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

Sander van der Burg
Department of Software Technology
Delft University of Technology
s.vanderburg@tudelft.nl

Eelco Dolstra
Department of Software Technology
Delft University of Technology
e.dolstra@tudelft.nl

Merijn de Jonge
Philips Research
Eindhoven, The Netherlands,
merijn.de.jonge@philips.com

Eelco Visser
Department of Software Technology
Delft University of Technology
visser@acm.org

Abstract

Hospital environments are currently primarily device-oriented*: software services are installed, often manually, on specific devices. For instance, an application to view MRI scans may only be available on a limited number of workstations. The medical world is changing to a* service-oriented *environment, which means that every software service should be available on every device. However, these devices have widely varying capabilities, ranging from powerful workstations to PDAs, and high-bandwidth local machines to low-bandwidth remote machines. To support running applications in such an environment, we need to treat the hospital machines as a* cloud*, where components of the application are automatically deployed to machines in the cloud with the required capabilities and connectivity. In this paper, we suggest an architecture for applications in such a cloud, in which components are reliably and automatically deployed on the basis of a declarative model of the application using the Nix package manager.*

1. Introduction

Hospitals are complex organizations, requiring the coordination of specialists and support staff operating complex medical equipment, involving large data sets, to take care of the health of large numbers of patients. The use of information technology for diagnosis and for storage and access of patient data is of increasing importance. Hospitals are evolving into integrated information environments, where patient data, ranging from administrative records to high-density 3D images, should be accessible in real time at any place in the hospital. These data are used by people in different roles such as doctors, nurses, analysts, administrators, and patients. Each user group uses different portions of the data for different purposes and at different locations, requiring careful administration and application of access rights. Data are also accessed by medical equipment and software, for example, to improve diagnosis by combining information from multiple sources.

The infrastructure of typical hospital environments is currently mostly *device-oriented*. That is, the components implementing a workflow are statically deployed to fixed devices, which leads to overcapacity due to suboptimal usage of resources (resources are reserved for particular workflows, even if not used); inflexibility in reacting to events; a multitude of deployment and maintenance scenarios; but above all, it requires users to go to the device that supports a particular task. Because of these problems, the medical world is changing to a *service-oriented* environment in which the access to services is decoupled from the physical access to particular devices. That is, users should be able to access data and perform computations from where they are, instead of having to go to a particular device for realizing a task.

However, the information technology infrastructure of hospitals is heterogeneous and consist of thousands of electronic devices, ranging from workstations to medical equipment such as MRI scanners. These devices are connected by wired and wireless networks with complex topologies with different security and privacy policies applicable to different nodes. These devices have widely varying capabilities in terms of processing speed, graphical rendering performance, storage space and reliability, and so on.

To support running applications in such an environment, we need to treat the hospital machines as a *cloud*, where components of the application are automatically deployed to machines in the cloud with the required capabilities and connectivity. For instance, when starting a CPU-intensive application (e.g., a viewer for 3D scans) on a sufficiently powerful PC, the computation component of the application would be deployed automatically to the local machine. On the other hand, if we ran it on a underpowered PDA, this component would be deployed to a fast server sufficiently close enough to the PDA in the network topology.

This kind of cloud deployment requires two things. First, it is necessary to *design* applications in a way that allows them to be distributed across different nodes dynamically, and to create a model of applications that describe their components and the dataflows between them. These components can then be mapped onto nodes in the cloud with the appropriate quality-of-service characteristics. We give an outline of how to design components implementing a service in Section 2.

Second, given a mapping from components to machines in the cloud, it is necessary to *deploy* each component to its selected machine. Software deployment — the transition from source code to a running software system on some (collection of) devices — in a heterogeneous environment is inherently difficult. Moreover, maintaining such installations is even more difficult, because of the growing amalgam of versions and variants of the software in combination with changing requirements. The practice of software deployment of complex medical software in hospital environments is based on ad-hoc mechanisms, making software deployment a semi-automatic process requiring significant human intervention. Thus, it is essential that deployment is automatic and reliable; the deployment of a component to a node should not interfere with other applications or other versions of the component running on the node.

We have previously developed Nix [8], a package management tool that builds and deploys software packages using declarative specifications of those packages, which allows this kind of reliable, automatic deployment. In Section 3, we show a distributed extension of Nix, called Disnix, that provides the same properties in distributed environments. This is a work in progress: we have applied Disnix to a prototype hospital application, but as we discuss in Section 4, much work remains to be done on improving the modeling of distributed applications, mapping components to cloud machines automatically, and performing an evaluation of real applications under realistic conditions.

2. Designing services for distributed deployment

We want to access every service from every device in a hospital environment. It is not feasible to perform all computations on a client such as a workstation or PDA, nor is it possible to perform all computations on remote servers. Not all devices are powerful enough for all types of computations on the local machine of the user.

Therefore every component that implements a service should be able to run on the machine of the client as well as on different machines (e.g. servers) in the cloud, so that it is possible to automatically derive several *variants* of services. For instance, in a thin client variant all computation is done on a server, while in a fat client variant all computation is done on the server; hybrids, where some computation (e.g. rendering) but not all is done locally should also be possible.

Thus, it is necessary to design applications so that they can be distributed across different nodes dynamically without too much programming effort. The approach used in a prototype medical system that we used as a case study, SDS2 [3], is to design the application as a large set of components, implemented as web services that communicate via protocols such as SOAP. Typical examples of such components are a user interface, a computational layer, and a data (storage) layer. Components should not assume that they run on the same machine; e.g., they should not have dependencies on each other through the local file system. Such an architecture allows many different deployment variants, depending on the available resources: for instance, in a fat client scenario, all web services can run on the local machine, while in a thin client scenario, they can be deployed to one or more remote machines.

However, to deploy the components of such a distributed application requires substantial effort from system administrators. The software deployment process in hospital environments is usually a semi-automatic process. There are software deployment tools that assist administrators in this task, but a large part of the deployment process is done by hand on the basis of deployment documentation. A semi-automatic software deployment process requires up-to-date documentation which describes how to execute the deployment steps, and is labour-intensive. If for some reason people with the appropriate skills disappear or if the documentation is out-of-date or incomplete, the software deployment process becomes more time consuming, more error prone, risky and more expensive. This is not unlikely for many large organizations. It also becomes more difficult or even impossible to reproduce a previous deployment scenario or reason about its correctness.

To automate the deployment of distributed applications, it is necessary to describe the application in a *model* that specifies its components and runtime dependencies between components, as well as non-functional aspects such as quality-of-service requirements, e.g. that a component requires a certain level of processing speed from the machine on which it runs. Given such a model, we can then automatically deploy each component of the application to a machine in the cloud that has the desired characteristics.

3. Deploying services in a cloud

Thus, we need *model-driven software deployment*: given a model of a software system that describes its components and dependencies, a deployment tool must be able to correctly install these components to the appropriate machines. It must guarantee that such deployment is reliable and deterministic: for instance, previous components installations on a target machine should not affect the success of the installation of the new component.

Alas, existing deployment tools are generally not up to this task. For instance, Unix package managers such as RPM [11] cannot guarantee correct deployment due to an inability to support multiple co-existing versions or variants of a component. E.g., if two users were to run two different applications in the cloud, involving two different versions of some component X, then we would be in trouble if these two versions were deployed to the same machine in the cloud: one version (typically the newer) would overwrite the other. Unless versions are perfectly backwards compatible, this will break applications. Furthermore, package managers typically describe how to build and deploy components but not *compositions* of components. It is possible to specify what libraries, compilers and so on are needed to build or run a component, but such specifications are *nominal* (i.e., they state dependencies such as “package named X with version greater than N”). Such specifications are inexact and do not allow the deployment system to automatically reproduce a configuration on a remote machine.

We have previously developed the purely functional package manager Nix, which solves these problems for local deployment (i.e., on a single machine). We are developing an extension called Disnix that supports distributed deployment on the basis of the kinds of models described above.

3.1. Nix

The Nix package manager [8, 5] builds and stores packages in a purely functional manner: packages are built from source using descriptions in a simple purely functional language, and build results are immutable. This ensures that multiple versions of a package can coexist on a system, that upgrades can be performed in an atomic manner, and that it is possible to roll back to previous configurations. It forms the basis for NixOS (`http://nixos.org/`), a Linux distribution with a purely functional configuration management model [7].

Rather than storing packages in global namespaces such as the /usr/bin directory on Unix, each package is stored in isolation of other packages in a subdirectory of the *Nix store*, the directory /nix/store. For instance, the directory /nix/store/pz3g9yq2x2ql...-firefox-2.0.0.16 contains a particular instance of Mozilla Firefox. The string pz3g9yq2x2ql... is a 160-bit cryptographic hash of the inputs to the build process of the package: its source code, the build script, dependencies such as the C compiler, etc. This scheme ensures that different versions or variant builds of a package do not interfere with each other in the file system. For instance, a change to the source code (e.g., due to a new version) or to a dependency (e.g., using another version of the C compiler) will cause a change to the hash, and so the resulting package will end up under a different path name in the Nix store.

```
rec {
  HelloService = derivation {
    name = "HelloService-1.0";
    src = fetchurl {
      url = http://nixos.org/.../HelloService.tar.gz;
      md5 = "de3187eac06baf5f0506c06935a1fd29";
    };
    buildInputs = [ant jdk axis2];
    buildCommand = ''
      tar xf $src
      cd HelloService
      ant generate.service.aar
      mkdir -p $out/webapps/axis2/WEB-INF/services
      cp HelloService.aar $out/webapps/axis2/WEB-INF/services
    '';
  };

  HelloWorldService = derivation { ... };
  stdenv = ...
  firefox = import ...
  ... # other package definitions
}
```

Figure 1. pkgs.nix**, an example of a Nix expression.**

Nix builds packages from a description in a purely functional language called the *Nix expression language*. Nix expressions describe dependency graphs of build actions, called *derivations*, that each build a path in the Nix store. Figure 1 shows a Nix expression defining a number of derivations bound to variables that can refer to each other. For instance, the value of the variable HelloService is a derivation that builds an Apache Axis web service. The function derivation is the primitive operation that produces a build action from a set of *attributes*, such as the name of the package, its source (which is produced by the build action returned by the call to the function fetchurl), a shell script that performs the actual build action (buildCommand), and so on. All attributes are passed through environment variables to the build command, e.g., the variable src will contain the path in the Nix store of the downloaded sources. The environment variable out contains the target path in the store for the package, e.g., /nix/store/*hash*-HelloService-1.0.

The user can install packages using a command such as

```
$ nix-env -f pkgs.nix -i firefox
```

which builds the derivation resulting from the evaluation of the firefox variable in Figure 1, along with all its dependencies, resulting in many new packages in the Nix store. To upgrade Firefox, the user updates the Nix expression in question (which is typically automated through a number of mechanisms, such as an automated download mechanism), and performs

```
$ nix-env -f pkgs.nix -u firefox
```

which builds the new Firefox package and makes it available in the user's PATH. The evaluation of the firefox value will lead to the build of a new Firefox version in the Nix store, which will not overwrite any previously installed versions thanks to the hashing scheme. Thus, it is possible to roll back to the previous version. Such upgrades are also transactional: the user will never observe part of the old and part of the new version at the same time.

3.2. Disnix

Disnix is an extension to the Nix deployment system that supports distributed software deployment operations [14]. The Disnix deployment system contains an interface which allows another process or user to access the Nix store and Nix user profiles remotely through a distributed communication protocol, e.g. SOAP. The Disnix system also consists of tools that support distributed installing, upgrading, uninstalling and other deployment activities by calling these interfaces on the nodes in the distributed system.

To deploy packages on a single computer Nix just needs to know the compositions of each package and how to build them. Disnix needs some additional information for deploying services on multiple computers. Therefore three models are introduced: a *services model*, an *infrastructure model* and a *distribution model*.

The *services model* describes the services that can be distributed across computers in the network. This is a simple example of the kind of model of a distributed application that we described in Section 2. It does not describe any quality of service requirements. Each service is essentially a package, except that it has an extra property called *dependsOn* which describes the *inter-dependencies* on other services. Figure 2 shows a Nix expression that describes a model for two services: HelloService, which has no dependencies, and HelloWorldService, which depends on the former.

We also need to specify what machines are available in the cloud. The *infrastructure model* is a Nix expression that describes certain attributes about each computer in the network. Figure 3 shows a simple example of a network with two computers; again, it does not specify QoS attributes. In the model we describe the hostname and the target endpoint references (targetEPR) of the Disnix interface. The targetEPR is a URL that points to the Disnix web service that can execute operations on the remote Nix store. This is needed to deploy components to the remote machine and execute them.

```
rec {
    pkgs = import ./pkgs.nix;

    HelloService = {
        pkg = pkgs.HelloService;
        dependsOn = [];
    };
    HelloWorldService = {
        pkg = pkgs.HelloWorldService;
        dependsOn = [ HelloService ];
    };
}
```

Figure 2. services.nix

```
{   itchy = {
        hostname = "itchy";
        targetEPR = http://itchy/.../DisnixService;
    };
    scratchy = {
        hostname = "scratchy";
        targetEPR = http://scratchy/.../DisnixService;
    };
}
```

Figure 3. infrastructure.nix

Finally, the *distribution model* is a Nix expression that connects the services and infrastructure model, mapping services to computers. It contains a function which takes a services and infrastructure model as its inputs and returns a list of pairs. Each pair describes what service should be distributed to which computer in the network. Figure 4 shows an example. It is also possible to specify a specific component multiple times in the model. In this case the same variant of the service will be deployed on multiple machines. It is also possible to use multiple variants of a specific component by defining their compositions in the *services* model.

With these three models, we can now deploy a distributed application. For instance, given the three example models, Disnix will build the HelloService and HelloWorldService components as well as their dependencies on the local machine, copy the *closure* of each component (that is, the Nix store paths of the component and its runtime dependencies) to the Nix store of the machine indicated by

```
{services, infrastructure}:

[   { service = services.HelloService;
      target = infrastructure.itchy; }
    { service = services.HelloWorldService;
      target = infrastructure.scratchy; }
]
```

Figure 4. distribution.nix

the distribution model, and then start each component. For instance, the closure of the HelloWorldService component will be copied to the machine scratchy and then started.

When performing an upgrade from a previous version of a distributed application, Disnix uses a variant of the *two-phase commit protocol* [13] to make the transition from the current deployment state to the new deployment state in the distributed system an atomic operation. Disnix also *blocks* access to services while the commit phase is in progress. In the commit phase, the old version of the service is stopped, the new version is started, and the blocked connections are unblocked and forwarded to the new version of the service. From this point, all connections will be to the new versions of the services in the system. There is no time window in which one can simultaneously reach the old version of some service *and* the new version of another [14].

4. Future work

Above we have outlined an architecture for deployment of distributed applications in clouds, along with tool support to deploy components reliably to machines in the cloud. We have applied it to a prototype research application called SDS2 [3]. However, while the Nix deployment system and the Disnix extension provide a solid foundation, much work remains to be done.

Most importantly, the models of services (Figure 2) and machines (Figure 3) should take quality-of-service properties into account. For instance, each component should declare the QoS properties it requires from the machine on which it runs, e.g. processing speed or storage capacity. Components should also declare the required bandwidth and latency of dataflows between components, and the model must address persistent storage aspects of components. Likewise, each machine should declare the QoS properties that it provides. Since a hospital environment is heterogeneous, we also have to capture the topology of the network and the properties of computer systems and service components, taking into account factors such as network bandwidth, available memory, and resource utilization.

Moreover, the mapping from components to machines should be performed automatically. Currently the distribution of services to machines in the network is a *static* process (e.g. by defining a mapping such as in Figure 4). Instead, it should should be computed automatically, taking into account the defined QoS properties, network topology and the current load on each machine. The Disnix toolset currently contains a very simple generator that generates a distribution model from services and infrastructure models by using a round-robin scheduling method. It does not take quality of service into account.

Given the dynamic nature of a hospital environment, machines should be autodiscovered, e.g. via a protocol such as Zeroconf. Also, the network topology and configuration is dynamic; the load on individual nodes changes over time; and not all machines are available continuously, as some may break, others need maintenance, new machines are added, and so on. Such events may prompt a redistribution of service components, and should not require a manual update of the deployment configurations. Redeploying running components automatically to other machines places additional constraints on components in order to make such upgrades atomic [14].

Components implementing a service can be *linked* in various ways. For instance, in a thin client scenario, where components may reside on different servers, components communicate through network protocols such as SOAP. On the other hand, in a fat client scenario, where all the computations are done on the client machine, this is suboptimal: we would rather link components together as libraries, communicating using intra-process function calls. Thus we should abstract over the communication mechanism, and use a generator to produce code for components implementing a service described in the model with different interfaces depending on how a component should be coupled, for instance as a shared library or as a web service. The generator is also used to integrate with the deployment system, so that it can distribute components to the right machines in the cloud.

Another consideration is the privacy of data. Where in the old, device-oriented situation data were naturally confined to a few devices, they may now be transported over long distances, requiring explicit access control. This may not be restricted to simple granting or denial of access. Sometimes data may be partially accessible to certain parties. For example, medical research may need to examine the results of tests, but should not be able to track this to individual patients, thus requiring anonymisation of data. Such scenarios require different variants of services to be deployed.

5. Related work

Software deployment has traditionally been done in a relatively *ad hoc* manner. There are many deployment tools (e.g. RPM [11]) that typically have various fundamental limitations with respect to correctness of deployment: e.g., they cannot guarantee that an upgrade of one application cannot affect others because of shared dependencies. The Nix deployment system solves many of the problems of mainstream deployment solutions, but it was not designed for distributed deployment, and it does not take non-functional requirements (such as performance) into account. The Disnix approach outlined here is thus a natural continuation of our previous work.

Deployment in the context of grid computing [10] is discussed in [12], which notes the difficulty in deploying software to grid nodes — an often manual process — as an obstacle to wider use. For instance, in the Globus Toolkit [9], users must provide XML files that describe what files should be copied to remote machines to perform a job. Similarly, Apple's Xgrid [2] by default simply copies the job's executable and the contents of the current directory to the remote machine. Thus, the user is burdened with figuring out the code dependencies of the job. In [6], we showed that Nix's notion of deploying *closures* to remote machines can make such tasks much simpler.

SOA deployment technologies often focus primarily on composition and discovery of services (for instance, see [15] for a comparison of several composition languages), and ignore building and installing services on computer systems. There seems to be little research activity in the area of service deployment, an exception being [1]. Deployment of components in distributed environments is addressed more frequently [4].

6. Conclusion

In this paper we have described our vision of moving from device orientation in hospital environments to service orientation. As it is not possible in general in such environments to perform all computations on a client, nor desirable to perform all computations on remote servers, this requires treating the systems in such an environment as a cloud to which distributed applications must be deployed automatically. This requires a component-based approach to software development and models to describe components and their dependencies. These can then be mapped dynamically onto machines in the cloud with the required quality-of-service properties. This requires a reliable deployment system, which is what the Nix package manager provides.

We have to extend Disnix with new features that make the distribution of services dynamic and deal with heterogeneous infrastructures, so that we can distribute services in a cloud in an optimal manner. Furthermore, we have to investigate methods and techniques for designing applications so that they can distributed across nodes in a cloud.

Acknowledgments This research is supported by Philips Healthcare and NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*.

References

[1] S. R. Amendolia, F. Estrella, C. del Frate, J. Galvez, W. Hassan, T. Hauer, D. Manset, R. McClatchey, M. Odeh, D. Rogulin, T. Solomonides, and R. Warren. Deployment of a grid-based medical imaging application, Dec. 2004.

[2] Apple Inc. Xgrid. `http://www.apple.com/macosx/features/xgrid/`, 2004.

[3] M. de Jonge, W. van der Linden, and R. Willems. eServices for hospital equipment. In B. Krämer, K.-J. Lin, and P. Narasimhan, editors, *Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*, pages 391–397, Sept. 2007.

[4] A. Dearle and S. Eisenbach, editors. *Component deployment: Third international working conference, CD 2005, Grenoble, France, November 28–29, 2005, proceedings*, volume 3798 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[5] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht University, The Netherlands, Jan. 2006.

[6] E. Dolstra, M. Bravenboer, and E. Visser. Service configuration management. In J. E. James Whitehead and A. P. Dahlqvist, editors, *12th International Workshop on Software Configuration Management (SCM-12)*, pages 83–98, Lisbon, Portugal, September 2005. ACM.

[7] E. Dolstra and A. Löh. NixOS: A purely functional Linux distribution. In *ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming*. ACM Press, Sept. 2008.

[8] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

[9] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, number 3779 in Lecture Notes in Computer Science, pages 2–13. Springer, 2005.

[10] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Nov. 1998.

[11] E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley & Sons, 2003.

[12] J. M. Schopf and B. Nitzberg. Grids: The top ten questions. *Scientific Programming*, 10(2):103–111, 2002.

[13] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. In *Concurrency control and reliability in distributed systems*, pages 295–317, New York, NY, USA, 1987. Van Nostrand Reinhold Co.

[14] S. van der Burg, E. Dolstra, and M. de Jonge. Atomic upgrading of distributed systems. In T. Dumitraş, D. Dig, and I. Neamtiu, editors, *First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp)*, pages 1–5. ACM Press, Oct. 2008.

[15] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Analysis of web services composition languages: The case of BPEL4WS. In *Conceptual Modeling - ER 2003*, pages 200–215, Oct. 2003.