A Self-Adaptive Deployment Framework for Service-Oriented Systems

Sander van der Burg
Delft University of Technology
The Netherlands
s.vanderburg@tudelft.nl

Eelco Dolstra
Delft University of Technology
The Netherlands
e.dolstra@tudelft.nl

ABSTRACT

Deploying components of a service-oriented system in a network of machines is often a complex and labourious process. Usually the environment in which such systems are deployed is *dynamic*: any machine in the network may crash, network links may temporarily fail, and so on. Such events may render the system partially or completely unusable. If an event occurs, it is difficult and expensive to *redeploy* the system to the take the new circumstances into account. In this paper we present a self-adaptive deployment framework built on top of *Disnix*, a model-driven distributed deployment tool for service-oriented systems. This framework dynamically discovers machines in the network and generates a mapping of components to machines based on non-functional properties. Disnix is then invoked to automatically, reliably and efficiently redeploy the system.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Management, Design, Experimentation

Keywords

Software Deployment, Service-Oriented Systems

1. INTRODUCTION

Service-oriented systems are composed of distributable components (*services*) using various technologies, such as web services, databases, web applications and batch processes, working together to achieve a common goal [17]. Service-oriented systems are usually deployed in networks of machines (such as a cloud environment) having various capabilities and properties. For example, a machine assigned for storage may not be suitable for running a component doing complex calculations or image rendering.

SEAMS '11, May 23–24, 2011, Waikiki, Honolulu, HI, USA

In networks various *events* may occur. A machine may crash and disappear from the network, or a new machine may be added. Moreover, a capability of a machine could change, such as an increase of memory. Some of these events may partially or completely break a system or make the current deployment suboptimal.

In order to take new circumstances into account after an event, a system must be *redeployed*. Deployment processes of service-oriented systems are usually difficult and expensive. For instance, they may require system administrators to perform manual actions such as installing components or editing configuration files. In many cases, it is a lot of effort to build components, transfer the components and all dependencies to the right machines and to activate the system. Upgrading is even more difficult as this may completely break the system. Furthermore, finding a suitable mapping of services to machines is a non-trivial process, since non-functional properties of the domain (such as quality-of-service requirements) must be supported. These factors cause a large degree of *inflexibility* in reacting to events.

The contribution of this paper is a self-adaptive deployment extension built on top of *Disnix* [19], a distributed deployment tool for service-oriented systems. This framework continuously discovers machines and their capabilities in a network. In case of an event, the system is automatically, reliably and efficiently redeployed to take the new circumstances into account. A quality-of-service (QoS) policy model can be specified to find a suitable mapping of services to machines dealing with desirable deployment constraints from the domain. We have applied this deployment extension to several case studies, including an industrial case study from Philips Research.

In contrast to other self-adaptive approaches dealing with events in distributed environments, Disnix redeploys a system instead of changing properties of already deployed components, such as its behaviour and connections. The advantages of this approach are that various types of services are supported (which may not all have self-adaptive properties) and that we do not require components to use a specific component technology to make them self-adaptive.

2. MOTIVATION

We have applied the dynamic Disnix extension to several use cases, including two custom developed systems, an open-source case study and an industrial case study, each implemented using various technologies, serving different goals and having different requirements. We will use these throughout this paper to motivate and explain our work.

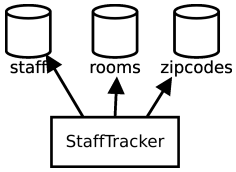

Figure 1: Architecture of the StaffTracker (PHP/MySQL version)

2.1 StaffTracker (PHP/MySQL version)

First, we present two toy systems that represent two different styles of distributed systems. Both implement the same application: allowing users to store and query information about staff members of a university, such as names, IP addresses and room numbers. From a room number of a staff member, a zip code can be determined; from a zip code, the system can determine the street and city in which the staff member is located. The first variant uses a web interface written in PHP that accesses data stored in different MySQL databases.

Figure 1 shows the architecture of this variant of the StaffTracker system. Each object in the figure denotes a distributable component, capable of running on a separate machine in the network. The arrows denote an *inter-dependency* relationship, which are dependencies between distributable components. The StaffTracker system is composed of various MySQL databases (denoted by a database icon) storing a dataset, and a web application front-end implemented in PHP (denoted by a rectangle) allowing end users to query and modify staff members.

In order to deploy this system several technical requirements must be met. First, all MySQL databases must be deployed on a machine running a MySQL DBMS. Secondly, the web application front-end must be deployed on a machine running an Apache webserver instance.

Various events may occur in a network of machines running this system. A machine may crash, making it impossible to access a specific dataset. Moreover, if the machine running the web application front-end crashes the system becomes completely inaccessible. It is also possible that a new machine is added, such as a database server, which may offer additional disk space. In such cases, the system must be redeployed to take the new circumstances into account.

To redeploy this system, a system administrator must install databases on a MySQL server or web applications on an Apache webserver. Moreover, he has to adapt the configuration of the web application front-end, so that a database from a different server is used. Furthermore, these steps must be performed in the right order. If, for instance, the system administrator activates the web application front-end, before all databases are activated specific features may be inaccessible to end users.

2.2 StaffTracker (Web services version)

The second variant, shown in Figure 2, represents a SOA-style system based on separate *web services* that each provide access to a different database. It also has a Geolocation web service to determine the location of a staff members, based on his or her IP address. The web services and web application components of this variant of the StaffTracker are

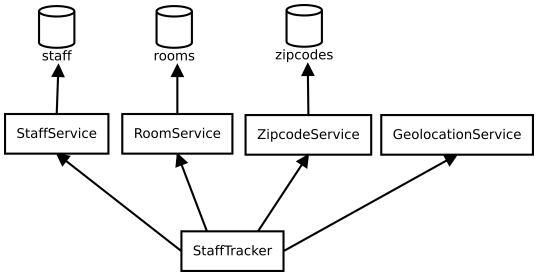

Figure 2: Architecture of the StaffTracker (Web services version)

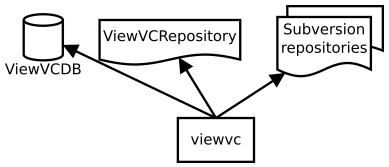

Figure 3: Architecture of ViewVC

implemented in Java and require Apache Tomcat (`http://tomcat.apache.org`) for web application hosting and management.

In order to deploy this system correctly, as in the previous example, we must deploy MySQL databases on machines running a MySQL DBMS (similar to the previous use case), but we must also deploy web service and web application components on machines running Apache Tomcat.

In case of an event, redeployment takes even more effort than the previous example. If the location of a database or web service changes, a system administrator has to adapt all the configuration files of all dependent services. Moreover, the system administrator has to repackage the web service and web applications, transfer them to the right machines in the network and deactivate and activate them in the right order (otherwise features may become inaccessible).

2.3 ViewVC

ViewVC (`http://viewvc.tigris.org`) is a free and open-source web-based Subversion and CVS repository viewer implemented in Python. ViewVC is able to connect to remote Subversion repositories in a network. It can optionally use a MySQL database to query commit records used for development analysis tasks. The architecture is shown in Figure 3.

As in the previous examples, we must ensure that the web application front-end is deployed on a machine running an Apache webserver and the database on a machine running a MySQL DBMS. We must also ensure that Subversion repositories are deployed on machines running svnserve, which provides remote access to a Subversion repository through the SVN protocol.

2.4 SDS2

SDS2 [8] is an industrial case study (described in a previous paper about Disnix [19]). It is a service-oriented architecture designed by Philips Research for Asset Tracking and Utilisation in hospitals. Medical devices generate status and location events. SDS2 captures these events, stores them

in databases and transforms and synthesizes these implicit datasets into something more useful, such as workflow analysis reports and utilisation summaries. The workflow analysis reports, summaries and live data of medical equipment can be accessed by end users through various web applications front-ends.

SDS2 is composed of 18 services. Data is stored in various MySQL databases distributed across various locations. All datasets and transformation components are accessible through web services. Messaging of status and location events is done by sending XMPP messages by using an Ejabberd server. SDS2 offers three separate web application front-ends. Web service components and batch processes are implemented in the Java programming language. The web application front-ends are implemented using the Google Web Toolkit (GWT).

Apart from the technical requirements for deployment, such as mapping services with a specific type to machines capable of running them, there are also non-functional requirements which must be supported. For example, the datasets containing events logs must be stored inside a hospital environment for performance and privacy reasons. The services generating workflow analysis results and summaries must be deployed inside the Philips enterprise environment, since Philips has a more powerful infrastructure and may want to use generated data for other purposes.

In order to redeploy SDS2, nearly every service must be adapted (due to the fact that every service uses a global configuration file containing all locations of every service), repackaged, transferred to the right machines in the network and deactivated and activated in the right order (a process which normally takes several hours to perform manually).

3. DISNIX

We previously have developed *Disnix* [19, 20], a distributed deployment tool built on top of the Nix package manager [10] to deal with the complexity of the deployment process of service-oriented systems. Disnix offers various advantages compared to related tooling, such as a declarative language to capture various properties of distributed systems, the ability to support various types of components, to safely install components next to other components without interference (which ensures that if a machine crashes during an upgrade, the system is not left in an inconsistent state), and to efficiently and reliably upgrade a distributed system in a way that takes all dependencies into account, activating and deactivating components in the right order.

Existing deployment tooling is usually designed for a particular class of components (e.g. Enterprise Java Beans [2]) or specific environments (such as Grid Computing [5]). Some general approaches have been developed (such as the Software Dock [12]) but they lack certain desired properties, such as the ability to safely store components next to each other without interference and to atomically upgrade a distributed system. This usually results in relatively long downtimes and greater possibility of errors during upgrades.

3.1 Overview

Figure 4 gives an overview of the Disnix concepts. On the left side of the picture three types of models are shown, which are written in the Nix expression language, a domain-specific language to capture build actions. The *services model* is used to describe the services of which the service-oriented system is composed, their properties and *inter-dependencies* (which are dependencies on other services which may be located on different systems in the network). The *infrastructure model* captures machines in the network, how they can be reached so that deployment steps can be performed remotely, and other relevant properties and capabilities. The *distribution model* maps services defined in the services model to target machines defined in the infrastructure model.

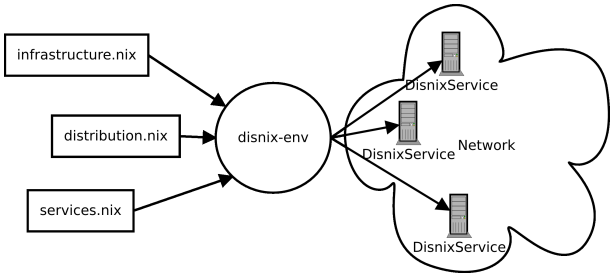

Figure 4: Disnix overview

On the right side of the picture the network is shown. Each target machine must run the *DisnixService* running so that the coordinator machine can perform remote deployment operations.

By writing instances of the models shown in Figure 4 and by running the following command:

```
disnix-env -s services.nix -i infrastructure.nix \
  -d distribution.nix
```

Disnix builds all the services (including all intra-dependencies) defined in the services model from source code, then transfers the services and all their dependencies to the target machines. Finally, it activates the services in the right order. By adapting the models and by running disnix-env again, an upgrade is performed instead of a full installation. In this case only the changed components are built and transferred to the target machines, obsolete services are deactivated, and new services are activated. This makes the upgrade process reliable and efficient.

3.2 Services Model

Figure 5 shows a partial services model for the StaffTracker application described in Section 2.2, a trivial service-oriented system included in the Disnix example repository. The model is essentially a function that takes two arguments (defined at 1). The distribution argument refers to the distribution model which is shown later in this paper. The system attribute specifies a platform identifier, such as i686-linux, which is used to build a service for a 32-bit Linux platform.

The remainder of the model is an attribute set in the form: $\{arg_1 = value_1; arg_2 = value_2; ...; arg_n = value_n;\}$. The rec keyword defined at 3 allows attributes to refer to each other. Each attribute, (such as 4) defines a distributable component (or service) part of the architecture of the StaffTracker system. These services correspond to the components shown in Figure 2.

Every service attribute refers to a set of attributes required for deployment. 5 defines a name for the service. 6 refers to a function which builds and configures the component and all its intra-dependencies (such as compilers and

```
{distribution, system}: 1

let pkgs = import ../top-level/all-packages.nix { 2
  inherit system;
}; in
rec { 3
### Databases
  rooms = { 4
    name = "rooms";
    pkg = pkgs.rooms;
    dependsOn = {};
    type = "mysql-database";
    requiredZone = "private";
  };
  ...
### Web services
  RoomService = {
    name = "RoomService"; 5
    pkg = pkgs.RoomService; 6
    dependsOn = { 7
      inherit rooms;
    };
    type = "tomcat-webapplication"; 8
    requiredZone = "private"; 9
  };
  ...
### Web applications
  StaffTracker = {
    name = "StaffTracker";
    pkg = pkgs.StaffTracker;
    dependsOn = {
      inherit GeolocationService RoomService;
      inherit StaffService ZipcodeService;
    };
    type = "tomcat-webapplication";
    requiredZone = "public";
  };
}
```

Figure 5: services.nix: Partial services model for the StaffTracker

libraries) and inter-dependencies. The function that actually builds the component is defined in the file included at 2, which is not shown here. More details about these models are described in [19].

The inter-dependencies of the service are defined at 7. The RoomService requires access to the rooms database service, storing the actual room numbers in a MySQL database. The inherit rooms; statement copies the value of the function argument rooms from the lexical scope. A different inter-dependency composition can be specified by writing, e.g., rooms = myotherrooms;. The inter-dependencies correspond to the arrows defined in Figure 2.

The type attribute 8 defines the type of the service. Because services can be implemented in many different ways, there is no generic way to activate or deactivate them. The type attribute is used for activation of the service. For example, by specifying tomcat-webapplication, a WAR archive is hot-deployed on an Apache Tomcat instance on the target machine. Other service attributes can be freely chosen. For example, 9 is a custom attribute, which is used for this system to specify that this component must be deployed in a particular zone (in this case, a private zone that restricts end-users from accessing this service). This attribute is not directly used by Disnix for deployment, but can be used for purposes such as expressing QoS requirements, as we will see in Section 5.

```
{
  test1 = {
    hostname = "test1.testdomain.net";
    tomcatPort = 8080;
    system = "i686-linux";
    supportedTypes = [ "tomcat-webapplication" "process" ];
    zone = "public";
  };
  test2 = { 10
    hostname = "test2.testdomain.net"; 11
    tomcatPort = 8080; 12
    mysqlPort = 3306;
    mysqlUsername = "user";
    mysqlPassword = "secret";
    system = "x86_64-linux"; 13
    supportedTypes = [
      "tomcat-webapplication" "mysql-database" "process"
    ]; 14
    zone = "private"; 15
  };
}
```

Figure 6: An infrastructure.nix model for the StaffTracker

```
{infrastructure}: 16

{
  rooms = [ infrastructure.test2 ]; 17
  RoomService = [
    infrastructure.test1 infrastructure.test2
  ]; 18
  StaffTracker = [ infrastructure.test2 ];
  ...
}
```

Figure 7: Partial distribution.nix model for the StaffTracker

3.3 Infrastructure Model

Figure 6 shows an infrastructure model. In essence, an infrastructure model is a set in which every attribute defines a machine in the network, such as 10. For each machine, a number of attributes are specified. For instance, 11 defines how the machine can be reached to perform remote deployment operations. The system attribute 13 defines the architecture of the system, so that Disnix can build the service for that particular platform. The supportedTypes attribute is a list of strings indicating what types of services can be deployed and run. For instance, the machine test2 can run MySQL databases, Apache Tomcat web applications and generic processes 14. Other attributes can be freely defined, such as the port number of Apache Tomcat 12 and the zone in which the machine resides 15. These properties are used by activation scripts to activate a service on that machine and can be used for other purposes shown later in this paper.

3.4 Distribution Model

Figure 7 shows a partial distribution model, mapping services onto machines. This model is a function that takes the infrastructure model defined in Figure 6 as a parameter

[16]. The remainder of the model is an attribute set in which each attribute represents a service defined in the services model. Each attribute refers to a list of targets defined in the infrastructure model.

For example, the rooms service must be deployed on machine test2 defined in the infrastructure model [17]. By specifying multiple machines in a list, redundant services can be deployed for fallback or load-balancing [18].

3.5 Implementation

Disnix uses the three models described in the previous sections to deploy a service-oriented system. First all services and all required dependencies are *built* from source code for the given target machine. Every service is stored under a unique path in a special directory called the *Nix store*, such as /nix/store/24pwpsg4da1...-StaffTracker. The first part of the filename: 24pwpsg4da1... represents a cryptographic hash code derived from all build-time dependencies of the component. This makes it safe to store multiple versions or variants next to each other and to upgrade a system without interference. For example, if a different compiler or library is used to build the component, this hash code changes. Moreover, it also serves as a *cache* for the build functions. If a component with the same dependencies has been built before, the result from the Nix store is used instead of performing the build again.

After building the services, every service and its intra-dependencies are *transferred* to target machines in the network. Intra-dependencies of a component are detected by scanning for occurrences of hash codes inside a component. If a component with a hash code already exists on a target machine, it does not have to be transferred again.

Finally, the *transition* phase starts. In this phase obsolete services are deactivated and new services are activated. The order of deactivation and activation is derived from the inter-dependencies specified in the service model. Disnix ensures dependency relationship are satisfied at all times. During this phase a user may observe that the system is changing, but Disnix makes this time-window as small as possible. Optionally, a proxy can be used to queue connections during this phase, making the transition process truly atomic. More details about the deployment process can be found in [19].

4. SELF-ADAPTIVE DEPLOYMENT

Disnix is a good basis for distributed deployment, because it can safely install components without interference and reliably install or upgrade a distributed system. Moreover, it makes the time of the transition phase (in which the system is inconsistent) as small as possible. All these features significantly reduce the redeployment effort of a system. In order to react automatically to events occurring in the network, we have extended the Disnix toolset by providing automatic generation of the infrastructure and distribution models.

Figure 8 shows an extended architecture of Disnix to support self-adaptive deployment. Rectangles represent objects that are manipulated by the tools. The ovals represent a tool from the toolset performing a step in the deployment process.

In this architecture several new tools are visible. The *infrastructure generator* uses a discovery service to detect machines in the network and their capabilities and properties. An infrastructure model is generated from these discovered properties. The *infrastructure augmenter* adds additional infrastructure properties to the generated infrastructure model that are not announced by the machines. (Some properties are not desirable to announce because they are privacy-sensitive, such as authentication credentials). The *distribution generator* maps services in the services model to targets defined in the generated and augmented infrastructure model. A quality of service (QoS) model is used to specify a *policy* describing how this mapping should be performed, based on quality attributes specified in the services and infrastructure model.

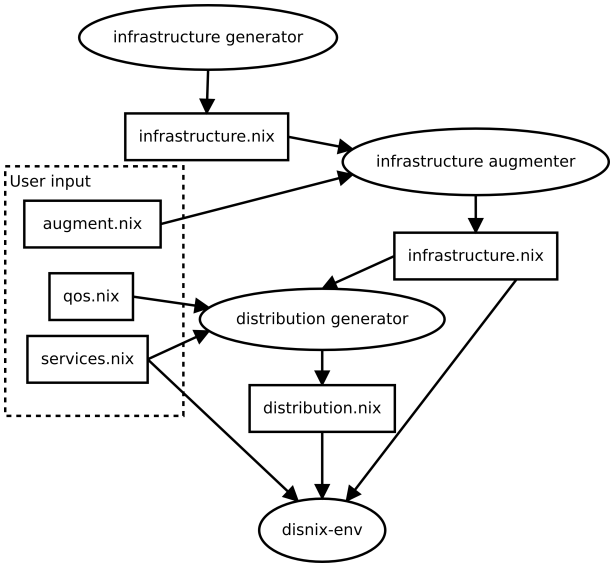

Figure 8: A self-adaptive deployment architecture for Disnix

In this framework, a user provides three models as input parameters. The service model, as we have seen in Section 3.2, captures all the services constituting the system. The augmentation model is used to add infrastructure properties to the infrastructure model, such as authentication credentials. The quality-of-service model (QoS) is used to describe how services can be distributed to target machines, based on QoS properties defined in the services and infrastructure models.

After the infrastructure and distribution models have been generated, they are passed to the disnix-env tool along with the services model. disnix-env then performs the actual deployment and the system is upgraded according to the new distribution. By continuously executing this framework when events occur, a system becomes self-adaptive.

4.1 Infrastructure Generator

We have implemented an infrastructure generator based on Avahi, a multicast DNS (mDNS) and DNS service discovery (DNS-SD) implementation [1]. We have chosen Avahi for practical reasons, because it is light-weight and included in many modern Linux distributions. Although we only have a DNS-SD-based infrastructure generator, the architecture of the self-adaptive deployment extension is not limited to a particular discovery protocol. The infrastructure generator is a process that can be relatively easily replaced by a generator using a different discovery protocol, such as SSDP.

Every target machine in the network multicasts properties and capabilities in DNS records through DNS-SD. Some infrastructure properties are published by default, such as the hostname which is required for the coordinator machine to perform remote deployment steps, system specifying the operating system and architecture, and supportedTypes which specifies the types of services which can be activated on the machine, such as mysql-database and tomcat-webapplication. The list of supported types is configured by the installation of the *DisnixService*, which for instance provides a mysql-database activation script if the configuration script detects a MySQL instance running on the target machine. Moreover, some hardware capabilities are published, such as mem which publishes the amount of RAM available. A system administrator can also publish custom properties through the same service as DNS records, such as zone representing the location in which the machine is based.

The infrastructure generator captures the DNS-SD properties announced by the target machines and uses it to generate an infrastructure model. The generation step is quite trivial, because DNS records consist of $key = value$ pairs that map onto Nix attribute sets in a straight-forward manner.

4.2 Distribution Generator

Given the automatically discovered infrastructure model, Disnix must now generate an distribution model that maps services onto machines. Automatically generating a suitable distribution is not a trivial task. In some cases it is impossible to find an optimal distribution, because optimising a particular attribute may reduce the quality of another important quality attribute, a phenomenon called *Pareto optimality* in multidimensional optimisation [15]. Some algorithms for finding a suitable distribution are NP-hard, which makes it infeasible to find an optimal distribution for large systems.

Moreover, in many cases it is desired to implement constraints from the domain, which must be reflected in the distribution of services to machines. These constraints are largely context-specific. For example, constraints in the medical domain (such as privacy) are not always as important in other domains in which performance may be much more important. Also, the notion of certain quality attributes may be different in each domain.

Because of all these issues, there is no silver bullet that always generates a suitable distribution for a service-oriented system. Therefore, a developer or system administrator must specify a suitable deployment strategy in a quality of service model to deal with non-functional requirements from the domain. The Disnix toolset contains various functions that assist users with specifying deployment strategies. Apart from a number of included functions, custom algorithms and tooling can be easily integrated into the toolset to support desirable distribution strategies.

The structure of a quality-of-service model is shown in Figure 9. A QoS model is a function 19 taking the several arguments. The services argument represents the services model and the infrastructure argument represents the infrastructure model. The initialDistribution represents an distribution model which maps services to candidate targets. Normally, every target defined in the infrastructure model is considered a candidate host for every service. The previousDistribution argument represents the distribution model

```
{ services, infrastructure
, initialDistribution, previousDistribution, filters}: 19

filters.filterB {
  inherit services infrastructure;
  ...

  distribution = filters.filterA {
    inherit services infrastructure;
    distribution = initialDistribution;
    ...
  };
}
```

Figure 9: qos.nix: A quality-of-service model implementing a policy

of the previous deployment (or null in case of an initial deployment). This attribute set can be used to reflect about the upgrade effort or to keep specific services on the same machines. The filters attribute represents an attribute set containing various functions, which filter and divide services defined in the initial distribution model. In the body of the model, a developer must filter and transform the initialDistribution attribute set into a suitable distribution, by applying various functions defined in filters. Every function takes a distribution model as input argument and optionally uses the services and infrastructure models. The result of a function is a new distribution model which can be passed to another filter function or used for the actual deployment. The process of composing filter functions is repeated until a desirable distribution is achieved.

5. IMPLEMENTING A QUALITY OF SERVICE MODEL

The QoS policy specifications in Disnix consist of two parts, which are inspired by [14, 13]. First, a *candidate host selection* phase is performed, in which for each service suitable candidate hosts are determined. This phase sets the deployment boundaries for each service. After a suitable candidate host selection is generated, the *division phase* is performed in which services are allocated to the candidate hosts. The dynamic deployment framework offers various functions defined in the Nix expression language that can be composed together to generate a desirable distribution strategy dealing with non-functional properties from the domain.

5.1 Candidate Host Selection Phase

Before we can reliably divide services on machines in a network, we must first know on which machines a service is deployable. There are many constraints which render a service undeployable on a particular machine. First, there are *technical constraints*. For example, in order to deploy a MySQL database on a machine in the network, a running MySQL DBMS is required. Moreover, a component may also have other technical limitations, e.g. if a component is designed for Linux-based systems it may not be deployable on a different operating system such as Windows.

Second, there are *non-functional* constraints from the domain. For instance, in the medical domain, some data repositories must be stored within the physical borders of a spe-

```
{services, infrastructure, initialDistribution, filters}:

filters.mapAttrOnAttr {
  inherit services infrastructure; [20]
  distribution = initialDistribution; [21]
  serviceProperty = "requiredZone"; [22]
  infrastructureProperty = "zone"; [23]
}
```

Figure 10: Candidate host selection using mapAttrOnAttr

```
{services, infrastructure, initialDistribution, filters}:

filters.mapAttrOnList {
  inherit services infrastructure;
  distribution = initialDistribution;
  serviceProperty = "type"; [24]
  infrastructureProperty = "supportedTypes"; [25]
}
```

Figure 11: Candidate host selection using attrOnList

cific country because this is required by the law of that particular country.

In order to filter a distribution, various mapping functions are available. The function mapAttrOnAttr can be used to map a value of an attribute defined in the services model onto a value of an attribute defined in the infrastructure model.

Figure 10 shows an example of this function, which creates a candidate host selection for the service model defined in Figure 5 and infrastructure model defined in Figure 6 ensuring that a service that must be deployed in a particular zone is mapped onto machines located in that zone. The mapAttrOnAttr function requires all the three models as input parameters ([20], [21]). [22] specifies that the requiredZone attribute defined in the services model must be mapped onto the zone attribute in the infrastructure model (specified at [23]). The result of this function is a new distribution model, in which every service belonging to a particular zone refers to machines that are located in that zone.

The function mapAttrOnList can be used to map a value of a service attribute onto a value defined in a list of values defined in the infrastructure model. Figure 11 shows how this function can be used to create a candidate host selection that ensures that a service of a specific type is mapped to machines supporting that particular type. [24] specifies that the type attribute in the services model is mapped onto one of the values in the supportedTypes list (defined at [25]). The expression described in Figure 11 is very valuable for all our example cases. This function ensures that a MySQL database is deployed on a machine running a MySQL DBMS, a Apache Tomcat web application is deployed on a server running Apache Tomcat and that other services of a specific type are deployed on the right machines. Similarly, a function mapListOnAttr is provided to map a list of values in the service model onto an attribute value defined in the infrastructure model.

Another issue is that services of a system may be *stateful*. For instance, data in a MySQL database may change after it has been deployed. Disnix does not migrate these changes to another machine in case of a redeployment. By defining the property stateful = true in the services model, a developer can specify that a service has mutable state. By using the mapStatefulToPrevious function, a developer can specify that in case of an upgrade, the stateful service is mapped only to the hosts of the previous deployment scenario. This ensures that a service such as a MySQL database is not moved and the data is preserved.

5.2 Division Phase

After a candidate host selection has been made, the services must be divided over the candidate hosts by using a suitable strategy. In the literature various deployment planning algorithms are described that try to find a suitable mapping of services to machines in certain scenarios. However, there is no algorithm available that always provides the best distribution for a particular system, nor it is always doable to find an optimal distribution. Therefore, this strategy must be selected and configured by the user.

Disnix provides an interface for integrating external algorithms as separate tooling (because the Nix expression language is limited for this purpose) and also *caches* the result in the Nix store like software components (i.e. if the same algorithm is executed with the same parameters the result from the Nix store is immediately returned instead of performing the algorithm again).

We have implemented a number of division methods. The most trivial method is a one-dimensional division generator, which uses a single variable from the services and infrastructure models and has three strategies. The *greedy* strategy iterates for each service over the possible candidate hosts and picks the first target until it is out of system resources, then moves over to the next until all services are mapped. The *highest-bidder* strategy picks for each iteration the target machine with the most system resources available. Similarly, the *lowest-bidder* strategy picks the target with the least system resources capable of running the service.

Additionally, we have support for a number of more complex algorithms, such as an approximation algorithm of the minimum set cover problem [14]. This algorithm can be used to approximate the most effective cost deployment scenario when every server has a fixed price, no matter how many services it will host. Another complex is algorithm is an approximation algorithm for the Multiway Cut problem [13], which can be used to approximate a deployment scenario in which the number of network links between services is minimised to make a system more reliable.

6. EVALUATION

We have evaluated the case studies described in Section 2 in a cluster of Xen virtual machines, each having the same amount of system resources. For each case study, we have implemented a QoS model to deal with desirable constraints, and we have used a separate division algorithm for each case.

We first performed an initial deployment in which the system is deployed in a small network of machines. Then we trigger various events, such as adding a server with specific capabilities or removing a server from the network. For each event triggering a redeployment, we check whether the features of the system are still accessible and measure the times of discovering the network state, generating the distribution model, building the system, distributing the services and transiting to (activating) the new configuration.

State	Description	Discovery (s)	Generation (s)	Build (s)	Distribution (s)	Transition (s)
StaffTracker (PHP/MySQL)						
0	Initial deployment	1.0	27.9	31.9	1.2	1.6
1	Add MySQL server 2	1.0	8.3	4.7	1.7	1.7
2	Add MySQL server 3	1.0	4.4	4.5	1.2	1.3
3	Add Apache server 2	1.0	4.4	4.4	1.3	1.5
4	Remove MySQL server 2	1.0	8.2	4.6	1.5	1.8
5	Remove Apache server 1	1.0	8.8	4.4	0.9	1.2
StaffTracker (Web services)						
0	Initial deployment	1.0	28.6	169.8	13.1	4.0
1	Add MySQL server $500	1.0	10.7	17.7	6.1	2.9
2	Add Tomcat server $500	1.0	8.4	10.2	13.7	5.7
3	Remove Tomcat server $500	1.0	4.2	0.5	0.6	2.0
4	Remove MySQL server $500	1.0	4.2	0.5	1.2	1.8
ViewVC						
0	Initial deployment	1.0	28.5	555.9	9.5	17.5
1	Add MySQL server	1.0	8.2	0.6	0.6	1.1
2	Add Apache server	1.0	8.4	0.5	0.6	1.0
3	Add Subversion server + repo	1.0	8.4	6.3	3.7	8.2
4	Remove Apache server 1	1.0	8.4	4.6	7.5	1.9
5	Remove MySQL server 1	1.0	8.3	5.7	1.9	1.8
SDS2						
0	Initial deployment	1.0	28.6	642.9	122.8	147.7
1	Add MySQL server 2 (hospital)	1.0	6.7	0.6	19.1	1.9
2	Add Tomcat server 2 (Philips)	1.0	8.4	0.6	18.5	1.8
3	Remove Tomcat server 1 (Philips)	1.0	8.3	360.2	26.9	6.3
4	Remove MySQL server 1 (hospital)	1.2	10.5	370.4	120.8	124.3

Table 1: Evaluation times of the example cases

Table 1 summarises the results of the evaluation for the four systems described in Section 2. The first is the StaffTracker (PHP/MySQL version). The size of all components and dependencies of this system is 120 KiB. The quality-of-service model that we defined for this system maps services of a specific type to the machines supporting them (e.g. MySQL database to servers running a MySQL DBMS). Moreover, the database storing the staff members is marked as stateful and must not be relocated in case of a redeployment. The highest bidder strategy is used to divide services over the candidate hosts.

Initially, the system is deployed in a network consisting of an Apache web server machine and a MySQL machine, which takes quite some time. Then we fire 2 events in which we add a machine to the network. Since Disnix can build and deploy components safely next to other components, this gives us no downtime. Only the transition phase (which only takes about 1.3-1.7 seconds) introduces some inconsistency, which is very short. Finally, we fire two remove events. In this case a new deployment is calculated and the system configuration is adapted. During these events downtimes are a bit longer (about 15 seconds in total), because the system must be partially rebuilt and reconfigured, during which time those services are not available.

Next is the SOA variant of the StaffTracker. The size of all components and dependencies of this variant is 94 MiB (which is significantly larger than the PHP version). We have used the same candidate host selection strategy as the previous example and the minimum set cover approximation algorithm for dividing services over the candidate hosts.

Similar results are observable compared to the previous example. The initial deployment also takes a significant amount of time. Events in which machines are added only take a couple of seconds. The failure events in this scenario are very cheap and only take about 8 seconds of redeployment time (including building, transferring and activating). This event is so cheap because Disnix does not remove already deployed components from machines, unless they are explicitly garbage collected. Because after these failure events the system is essentially the same as in a previous configuration, we can switch back nearly instantly.

In the ViewVC case, the size of all components and dependencies of the system is 360 MiB. We used the same candidate host mapping method as in the previous examples. We deployed ViewVC and its dependencies into a network consisting of an Apache webserver machine, a MySQL server machine and a Subversion machine. All Subversion repositories were marked as stateful, so that they were not relocated in case of an event. For dividing the services we have used the greedy algorithm to use the maximum amount of diskspace of a server.

As the table shows, an add event only introduces a couple of seconds of downtime. Event 3 takes a little longer, because a new Subversion repository must be copied and transferred to a new Subversion server. Event 4 and 5 trigger a redeployment of the web application front-end and MySQL cache.

The final evaluation subject is SDS2. For SDS2 we used, apart from the same candidate host mapping criteria as the previous examples, a strategy that deploys several services into a hospital environment and others in a Philips environment. The size of all components and dependencies of SDS2 is 599 MiB. The division method was the multiway cut approximation algorithm, which tries to find a distribution

with a minimum number of network links between services. For the initial deployment, we used 5 machines: 2 Apache Tomcat servers (in the Philips and hospital environments), 2 MySQL servers (in the Philips and hospital environments) and 1 Ejabberd server (in the hospital environment). The results of the deployment are comparable to the previous examples, e.g., a long initial deployment time and a very short redeployment time in case of a newly added server. The notable exception is the long redeployment times in case of a failing server. In these cases SDS2 had to be completely rebuilt from source (apart for some shared library dependencies). This is due to the fact that the SDS2 developers included a global configuration file containing the locations of all services with each service. In case of a change in the distribution, all services had to be rebuilt, repackaged and redeployed on all machines in the network. This problem could have been avoided by specifying small configuration files for each service which only capture settings of its inter-dependencies.

Another notable observation is the generation time for the initial deployment step. These values were always close to 28 seconds for all case studies and much smaller during the redeployment steps. This is due to the fact that these algorithms are invoked by the same build tooling used for Disnix, which will first download some required dependencies in order to execute the algorithms.

7. DISCUSSION

We have applied Disnix to several use cases and have shown that we can efficiently and reliably redeploy a system in case of an event dealing with desirable non-functional properties. However, there are some limitations in using Disnix for self-adaptive deployment of distributed systems.

The first issue is *applicability*. Disnix is designed for the deployment of service-oriented systems, that is, systems composed of distributable components, which are preferably small and relocatable. This requires developers to properly divide a system into distributable components. Some distributed systems are composed of components that are so tightly integrated into the environment, that it is infeasible to relocate the component in case of an event. For instance, the system may require a restart or a complete adaptation of the system configuration. Another issue is that some distributable components may completely fail after a disconnection without recovering afterwards. Disnix does not detect this behaviour. Moreover, Disnix uses a centralized approach, which means that components must be distributed and activated from a central location, and the complete system must be captured in the models used by Disnix. This is not possible for all types of distributed systems.

Disnix has only limited support for dealing with *mutable state*, such as databases, Subversion repositories, caches and log files. For services such as databases and Subversion repositories, Disnix can initialize a database schema and initial data at startup time, but Disnix does not automatically migrate the changes in the data made afterwards.

There are couple of solutions to deal with this issue. A developer or system administrator may want to keep a database on the same machine after the initial deployment. (This is why we provide the mapStatefulToPrevious function in Section 5.) Another solution is to create a cluster of databases that replicate the contents of the master database. We are also working on a generic approach to deal with mutable state of services, which snapshots mutable state and migrates it automatically as required.

Disnix currently only supports simple *network topologies*, in which every target machine is directly reachable. There is also no method to model complex network topologies yet. This prevents Disnix from using advanced deployment planning algorithms, such as Avala [16], that improve availability and take the network topology into account.

8. RELATED WORK

There is a substantial amount of work on self-adaptation in distributed environments. Several approaches involve specialized middleware. For instance, [6] describes using self-adaptive middleware to propose routing strategies of messages between components to meet requirements in a service-level agreement. In [11] a framework of models for QoS adaptation is described that operates at the middleware and transport levels of an application. In [21] the authors propose a translation system to develop QoS-aware applications, which for instance support CPU scheduling.

Another approach is to reconfigure previously deployed components, such as by changing the behaviour of components (e.g. [4]) or their connections (such as [7, 3] for adapting compositions of web services).

Compared to middleware approaches, Disnix is a more general approach and does not require components of a distributed system to use a particular component technology or middleware. Moreover, Disnix does not adapt component behaviour or connections at runtime. Instead it generates or builds a new component and redeploys it. In some cases this takes more effort, but the advantage is that Disnix tries to minimize this amount of effort and that this approach can be used with various types of components regardless of component technology. Moreover, if a crucial component breaks, the system is redeployed to fix the system, which reconfiguration approaches cannot always do properly. In some cases runtime adaptation may still be required, which should be managed by other means, such as by implementing such properties in the services themselves.

Some aspects of the present work are suggested in [9], such as an application model describing components and connections, a network model, and utility functions to perform a mapping of components to machines. Although the authors assess the usefulness of some utility function approaches, they do not perform actual deployment of a system, nor describe a framework that reacts to events.

Avala [16] is an approximation algorithm to improve availability through redeployment. The paper assesses the performance of the algorithm, but the authors do not actually redeploy a system. DICOMPLOY [18] is a decentralized deployment tool for components using the DRACO component model and uses Collaborative Reinforcement Learning (CRL) in the design of the component deployment framework. Disnix uses a centralized approach compared to DICOMPLOY. In some cases DICOMPLOY may be more efficient in redeploying a system, but Disnix is more general in that it supports various types of components and various strategies for mapping components to machines.

9. CONCLUSION

We have shown a dynamic deployment extension built on top of Disnix, which we have applied to several use cases.

This deployment extension efficiently and reliably redeploys a system in case of an event in the network (such as a crashing machine) to keep the overall system working and to ensure that desired non-functional requirements are met.

In the future we intend to improve this deployment system to properly deal with mutable state of services, so that associated data with a service is automatically migrated in case of a redeployment. Moreover, Disnix has to be extended to support complex network topologies, so that more sophisticated deployment algorithms can be supported.

Acknowledgements This research is supported by NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*. We wish to thank the contributors and developers of NixOS and SDS2, in particular Merijn de Jonge, who also contributed significantly to the development of Disnix.

10. REFERENCES

[1] Avahi. `http://avahi.org`, 2010.

[2] A. Akkerman, A. Totok, and V. Karamcheti. Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In *CD '05: Proc. of the 3rd Working Conf. on Component Deployment*, pages 17–32. Springer-Verlag, 2005.

[3] L. Baresi and L. Pasquale. Live goals for adaptive service compositions. In *Proc. of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 114–123, New York, NY, USA, 2010. ACM.

[4] J. Camara, C. Canal, and G. Salaun. Behavioural self-adaptation of services in ubiquitous computing environments. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 28–37, Washington, DC, USA, 2009. IEEE Computer Society.

[5] E. Caron, P. K. Chouhan, and H. Dail. GoDIET: A Deployment Tool for Distributed Middleware on Grid 5000. Technical Report RR-5886, Laboratoire de l'Informatique du Parallélisme (LIP), Apr. 2006.

[6] Y. Caseau. Self-adaptive middleware: Supporting business process priorities and service level agreements. *Advanced Engineering Informatics*, 19(3):199 – 211, July 2005.

[7] K. S. M. Chan and J. Bishop. The design of a self-healing composition cycle for web services. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 20–27, Washington, DC, USA, 2009. IEEE Computer Society.

[8] M. de Jonge, W. van der Linden, and R. Willems. eServices for Hospital Equipment. In B. Krämer, K.-J. Lin, and P. Narasimhan, editors, *5th Intl. Conf. on Service-Oriented Computing (ICSOC 2007)*, pages 391 – 397, Sept. 2007.

[9] D. Deb, M. J. Oudshoorn, and J. Paxton. Self-managed deployment in a distributed environment via utility functions. In *20th International Conference on Software Engineering and Knowledge Engineering (SEKE '08)*, 2008.

[10] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

[11] K. Guennoun, K. Drira, N. V. Wambeke, C. Chassot, F. Armando, and E. Exposito. A framework of models for QoS-oriented adaptive deployment of multi-layer communication services in group cooperative activities. *Computer Communications*, 31(13):3003 – 3017, Aug. 2008.

[12] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the software dock. In *ICSE '99: Proc. of the 21st Intl. Conf. on Software Engineering*, pages 174–183, New York, NY, USA, 1999. ACM.

[13] A. Heydarnoori and F. Mavaddat. Reliable deployment of component-based applications into distributed environments. *3rd Intl. Conf. on Information Technology: New Generations*, 0:52–57, 2006.

[14] A. Heydarnoori, F. Mavaddat, and F. Arbab. Deploying loosely coupled, component-based applications into distributed environments. *IEEE International Conference on the Engineering of Computer-Based Systems*, 0:93–102, 2006.

[15] N. Medvidovic and S. Malek. Software deployment architecture and quality-of-service in pervasive environments. In *International workshop on Engineering of software services for pervasive environments*, ESSPE '07, pages 47–51, New York, NY, USA, 2007. ACM.

[16] M. Mikic-Rakic, S. Malek, and N. Medvidovic. Improving Availability in Large, Distributed Component-Based Systems Via Redeployment. In A. Dearle and S. Eisenbach, editors, *Component Deployment*, volume 3798 of *Lecture Notes in Computer Science*, pages 83–98. Springer Berlin / Heidelberg, 2005.

[17] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40:38–45, 2007.

[18] P. Rigole and Y. Berbers. Resource-driven collaborative component deployment in mobile environments. In *Proc. of the International Conference on Autonomic and Autonomous Systems*, ICAS '06, Washington, DC, USA, 2006. IEEE Computer Society.

[19] S. van der Burg and E. Dolstra. Automated deployment of a heterogeneous service-oriented system. In L. O'Conner, editor, *36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 183–190. IEEE Computer Society, Sep 2010.

[20] S. van der Burg and E. Dolstra. Disnix: A toolset for distributed deployment. In *Third International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3)*, pages 58–75, Sep 2010.

[21] D. Wichadakul and K. Nahrstedt. A Translation System for Enabling Flexible and Efficient Deployment of QoS-Aware Applications in Ubiquitous Environments. In *CD '02: Proc. of the 1st Working Conf. on Component Deployment*, pages 287–310. Springer-Verlag, 2002.