Software Deployment in a Dynamic Cloud
From Device to Service Orientation in a Hospital Environment

Sander van der Burg    Eelco Dolstra    Merijn de Jonge
Eelco Visser

Delft University of Technology, EEMCS,
Department of Software Technology

Philips Research, Healthcare Systems Architecture,
Eindhoven

May 23, 2009

TU Delft    PHILIPS
Hospitals are complex organizations:
The IT infrastructure of hospitals is mostly *device oriented*. Examples:

- Dedicated workstations are needed to access patient records and images
- Workstations with powerful graphics cards are needed to render 3D objects
Device orientation

Disadvantages:
- Overcapacity due to suboptimal usage of resources
- Inflexibility in reacting to events
- Requires users to go to a device for a particular task
The medical world is changing to a *service-oriented* environment:

- Access to services is decoupled from the physical access to particular devices
- Users should be able to access data and perform computations from:
  - Where they are
  - On any device
- Example: Any device could be used to access patient records and images
Service orientation

Hospital environments are heterogeneous:

CT  MR  Nuclear medicine  X-Ray  UM
How can we support such services in a heterogeneous environment?
To support running services in a heterogeneous environment we have to treat the hospital infrastructure as a *cloud*.

- Components implementing a service are automatically deployed on machines in the cloud.
- To machines with the required capabilities and connectivity.
Requirements

- **Design** services in a way that allows them to be distributed dynamically.
  - Workstation: fat client
  - PDA: thin client
- **Deploy** each component to a selected machine in the network:
  - Usually an ad-hoc process (semi automatic with human intervention).
  - Needs to be automatic and reliable.
Approach

Existing research:

- *Disnix*. Extension for Nix that supports distributed systems.
- *SDS2*. Experimental hospital system developed by Philips Research.

Goal:

- Extend the design of services and the Disnix deployment system
Designing services

- We want to access every service from every device in a hospital environment.
- It is not feasible to perform all computations on a client or all computations on remote servers.
Designing services

Thick application
Full local deployment

- PII workstation
- Application Services
- Infrastructure Services

Thin application
PRA services server

- PII thin client
- GUI
- Application Services
- Infrastructure Services

Thick application
PRA infra services server

- PII remote WS
- GUI
- Application Services
- Infrastructure Services
- PII infra server(s)

Thin application
backoffice only servers

- PII thin client
- GUI
- Application Services
- Infrastructure Services

Philips hosted & managed PII backoffice Servers

Enterprise Services
Variants of services introduce several software deployment scenarios.

Software deployment is usually a labour intensive semi-automatic process.

Software deployment should be an automatic process capturing the deployment in models.
Deploying services

Nix Deployment System:
- Is a package manager, like RPM
- Builds packages from Nix expressions
- Stores components in isolation in a Nix store
- Upgrading is atomic
- Deals with single systems
rec {
  XMPPLLogger = derivation {
    name = "XMPPLLogger-1.0";
    src = fetchurl {
      url = http://nixos.org/.../XMPPLLogger-1.0.tar.gz;
      md5 = "de3187eac06baf5f0506c06935a1fd29";
    };
    buildInputs = [ant jdk axis2];
    builder = ./builder.sh;
  };

  DemoPortal = derivation { ... };
  stdenv = ...
  firefox = import ...
  ... # other package definitions
}
An extension to Nix to allow distributed software deployment tasks

A webservice interface which allows remote access to the Nix store and Nix profiles

Introduces three model types to model a distributed system:
- Services model
- Infrastructure model
- Distribution model

Uses a variant of the two-phase commit algorithm to allow distributed atomic commits
let pkgs = ../top-level/pkgs.nix;
in
{
    XMPPLogger = {
        name = "XMPPLogger";
        pkg = pkgs.SDS2.webservices.XMPPLogger;
        dependsOn = [EjabberdService MySQLService];
    };

    MUlogService = {
        name = "MUlogService";
        pkg = pkgs.SDS2.webservices.MUlogService;
        dependsOn = [ME2MSService];
    };

    ...
}
Infrastructure model

{
  dtk15 = {
    hostname = "dtk15";
    targetEPR = http://dtk15:8080/axis2/services/DisnixService;
    tomcatPort = 8080;
  };

  dt2d1 = {
    hostname = "dt2d1";
    targetEPR = http://dt2d1:8080/axis2/services/DisnixService;
    tomcatPort = 8080;
    mysqlPort = 3306;
  };
}
Distribution model

{services, infrastructure}:

[
    { service = services.FloorPlanService;
      target = infrastructure.dtk15; }  
    { service = services.ME2MSService;
      target = infrastructure.dtk15; }  
    { service = services.MELogService;
      target = infrastructure.dt2d1; }  
    { service = services.XMPPLogger;
      target = infrastructure.dtk15; }  
    ...

]
Conclusions and Future work

- We have outlined a possible architecture for distributed deployment of services in a cloud with some tool support.
- Nix/Disnix tools should be extended with new features to make dynamic distributed software deployment possible.
- Service and infrastructure models should take quality of service attributes into account.
- Mapping of components should be done dynamically.
**Conclusion and Future work**

- Machines should be auto-discovered instead of specified manually.
- Components implementing a service can be linked together in various ways, some of them could be sub-optimal.
  - Thin client: SOAP connection
  - Fat client: Linking libraries
- Privacy of data should be taken into account.
- During the upgrade phase, blocking the system should be avoided.