Deploying .NET services with Disnix

Sander van der Burg

Delft University of Technology, EEMCS,
Department of Software Technology

Philips Healthcare, Philips Informatics Infrastructure (PII),
Best

February 2, 2012
The Service Oriented Computing (SOC) paradigm is very popular to build distributed applications.
Systems are composed of distributable components (or services), working together to achieve a common goal.
Deployment of Service Oriented Systems is complex:

- Many types of components.
- Various operating systems.
- Multiple variants.

Deploying .NET services with Disnix
Deployment of Service Oriented Systems is *laborious*:
• Deployment of Service Oriented systems is *error-prone*
• *Upgrading* is even harder
Deployment of Service Oriented systems is *error-prone*

*Upgrading* is even harder
Deployment of Service Oriented systems is *error-prone*

*Upgrading* is even harder

**Upgrading**

- How to make upgrades safe and reliable?
- What to do with incoming connections?
- Rollbacks?
Example case: StaffTracker

Data layer
- rooms
- staff
- zipcodes

Service layer
- RoomService
- StaffService
- ZipcodeService
- GeolocationService

Presentation layer
- StaffTracker
A toolset for \textit{distributed deployment} of service-oriented systems:

- Model-driven
- Component-agnostic
- Supported on multiple platforms:
  - Linux, FreeBSD, Mac OS X, Windows (through Cygwin)
Disnix is built on top of the Nix package manager:

- Nix expression language to *declaratively* specify build actions
- Ability to store *multiple* versions and variants simultaneously
- *Complete* dependencies
- *Reliable* and atomic upgrades and rollbacks
- *Garbage collector* to safely remove components no longer in use
Disnix: Models

```bash
$ disnix-env -s services.nix -i infrastructure.nix -d distribution.nix
```
Disnix expressions

{ stdenv, fetchurl, unzip, apacheAnt }:
{ staff }:

stdenv.mkDerivation {
    name = "StaffService";
    src = fetchurl {
        url = http://.../StaffService-0.1.zip;
        sha256 = "0fpjlr3bfindo0y94bk442x2p...";
    };
    buildInputs = [ unzip apacheAnt ];
    buildCommand = ''
        unzip $src
        ant webapp
        mkdir -p $out/webapps
        cp *.war $out/webapps

    cat > $out/conf/Catalina/StaffService.xml <<EOF
    <Context>
        <Resource name="jdbc/staff" auth="Container" type="javax.sql.DataSource"
            url="jdbc:mysql://${staff.target.hostname}/${staff.name}" ... />
    </Context>
    EOF
    '';
}
Disnix expressions

{ stdenv, fetchurl, unzip, apacheAnt }:
{ staff }:

stdenv.mkDerivation {
  name = "StaffService";
  src = fetchurl {
    url = http://.../StaffService-0.1.zip;
    sha256 = "0fpjlr3bfind0y94bk442x2p...";
  };
  buildInputs = [ unzip apacheAnt ];
  buildCommand = "unzip $src ant webapp mkdir -p $out/webapps cp *war $out/webapps cat > $out/conf/Catalina/StaffService.xml <<EOF
<Context>
  <Resource name="jdbc/staff" auth="Container" type="javax.sql.DataSource"
    url="jdbc:mysql://${staff.target.hostname}/${staff.name}" ...
  </Context>
EOF
"; 
}
Intra-dependency composition

{pkgs, system}:

rec {
    apacheAnt = ...
    stdenv = ...

    staff = import ../pkgs/staff {
        inherit stdenv;
    };

    StaffService = import ../pkgs/StaffService {
        inherit stdenv fetchurl unzip apacheAnt;
    };

    StaffTracker = import ../pkgs/StaffTracker {
        inherit stdenv fetchurl unzip apacheAnt;
    };

    ...
}

Composes components locally, by calling functions with *intra-dependency* arguments.
rec { 
    staff = { 
        name = "staff";
        pkg = pkgs.staff;
        type = "mysql-database";
    };
    StaffService = { 
        name = "StaffService";
        pkg = pkgs.StaffService;
        dependsOn = { inherit staff; };
        type = "tomcat-webapplication";
    };
    StaffTracker = { 
        name = "StaffTracker";
        pkg = pkgs.StaffTracker;
        dependsOn = { inherit StaffService ...; };
        type = "tomcat-webapplication";
    };
    ... 
}
Service model

{distribution, pkgs, system}:

rec {
  staff = {
    name = "staff";
    pkg = pkgs.staff;
    type = "mysql-database";
  }
}

StaffService = {
  name = "StaffService";
  pkg = pkgs.StaffService;
  dependsOn = { inherit staff; };
  type = "tomcat-webapplication";
};

StaffTracker = {
  name = "StaffTracker";
  pkg = pkgs.StaffTracker;
  dependsOn = { inherit StaffService ...; };
  type = "tomcat-webapplication";
};

...
Captures machines in the network and their relevant properties and capabilities.
Distribution model

{infrastructure}:

{
    staff = [ infrastructure.test1 ];
    StaffService = [ infrastructure.test2 ];
    StaffTracker = [ infrastructure.test1 infrastructure.test2 ];
    ...
}

Maps services to machines
Complete deployment process is derived from the models:

- **Building** services from source code
  - Optionally, build can be delegated to a capable machine
- **Transferring** services (and intra-dependencies) to target machines
- **Deactivation** of obsolete services and **activation** of new services in the right order
  - Optionally, incoming connections can be blocked/queued
Main idea: store all packages in isolation from each other:

```
/nix/store/rpdqxnilb0cg...-firefox-3.5.4
```

Paths contain a 160-bit **cryptographic hash of all** inputs used to build the package:

- Sources
- Libraries
- Compilers
- Build scripts
- ...
Nix detected runtime dependencies of a build
Nix scans all files of a component for hash-codes of a dependency, e.g. RPATH defined in a ELF header
Uses a notion of closures to guarantee completeness:
  - If a specific component needs to be deployed, all its runtime dependencies are deployed first
Runtime dependencies

/nix/store
  └── nqapqr5cyk...-glibc-2.9
      ├── lib
      │   └── libc.so.6
      └── ccayqylscm...-jre-1.6.0_16
            └── bin
                 └── java
                   └── SDS2Util.jar
  └── n7s283c5yg...-SDS2Util
      └── share
          └── java
              └── SDS2Util.jar
  └── y2ssvzcd86...-SDS2EventGenerator
      └── bin
          └── SDS2EventGenerator
              └── share
                  └── java
                      └── SDS2EventGenerator.jar
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
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)
Example cases:

- **Staff Tracker.** Toy SOA system which comes in two variants (PHP/MySQL and Webservices/Java).
- **ViewVC.** Open-source web based Subversion repository viewer (http://viewvc.tigris.org).
- **SDS2.** Industrial SOA case study from Philips Research. Java EE based.
So far, Disnix case studies are deployed on UNIX-like systems, using portable component technologies.

Deployment of .NET services on a Windows infrastructure is challenging because of the underlying purely functional deployment model of Nix and its UNIX heritage.
Requirements

Buildtime support:
- Compile Visual Studio solutions and produce assemblies
  - Invoke MSBuild.exe with right parameters
- The build process must find its buildtime dependencies

Runtime support:
- Assemblies must find its runtime dependencies
- Integrate .NET CLR dependency resolver with Nix

Activation support:
- Activate web application components in IIS
- Activate databases in Microsoft SQL server
How to extend Disnix to support .NET service deployment?
Nix is supported on Windows through Cygwin:

- Offers UNIX implementation on top of Windows
- Allows combining UNIX and Windows utilities
Several Cygwin utilities map Windows concepts to UNIX concepts and vice versa:

- Can be used to glue UNIX tooling to Windows processes
- `cygpath`, UNIX path names $\leftrightarrow$ Windows path names
A .NET Nix build function dotnetenv.buildSolution is implemented:

- Takes source code of a Visual Studio C# solution
- Produces output of the solution into a unique store path, based on the input arguments of this function
Build function implementation

{stdenv, dotnetfx}:
{ name, src, slnFile, targets ? "ReBuild"
, options ? "/p:Configuration=Debug;Platform=Win32"
, assemblyInputs ? []
}:

stdenv.mkDerivation {
    inherit name src;
    buildInputs = [ dotnetfx ];

    installPhase = ''
    for i in ${toString assemblyInputs}; do
        windowsPath=$(cygpath --windows $i)
        AssemblySearchPaths="$AssemblySearchPaths;$windowsPath"
    done
    export AssemblySearchPaths

    ensureDir $out
    outputPath=$(cygpath --windows $out)\$
    MSBuild.exe ${slnFile} /nologo /t:${targets} \
        /p:OutputPath=$outPath ${options} ...
    '';
}
Disnix expression

```powershell
{dotnetenv}:
{zipcodes}:

dotnetenv.buildSolution {
    name = "ZipcodeService";
    src = ../../services/webservices/ZipcodeService;
    baseDir = "ZipcodeService";
    slnFile = "ZipcodeService.csproj";
    targets = "Package";
    preBuild = '
        sed -e 's|\SQLEXPRESS|${zipcodes.target.hostname}\SQLEXPRESS|'
            -e 's|Initial Catalog=zipcodes|Initial catalog=${zipcodes.name}|'
            -e 's|User ID=sa|User ID=${zipcodes.target.microsoft.SqlServerUsername}|'
            -e 's|Password=admin123|Password=${zipcodes.target.microsoft.SqlServerPassword}|'
            Web.config
    ';
}
```

Builds and configures a Visual Studio ASP.NET project
The .NET runtime has complicated means to locate runtime dependencies.
Resolving runtime dependencies

.NET runtime locaties assemblies:

- Determines the correct version of the assembly (strong-named assemblies only)
- Checks whether the assembly has been bound before (strong-named assemblies only)
  - If so, it uses this version
  - Shares same assemblies in memory between applications
- Checks the Global Assembly Cache (GAC) (strong-named assemblies only)
- Probes the assembly
  - Checking the <codebase> element in the application .config
  - Probing heuristics
A `<codebase>` element in the application `.config` file can specify its own assembly references:

- Private assemblies can only reside in the directory or a subdirectory of the executable
- Strong-named assemblies can be invoked from an arbitrary location, including remote locations
- Referenced assembly is cached, therefore a strong-name is required
You can only automatically resolve runtime dependencies in the basedir as an executable.

.NET reflection API can load private assemblies from any location.

.NET CLR fires an AssemblyResolve event if a dependency cannot be found.

- Can be used to load a private runtime assembly from arbitrary locations, e.g. the Nix store.
namespace HelloWorldWrapper {
    class HelloWorldWrapper {
        private String[] AssemblySearchPaths = {
            @"C:\cygwin\nix\store\23ga...-ALibrary",
            @"C:\cygwin\nix\store\833p...-BLibrary"
        };
        private String ExePath = @"C:\cygwin\nix\store\27f2...-Executable\Executable.exe";
        private String MainClassName = "SomeExecutable.Executable";

        public HelloWorldWrapper(string[] args) {
            AppDomain currentDomain = AppDomain.CurrentDomain;
            currentDomain.AssemblyResolve +=
                new ResolveEventHandler(MyResolveEventHandler);

            Assembly exeAssembly = Assembly.LoadFrom(ExePath);
            Type mainClass = exeAssembly.GetType(MainClassName);
            MethodInfo mainMethod = mainClass.GetMethod("Main");
            mainMethod.Invoke(this, new Object[] {args});
        }
    }
}

static void Main(string[] args) {
    new HelloWorldWrapper(args);
}
private Assembly MyResolveEventHandler(object sender, ResolveEventArgs args) {
    Assembly MyAssembly;
    String assemblyPath = "";
    String requestedAssemblyName = args.Name.Substring(0, args.Name.IndexOf(','));

    foreach (String curAssemblyPath in AssemblySearchPaths) {
        assemblyPath = curAssemblyPath + "/" + requestedAssemblyName + "\dll";

        if (File.Exists(assemblyPath))
            break;
    }

    MyAssembly = Assembly.LoadFrom(assemblyPath);

    return MyAssembly;
}
Wrapper function

{dotnetenv, MyAssembly1, MyAssembly2}:

dotnetenv.buildWrapper {

    name = "My.Test.Assembly";
    src = /path/to/source/code;
    slnFile = "Assembly.sln";
    assemblyInputs = [
        dotnetenv.assembly20Path
        MyAssembly1
        MyAssembly2
    ];
    namespace = "TestNamespace";
    mainClassName = "MyTestApplication";
    mainClassFile = "MyTestApplication.cs";
}

Builds a Visual Studio project and creates a wrapper around the executable
Activation scripts for *mssql-database* and *iis-webapplication* must be provided.

- *mssql-database* invokes `OSQL.EXE` to automatically create a database and to import table definitions.
- *iis-webapplication* invokes `MSDeploy.exe` to deploy web applications on a IIS server.
Example case: StaffTracker.NET

Ported from Java / Apache Axis2 to .NET / C# / WCF
Conclusion

With a number of Disnix extensions it’s possible to automatically deploy a service-oriented system using .NET technology
Disnix only manages service components, not infrastructure
  - Microsoft Windows, SQL server and IIS must be installed by other means
- .NET framework is not deployed by Nix/Disnix
- .NET deployment features are relatively new and untested
References

- Cygwin, http://www.cygwin.com
- StaffTracker.NET example,