Wednesday, February 16, 2011

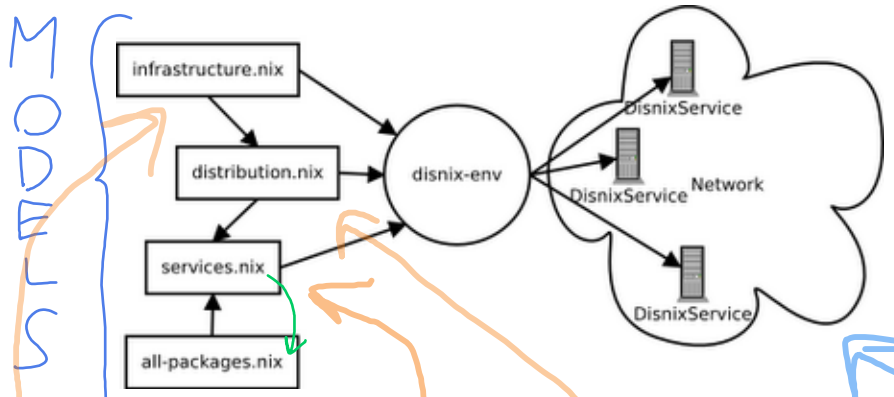
Disnix: A toolset for distributed deployment

QUESTION so there is a definition here (intra-dependency) and another definition (service) that is so short (distributable components) that it is tempting to use it as a synonym at times.
How to capture this with ontological / semantic tools?

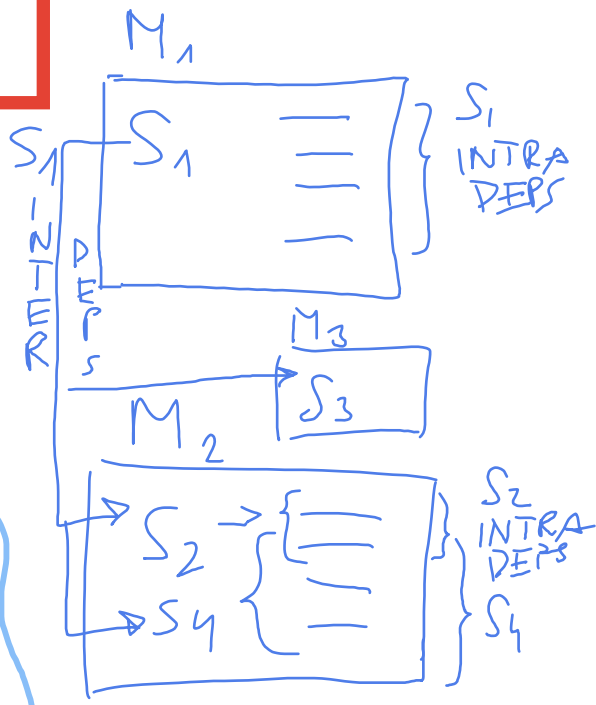
On February the 14th, I have released Disnix 0.2. It seems that I have picked a nice date for it, just like the release date of the of Disnix 0.1, which was released on April the 1st 2010 (and that's no joke). Since I haven't written any blog post about Disnix yet, I'll give some info here.

Disnix is a distributed deployment extension for the Nix package manager. Whereas Nix manages packages and dependencies residing on the same system in the Nix store (which we call **intra-dependencies** later on), Disnix manages distributable components (or **services**). Services have intra-dependencies on components residing on the same system, but also dependencies on other services, which may be located on a different machine in the network (which we call **inter-dependencies** later on). Disnix extends the Nix approach and offers various features to deploy to service-oriented systems, including the management of inter-dependencies.

TODO this should (also) be captured in a figure. in fact, this should have been the priority...



MODELS



The figure above shows how Disnix works in a nutshell. In the center the **disnix-env** command-line tool is shown, which performs the complete deployment process of a service-oriented system. On the left, various models are shown. The **services** model captures all the service (distributable components) of which a system consists, their types and inter-dependencies. This model includes a reference to **all-packages.nix**, a Nix expression capturing intra-dependency compositions. The **infrastructure** model captures the available machines in the network and their relevant properties/capabilities. The **distribution** model maps

TODO make it a list or something that has a "breezier" structure

services defined in the services model to machines defined in the infrastructure model. On the right, a network of machines is shown, which have all have the **DisnixService** installed providing remote access to deployment operations.

By writing instances of the models mentioned earlier and by running:

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

QUESTION how is the inter-dependency graph constructed?
I think there is a lot of stuff left out here that are only implied; eg the models (Nix expressions) paint a picture, the distribution model provides a map, and all this info is used to create a DAG that will "just" has to be traversed.

The system is built from source-code including all required intra-dependencies. Then the services and intra-dependencies are efficiently transferred to the target machines in the network. Finally, the services are activated in the right order derived from the inter-dependency graph.

By adapting the models and running **disnix-env** again, an upgrade is performed instead of a full installation. In this case only components which have changed are rebuilt and transferred to the target machines in the network. Moreover, only obsolete services are deactivated and new services are activated.

NOTE / WARNING would have liked an idempotent operation better with an explicit upgrade switch; this way one has to keep in mind what the state was when disnix- env was run the last time - or risk service disruption

Similar to writing ordinary Nix expressions for each package, you also write Disnix expressions for each service describing how it can be built from source and its dependencies.

and the filename is probably something like staff-service.nix

QUESTION what is the name of this file AND which model is this?

```
{stdenv, StaffService}: intra-dependencies
{staff}: INTER-dependencies

let
  jdbcURL = "jdbc:mysql://"+
    staff.target.hostname+":"+
    toString (staff.target.mysqlPort)+"/"+
    staff.name+"?autoReconnect=true";
  contextXML = ''
    <Context>
      <Resource name="jdbc/StaffDB" auth="Container"
            type="javax.sql.DataSource"
            maxActivate="100" maxIdle="30" maxWait="10000"
            username="${staff.target.mysqlUsername}"
            password="${staff.target.mysqlPassword}"
            driverClassName="com.mysql.jdbc.Driver"
            url="${jdbcURL}" />
    </Context>
  '';
```

QUESTION where is 'target' coming from?
yes, from here but is this just a vague example, or there should be a 'target' attribute somewhere?

```
in
stdenv.mkDerivation {
  name = "StaffService";
  buildCommand = ''
    ensureDir $out/conf/Catalina
    cat > $out/conf/Catalina/StaffService.xml <<EOF
    ${contextXML}
    EOF
    ln -sf ${StaffService}/webapps $out/webapps
  '';
}
```

QUESTION I understand what this does but unfamiliar with this notation

The code fragement above shows a Disnix expression for the StaffTracker example included in the Disnix repository. The main difference between this expression and an ordinary Nix expression is that it has two function headers which takes intra-dependencies and inter-dependencies respectively to configure the component. The inter-dependency arguments are used in this expression to generate a so called context XML file, which Apache Tomcat uses to configure resources such as JDBC connections, containing a URL, port number and authentication credentials for a MySQL database residing on a different machine in the network. For other types of components a different configuration file has to be created.

Moreover, you also need to *compose* a Disnix expression. A Disnix expression must first be composed locally by calling the function with the right intra-dependency arguments. This is done in a similar way as ordinary Nix expressions. Later, the same function is called with the right inter-dependency arguments as well.

This is the service model (I think) as it calls out to all-packages.nix

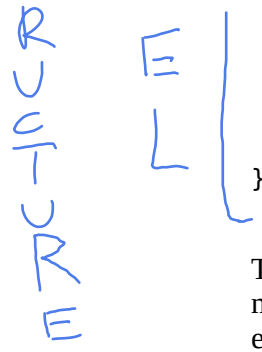
```
{distribution, system, pkgs}:

let customPkgs = import ../top-level/all-packages.nix {
  inherit system pkgs;
};
in
rec {
### Databases
  staff = {
    name = "staff";
    pkg = customPkgs.staff;
    dependsOn = {};
    type = "mysql-database";
  };
  ...

### Web services
```

SEMO

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

### Web applications

  StaffTracker = {
    name = "StaffTracker";
    pkg = customPkgs.StaffTracker;
    dependsOn = {
      inherit GeolocationService RoomService;
      inherit StaffService ZipcodeService;
    };
    type = "tomcat-webapplication";
  };
  ...
}
```

The above expression shows the **services** model, used to capture of which distributable components a system consists. Basically, this model is a function taking three arguments: the distribution model (shown later), a collection of Nixpkgs and a system identifier indicating the architecture of a target host. The function returns an attribute set in which each attribute represents a service. For each service various properties are defined, such as a **name**, a **pkg** attribute referring to a function which creates an intra-dependency composition of a service (defined in an external file not shown here), **dependsOn** composing the inter-dependencies and a **type**, which is used for activation and deactivation of the service.


```
{
  test1 = {
    hostname = "test1.example.org";
    tomcatPort = 8080;
    system = "i686-linux";
  };

  test2 = {
    hostname = "test2.example.org";
    tomcatPort = 8080;
    mysqlPort = 3306;
    mysqlUsername = "root";
```




```
    mysqlPassword = "secret";
    system = "i686-linux";
  };
}
```

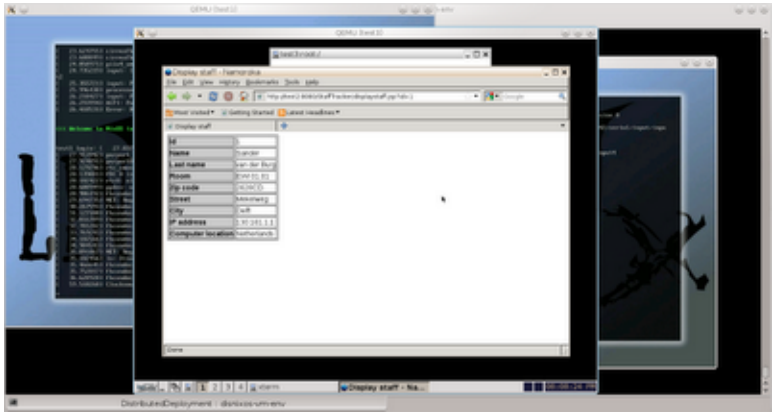
The expression shown above is an **infrastructure** model, which captures machines in the network and their relevant properties/capabilities. This expression is an attribute set in which each attribute represents a machine in the network. Some properties are mandatory, such as the **hostname** indicating how the Disnix service can be reached. The **system** property denotes the system architecture so that a service is built for that particular platform. Other properties can be freely chosen and are used for activation/deactivation of a component.

```
{infrastructure}:

{
  GeolocationService = [ infrastructure.test1 ];
  RoomService = [ infrastructure.test2 ];
  StaffService = [ infrastructure.test1 ];
  StaffTracker = [ infrastructure.test1 infrastructure.test2 ];
  ZipcodeService = [ infrastructure.test1 ];
  rooms = [ infrastructure.test2 ];
  staff = [ infrastructure.test2 ];
  zipcodes = [ infrastructure.test2 ];
}
```

The final expression shown above is the **distribution** model, mapping services to machines in the network. This expression is a function taking the infrastructure model as parameter. The body is an attribute set in which every attribute representing a service refers to a list of machines in the network. It also allows you to map a service to multiple machines for e.g. load balancing.

The models shown earlier are used by Disnix to perform the complete deployment process of a service-oriented system, i.e. building services, transferring services and the activation of services. Because Disnix uses the purely functional properties of Nix, this process is reliable and efficient. If a system is upgraded, no components are removed and overwritten, since everything is stored in isolation in the Nix store. So while upgrading, we can still keep the current system intact. Only during the transition phase in which services are deactivated and activated the system is inconsistent, but Disnix keeps this time window as small as possible. Moreover, a proxy can

be used during this phase to queue connections, which makes the upgrade process truly atomic.



Although Disnix supports the deployment of a service-oriented system, some additional extensions have been developed to make deployment more convenient:

- **DisnixWebService**. By default Disnix uses a SSH connection to connect to remote machines in the network. This extension provides a SOAP interface and **disnix-soap-client** to perform deployment through the SOAP protocol.
- **DisnixOS**. Disnix manages the services of which a system is composed, but not the system configurations of the underlying infrastructure. This extension provides additional infrastructure management features to Disnix based on the techniques described in the blog post titled: Using NixOS for declarative deployment and testing. By using this extension you can automatically deploy a network of NixOS configurations next to the services through Disnix. Moreover, you can also use this extension to generate a network of virtual machines and automatically deploy the system in the virtual network. A screenshot is shown above, which runs the StaffTracker example in a network of three virtual machines.
- **Dynamic Disnix**. Disnix requires developers or system administrators to manually write an infrastructure model and a distribution model. In a network in which *events* occur, such as a machine which crashes or a new machine with new system resource is added, this introduces a large degree of *inflexibility*. The Dynamic Disnix toolset offers a discovery service, which dynamically

discovers the machines in the network and their relevant properties/capabilities. Moreover, it also includes a distribution model generator, which uses a custom defined policy and a collection of distribution algorithms to dynamically distribute services to machines, based on non-functional properties defined in the services and infrastructure models.
The Dynamic Disnix extension is still under heavy development and not released as part of Disnix 0.2. It will become part of the next Disnix release.

Disnix, the extensions and some examples can be obtained from the Disnix web page: http://nixos.org/disnix. Disnix is also described in several academic papers. The paper: 'Disnix: A toolset for distributed deployment' describes the architecture of the Disnix toolset. This paper is however somewhat outdated, as there are some minor changes in the current implementation. The paper: 'Automated Deployment of a Heterogeneous Service-Oriented System' describes the 0.1 implementation, which we have used for a case study at Philips Research. The publications and presentation slides can be obtained from the publications and talks sections of my homepage. Moreover, there are some earlier publications about Disnix available as well. In a next blog post, I will explain more about the development process and development choices of Disnix.