web application on target machine
Deploying a service oriented system such as SDS2 is very difficult!
Deploying is *labourious*:

- *Time consuming*. Deploying SDS2 takes several hours for a single machine
- *Subject to errors*. Because it is performed manually
- *Complexity* is proportional to number of machines in the network
Deployment steps must be performed in the right order:

- For a web service providing data from a database, the database must be activated first
- If activated in the wrong order, data may be inaccessible
Challenges

We do not exactly know what the dependencies of a service are:

- Difficult to upgrade reliably
- Difficult to upgrade efficiently
Challenges

Upgrading is not *atomic*:

- A user can observe that the system is changing
- While upgrading certain features may become inaccessible
Deploying in *heterogeneous* networks is even more complex.
Existing deployment tools have various limitations:

- Designed for specific component technologies (requires to exclusively use one type of language/implementation):
  - Enterprise Java Beans
  - OSGi
  - Embedded systems

- Designed for specific environments:
  - GoDIET: Designed for the DIET grid computing platform
  - CODEWAN: Ad-hoc networks

- Generic approaches (lack desirable non-functional properties):
  - Software Dock: no dependency-completeness, atomic upgrading
  - TACOMA: no dependency-completeness, atomic upgrading
Distributed deployment extension for the Nix package manager
Captures deployment specification in models
Performs complete deployment process from models
Guarantees complete dependencies
Component agnostic
Supports atomic upgrades and rollbacks
Disnix

$ disnix-env -s services.nix -i infrastructure.nix -d distribution.nix

Sander van der Burg, Eelco Dolstra
Automated Deployment
{distribution, system}:

let pkgs = import ../top-level/all-packages.nix {
  inherit distribution system;
}; in

{ mobileeventlogs = {
    name = "mobileeventlogs";
    pkg = pkgs.mobileeventlogs;
    type = "mysql-database";
  };
MELogService = {
    name = "MELogService";
    pkg = pkgs.MELogService;
    dependsOn = { inherit mobileeventlogs; };
    type = "tomcat-webapplication";
  };
SDS2AssetTracker = {
    name = "SDS2AssetTracker";
    pkg = pkgs.SDS2AssetTracker;
    dependsOn = { inherit MELogService ...; };
    type = "tomcat-webapplication";
  };
...}
Infrastructure model

```
{
    test1 = {
        hostname = "test1.net";
        tomcatPort = 8080;
        mysqlUser = "user";
        mysqlPassword = "secret";
        mysqlPort = 3306;
        targetEPR = http://test1.net/.../DisnixService;
        system = "i686-linux";
    };
    test2 = {
        hostname = "test2.net";
        tomcatPort = 8080;
        ...
        targetEPR = http://test2.net/.../DisnixService;
        system = "x86_64-linux";
    };
}
```

Captures machines in the network and their relevant properties and capabilities.
{infrastructure}:

{
    mobileeventlogs = [ infrastructure.test1 ];
    MELogService = [ infrastructure.test2 ];
    SDS2AssetTracker = [ infrastructure.test1 infrastructure.test2 ];
    ...
}

Maps services to machines
Specifications are used to derive deployment process:

- **Building** services from source code
- **Transferring** services to target machines
- **Deactivating** obsolete services and **activating** new services
Every component built from source code is stored in the Nix store:

/nix/store/y2ssvzcd86...-SDS2EventGenerator

- Former part: y2ssvzcd86..., SHA256 hash code derived from all build-time dependencies of the component: compilers, libraries, build scripts
- Latter part: SDS2EventGenerator: name of the component
Building services
Transferring closures

- Copy services and intra-dependencies to target machines in the network
- Only components missing on target machines are transferred
- Always safe. Files are never overwritten/removed
Transition phase

- Inter-dependency graph is derived from specifications
- Deactivate obsolete services (none, if activating new configuration)
- Activate new services
- Derive order and dependencies from dependency-graph
Service activation

- Every service has a type: mysql-database, tomcat-webapplication, process, ...
- Types are attached to external processes, which take 2 arguments
  - activate or deactivate
  - Nix store component
- Infrastructure model properties are passed as environment variables
Atomic upgrading

- Two-phase commit protocol mapped onto Nix primitives
- Commit-request-phase (distribution phase):
  - Build sources
  - Transfer service closures
- Commit-phase (transition phase):
  - Deactivating/activating services
  - Optionally block/queue connections from end-users
Atomic upgrading

- In case of *failure* during commit-request: system is not affected
  - No files overwritten
  - Will be removed by garbage collector
- In case of *failure* during commit:
  - Rollback (transition to previous configuration)
We have created deployment models for SDS2
Used to atomatically deploy SDS2 in a network of 4 32-bit Linux and 4 64-bit Linux machines
Deployment takes minutes for a single machine. For each additional machine a couple of minutes.
Upgrading only took seconds in most cases (only changed parts were replaced)
Because all components and dependencies are known:

- Deployment steps were always performed in the right order
- No failures due to breaking inter-dependencies
- Upgrading is *efficient*; we only have to upgrade what is necessary
- Upgrading is *reliable*; we always know and include the prerequisites
Results

- Upgrades are (almost) atomic
  - Minor issue: web application in browser must be refreshed, which is not under Disnix’ control
- Components can be built for a particular platform
  - On the coordinator machine (if capable)
  - On the target machines
Conclusion:

- With Disnix we can automatically, efficiently and reliably deploy service oriented systems similar to SDS2 in heterogeneous environments

Future work:

- Support dynamic migration of database (and other mutable state)
- Experimenting in larger networks
- More investigation in upgrading web applications
Questions

- Disnix and other related Nix software can be found at: http://nixos.org
- Any questions?