Friday, March 12, 2021

Using the Nix process management framework as an infrastructure deployment solution for Disnix

As explained in many previous blog posts, I have developed Disnix as a solution for automating the deployment of service-oriented systems -- it deploys **heterogeneous systems**, that consist of many different kinds of components (such as web applications, web services, databases and processes) to **networks** of machines.

The deployment models for Disnix are typically not fully self-contained. Foremost, a precondition that must be met before a service-oriented system can be deployed, is that all target machines in the network require the presence of Nix package manager, Disnix, and a remote connectivity service (e.g. SSH).

For multi-user Disnix installations, in which the user does not have super-user privileges, the Disnix service is required to carry out deployment operations on behalf of a user.

Moreover, the services in the services model typically need to be managed by other services, called **containers** in Disnix terminology (not to be confused with Linux containers).

Examples of container services are:

- The MySQL DBMS container can manage multiple databases deployed by Disnix.
- The Apache Tomcat servlet container can manage multiple Java web applications deployed by Disnix.
- systemd can act as a container that manages multiple systemd units deployed by Disnix.

Managing the life-cycles of services in containers (such as activating or deactivating them) is done by a companion tool called Dysnomia.

In addition to Disnix, these container services also typically need to be deployed in advance to the target machines in the network.

The problem domain that Disnix works in is called **service deployment**, whereas the deployment of machines (bare metal or virtual machines) and the container services is called **infrastructure deployment**.

Disnix can be complemented with a variety of infrastructure deployment solutions:

- NixOps can deploy networks of NixOS machines, both physical and virtual machines (in the cloud), such as Amazon EC2.

  As part of a NixOS configuration, the Disnix service can be deployed that facilitates multi-user installations. The Dysnomia NixOS module can expose all relevant container services installed by NixOS as container deployment targets.
- *disnixos-deploy-network* is a tool that is included with the DisnixOS extension toolset. Since services in Disnix can be any kind of deployment unit, it is also possible to deploy an entire NixOS configuration as a service. This tool is mostly developed for demonstration purposes.

  A limitation of this tool is that it cannot instantiate virtual machines and bootstrap Disnix.
- Disnix itself. The above solutions are all NixOS-based, a software distribution that is Linux-based and fully managed by the Nix package manager.

  Although NixOS is very powerful, it has two drawbacks for Disnix:

  - NixOS uses the NixOS module system for configuring system aspects. It is very powerful but you can only deploy one instance of a system service -- Disnix can also work with multiple container instances of the same type on a machine.
  - Services in NixOS cannot be deployed to other kinds software distributions: conventional Linux distributions, and other operating systems, such as macOS and FreeBSD.

  To overcome these limitations, Disnix can also be used as a container deployment solution on any operating system that is capable of running Nix and Disnix. Services deployed by Disnix can automatically be exposed as container providers.

  Similar to *disnix-deploy-network*, a limitation of this approach is that it cannot be used to bootstrap Disnix.

Last year, I have also added a new major feature to Disnix making it possible to deploy both application and container services in the same Disnix deployment models, minimizing the infrastructure deployment problem -- the only requirement is to have machines with Nix, Disnix, and a remote connectivity service (such as SSH) pre-installed on them.

Although this integrated feature is quite convenient, in particular for test setups, a

separated infrastructure deployment process (that includes container services) still makes sense in many scenarios:

- The infrastructure parts and service parts can be managed by different people with **different specializations**. For example, configuring and tuning an application server is a different responsibility than developing a Java web application.
- The service parts typically change more frequently than the infrastructure parts. As a result, they typically have **different** kinds of **update cycles**.
- The infrastructure components can typically be **reused** between projects (e.g. many systems use a database backend such as PostgreSQL or MySQL), whereas the service components are typically very project specific.

I also realized that my other project: the Nix process management framework can serve as a partial infrastructure deployment solution -- it can be used to bootstrap Disnix and deploy container services.

Moreover, it can also deploy multiple instances of container services and used on any operating system that the Nix process management framework supports, including conventional Linux distributions and other operating systems, such as macOS and FreeBSD.

Deploying and exposing the Disnix service with the Nix process management framework

As explained earlier, to allow Disnix to deploy services to a remote machine, a machine needs to have Disnix installed (and run the Disnix service for a multi-user installation), and be remotely connectible, e.g. through SSH.

I have packaged all required services as constructor functions for the Nix process management framework.

The following process model captures the configuration of a basic multi-user Disnix installation:

```
{ pkgs ? import <nixpkgs> { inherit system; }
, system ? builtins.currentSystem
, stateDir ? "/var"
, runtimeDir ? "${stateDir}/run"
, logDir ? "${stateDir}/log"
, spoolDir ? "${stateDir}/spool"
, cacheDir ? "${stateDir}/cache"
, tmpDir ? (if stateDir == "/var" then "/tmp" else "${stateDir}/tmp")
```

```
, forceDisableUserChange ? false
, processManager
}:

let
  ids = if builtins.pathExists ./ids-bare.nix then (import ./ids-bare

  constructors = import ../../services-agnostic/constructors.nix {
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
  };
in
rec {
  sshd = {
    pkg = constructors.sshd {
      extraSSHDConfig = ''
        UsePAM yes
      '';
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  dbus-daemon = {
    pkg = constructors.dbus-daemon {
      services = [ disnix-service ];
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  disnix-service = {
    pkg = constructors.disnix-service {
      inherit dbus-daemon;
    };

    requiresUniqueIdsFor = [ "gids" ];
  };
}
```

The above processes model (*processes.nix*) captures three process instances:

- *sshd* is the OpenSSH server that makes it possible to remotely connect to the machine by using the SSH protocol.
- *dbus-daemon* runs a D-Bus system daemon, that is a requirement for the Disnix service. The *disnix-service* is propagated as a parameter, so that its service directory gets added to the D-Bus system daemon configuration.
- *disnix-service* is a service that executes deployment operations on behalf of an authorized unprivileged user. The *disnix-service* has a dependency on the *dbus-service* making sure that the latter gets activated first.

We can deploy the above configuration on a machine that has the Nix process management framework already installed.

For example, to deploy the configuration on a machine that uses supervisord, we can run:

```
$ nixproc-supervisord-switch processes.nix
```

Resulting in a system that consists of the following running processes:

```
$ supervisorctl
dbus-daemon                      RUNNING   pid 2374, uptime 0:00:34
disnix-service                   RUNNING   pid 2397, uptime 0:00:33
sshd                             RUNNING   pid 2375, uptime 0:00:34
```

As may be noticed, the above supervised services correspond to the processes in the processes model.

On the coordinator machine, we can write a **bootstrap** infrastructure model (*infra-bootstrap.nix*) that only contains connectivity settings:

```
{
  test1.properties.hostname = "192.168.2.1";
}
```

and use the bootstrap model to capture the full infrastructure model of the system:

```
$ disnix-capture-infra infra-bootstrap.nix
```

resulting in the following configuration:

```
{
  "test1" = {
    properties = {
      "hostname" = "192.168.2.1";
      "system" = "x86_64-linux";
    };
    containers = {
      echo = {
      };
      fileset = {
      };
      process = {
```

```
      };
      supervisord-program = {
        "supervisordTargetDir" = "/etc/supervisor/conf.d";
      };
      wrapper = {
      };
    };
    "system" = "x86_64-linux";
  };
}
```

Despite the fact that we have not configured any containers explicitly, the above configuration (*infrastructure.nix*) already exposes a number of container services:

- The *echo*, *fileset* and *process* container services are built-in container providers that any Dysnomia installation includes.

  The *process* container can be used to automatically deploy services that daemonize. Services that daemonize themselves do not require the presence of any external service.
- The *supervisord-program* container refers to the process supervisor that manages the services deployed by the Nix process management framework. It can also be used as a container for processes deployed by Disnix.

With the above infrastructure model, we can deploy any system that depends on the above container services, such as the trivial [Disnix proxy example]:

```
{ system, distribution, invDistribution, pkgs
, stateDir ? "/var"
, runtimeDir ? "${stateDir}/run"
, logDir ? "${stateDir}/log"
, cacheDir ? "${stateDir}/cache"
, tmpDir ? (if stateDir == "/var" then "/tmp" else "${stateDir}/tmp")
, forceDisableUserChange ? false
, processManager ? "supervisord"
, nix-processmgmt ? ../../../nix-processmgmt
}:

let
  customPkgs = import ../top-level/all-packages.nix {
    inherit system pkgs stateDir logDir runtimeDir tmpDir forceDisabl
  };

  ids = if builtins.pathExists ./ids.nix then (import ./ids.nix).ids

  processType = import "${nix-processmgmt}/nixproc/derive-dysnomia-pr
    inherit processManager;
  };
in
```

```
rec {
  hello_world_server = rec {
    name = "hello_world_server";
    port = ids.ports.hello_world_server or 0;
    pkg = customPkgs.hello_world_server { inherit port; };
    type = processType;
    requiresUniqueIdsFor = [ "ports" ];
  };

  hello_world_client = {
    name = "hello_world_client";
    pkg = customPkgs.hello_world_client;
    dependsOn = {
      inherit hello_world_server;
    };
    type = "package";
  };
}
```

The services model shown above (*services.nix*) captures two services:

- The *hello_world_server* service is a simple service that listens on a TCP port for a "hello" message and responds with a "Hello world!" message.
- The *hello_world_client* service is a package providing a client executable that automatically connects to the *hello_world_server*.

With the following distribution model (*distribution.nix*), we can map all the services to our deployment machine (that runs the Disnix service managed by the Nix process management framework):

```
{infrastructure}:

{
  hello_world_client = [ infrastructure.test1 ];
  hello_world_server = [ infrastructure.test1 ];
}
```

and deploy the system by running the following command:

```
$ disnix-env -s services-without-proxy.nix \
  -i infrastructure.nix \
  -d distribution.nix \
  --extra-params '{ processManager = "supervisord"; }'
```

The last parameter: *--extra-params* configures the services model (that indirectly

invokes the *createManagedProcess* abstraction function from the Nix process management framework) in such a way that supervisord configuration files are generated.

(As a sidenote: without the *--extra-params* parameter, the process instances will be built for the *disnix* process manager generating configuration files that can be deployed to the process container, expecting programs to daemonize on their own and leave a PID file behind with the daemon's process ID. Although this approach is convenient for experiments, because no external service is required, it is not as reliable as managing supervised processes).

The result of the above deployment operation is that the *hello-world-service* service is deployed as a service that is also managed by supervisord:

```
$ supervisorctl
dbus-daemon                      RUNNING   pid 2374, uptime 0:09:39
disnix-service                   RUNNING   pid 2397, uptime 0:09:38
hello-world-server               RUNNING   pid 2574, uptime 0:00:06
sshd                             RUNNING   pid 2375, uptime 0:09:39
```

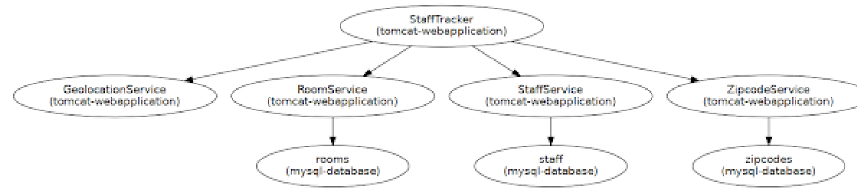
and we can use the *hello-world-client* executable on the target machine to connect to the service:

```
$ /nix/var/nix/profiles/disnix/default/bin/hello-world-client
Trying 192.168.2.1...
Connected to 192.168.2.1.
Escape character is '^]'.
hello
Hello world!
```

Deploying container providers and exposing them

With Disnix, it is also possible to deploy systems that are composed of different kinds of components, such as web services and databases.

For example, the Java variant of the ridiculous Staff Tracker example consists of the following services:



The services in the diagram above have the following purpose:

- The *StaffTracker* service is the front-end web application that shows an overview of staff members and their locations.
- The *StaffService* service is web service with a SOAP interface that provides read and write access to the staff records. The staff records are stored in the *staff* database.
- The *RoomService* service provides read access to the rooms records, that are stored in a separate *rooms* database.
- The *ZipcodeService* service provides read access to zip codes, that are stored in a separate *zipcodes* database.
- The *GeolocationService* infers the location of a staff member from its IP address using the GeoIP service.

To deploy the system shown above, we need a target machine that provides Apache Tomcat (for managing the web application front-end and web services) and MySQL (for managing the databases) as container provider services:

```
{ pkgs ? import <nixpkgs> { inherit system; }
, system ? builtins.currentSystem
, stateDir ? "/var"
, runtimeDir ? "${stateDir}/run"
, logDir ? "${stateDir}/log"
, spoolDir ? "${stateDir}/spool"
, cacheDir ? "${stateDir}/cache"
, tmpDir ? (if stateDir == "/var" then "/tmp" else "${stateDir}/tmp")
, forceDisableUserChange ? false
, processManager
}:

let
  ids = if builtins.pathExists ./ids-tomcat-mysql.nix then (import ./

  constructors = import ../../services-agnostic/constructors.nix {
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
  };

  containerProviderConstructors = import ../../service-containers-agn
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
```

```
  };
in
rec {
  sshd = {
    pkg = constructors.sshd {
      extraSSHDConfig = ''
        UsePAM yes
      '';
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  dbus-daemon = {
    pkg = constructors.dbus-daemon {
      services = [ disnix-service ];
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  tomcat = containerProviderConstructors.simpleAppservingTomcat {
    commonLibs = [ "${pkgs.mysql_jdbc}/share/java/mysql-connector-jav
    webapps = [
      pkgs.tomcat9.webapps # Include the Tomcat example and managemen
    ];

    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  mysql = containerProviderConstructors.mysql {
    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  disnix-service = {
    pkg = constructors.disnix-service {
      inherit dbus-daemon;
      containerProviders = [ tomcat mysql ];
    };

    requiresUniqueIdsFor = [ "gids" ];
  };
}
```

The process model above is an extension of the previous processes model, adding two container provider services:

- *tomcat* is the Apache Tomcat server. The constructor function: *simpleAppServingTomcat* composes a configuration for a supported process manager, such as supervisord.

  Moreover, it bundles a Dysnomia container configuration file, and a

Dysnomia module: *tomcat-webapplication* that can be used to manage the life-cycles of Java web applications embedded in the servlet container.
- *mysql* is the MySQL DBMS server. The constructor function also creates a process manager configuration file, and bundles a Dysnomia container configuration file and module that manages the life-cycles of databases.
- The container services above are propagated as *containerProviders* to the *disnix-service*. This function parameter is used to update the search paths for container configuration and modules, so that services can be deployed to these containers by Disnix.

After deploying the above processes model, we should see the following infrastructure model after capturing it:

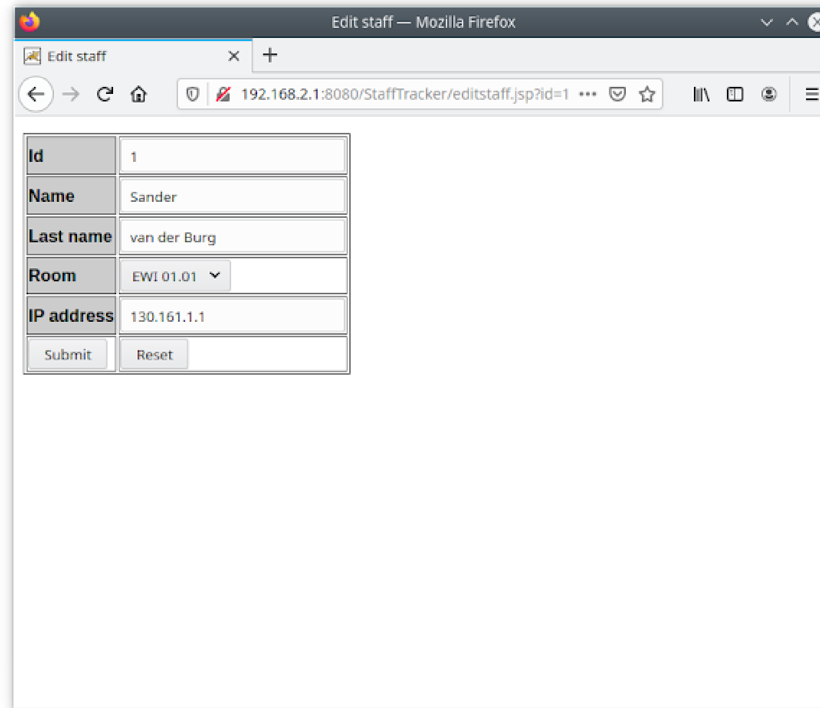
```
$ disnix-capture-infra infra-bootstrap.nix
{
  "test1" = {
    properties = {
      "hostname" = "192.168.2.1";
      "system" = "x86_64-linux";
    };
    containers = {
      echo = {
      };
      fileset = {
      };
      process = {
      };
      supervisord-program = {
        "supervisordTargetDir" = "/etc/supervisor/conf.d";
      };
      wrapper = {
      };
      tomcat-webapplication = {
        "tomcatPort" = "8080";
        "catalinaBaseDir" = "/var/tomcat";
      };
      mysql-database = {
        "mysqlPort" = "3306";
        "mysqlUsername" = "root";
        "mysqlPassword" = "";
        "mysqlSocket" = "/var/run/mysqld/mysqld.sock";
      };
    };
    "system" = "x86_64-linux";
  };
}
```

As may be observed, the *tomcat-webapplication* and *mysql-database* containers (with their relevant configuration properties) were added to the infrastructure model.

With the following command we can deploy the example system's services to the containers in the network:

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

resulting in a fully functional system:



Deploying multiple container provider instances

As explained in the introduction, a limitation of the NixOS module system is that it is only possible to construct one instance of a service on a machine.

Process instances in a processes model deployed by the Nix process management framework as well as services in a Disnix services model are instantiated from

functions that make it possible to deploy **multiple instances** of the same service to the same machine, by making conflicting properties configurable.

The following processes model was modified from the previous example to deploy two MySQL servers and two Apache Tomcat servers to the same machine:

```
{ pkgs ? import <nixpkgs> { inherit system; }
, system ? builtins.currentSystem
, stateDir ? "/var"
, runtimeDir ? "${stateDir}/run"
, logDir ? "${stateDir}/log"
, spoolDir ? "${stateDir}/spool"
, cacheDir ? "${stateDir}/cache"
, tmpDir ? (if stateDir == "/var" then "/tmp" else "${stateDir}/tmp")
, forceDisableUserChange ? false
, processManager
}:

let
  ids = if builtins.pathExists ./ids-tomcat-mysql-multi-instance.nix

  constructors = import ../../services-agnostic/constructors.nix {
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
  };

  containerProviderConstructors = import ../../service-containers-agn
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
  };
in
rec {
  sshd = {
    pkg = constructors.sshd {
      extraSSHDConfig = ''
        UsePAM yes
      '';
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  dbus-daemon = {
    pkg = constructors.dbus-daemon {
      services = [ disnix-service ];
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  tomcat-primary = containerProviderConstructors.simpleAppservingTomc
    instanceSuffix = "-primary";
    httpPort = 8080;
    httpsPort = 8443;
    serverPort = 8005;
    ajpPort = 8009;
```

```
    commonLibs = [ "${pkgs.mysql_jdbc}/share/java/mysql-connector-jav
    webapps = [
      pkgs.tomcat9.webapps # Include the Tomcat example and managemen
    ];
    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  tomcat-secondary = containerProviderConstructors.simpleAppservingTo
    instanceSuffix = "-secondary";
    httpPort = 8081;
    httpsPort = 8444;
    serverPort = 8006;
    ajpPort = 8010;
    commonLibs = [ "${pkgs.mysql_jdbc}/share/java/mysql-connector-jav
    webapps = [
      pkgs.tomcat9.webapps # Include the Tomcat example and managemen
    ];
    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  mysql-primary = containerProviderConstructors.mysql {
    instanceSuffix = "-primary";
    port = 3306;
    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  mysql-secondary = containerProviderConstructors.mysql {
    instanceSuffix = "-secondary";
    port = 3307;
    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  disnix-service = {
    pkg = constructors.disnix-service {
      inherit dbus-daemon;
      containerProviders = [ tomcat-primary tomcat-secondary mysql-pr
    };

    requiresUniqueIdsFor = [ "gids" ];
  };
}
```

In the above processes model, we made the following changes:

- We have configured two Apache Tomcat instances: *tomcat-primary* and *tomcat-secondary*. Both instances can co-exist because they have been configured in such a way that they listen to unique TCP ports and have a unique instance name composed from the *instanceSuffix*.
- We have configured two MySQL instances: *mysql-primary* and *mysql-secondary*. Similar to Apache Tomcat, they can both co-exist because they

listen to unique TCP ports (e.g. *3306* and *3307*) and have a unique instance name.
- Both the primary and secondary instances of the above services are propagated to the *disnix-service* (with the *containerProviders* parameter) making it possible for a client to discover them.

After deploying the above processes model, we can run the following command to discover the machine's configuration:

```
$ disnix-capture-infra infra-bootstrap.nix
{
  "test1" = {
    properties = {
      "hostname" = "192.168.2.1";
      "system" = "x86_64-linux";
    };
    containers = {
      echo = {
      };
      fileset = {
      };
      process = {
      };
      supervisord-program = {
        "supervisordTargetDir" = "/etc/supervisor/conf.d";
      };
      wrapper = {
      };
      tomcat-webapplication-primary = {
        "tomcatPort" = "8080";
        "catalinaBaseDir" = "/var/tomcat-primary";
      };
      tomcat-webapplication-secondary = {
        "tomcatPort" = "8081";
        "catalinaBaseDir" = "/var/tomcat-secondary";
      };
      mysql-database-primary = {
        "mysqlPort" = "3306";
        "mysqlUsername" = "root";
        "mysqlPassword" = "";
        "mysqlSocket" = "/var/run/mysqld-primary/mysqld.sock";
      };
      mysql-database-secondary = {
        "mysqlPort" = "3307";
        "mysqlUsername" = "root";
        "mysqlPassword" = "";
        "mysqlSocket" = "/var/run/mysqld-secondary/mysqld.sock";
      };
    };
    "system" = "x86_64-linux";
  };
}
```

As may be observed, the infrastructure model contains two Apache Tomcat instances and two MySQL instances.

With the following distribution model (*distribution.nix*), we can divide each database and web application over the two container instances:

```
{infrastructure}:

{
  GeolocationService = {
    targets = [
      { target = infrastructure.test1;
        container = "tomcat-webapplication-primary";
      }
    ];
  };
  RoomService = {
    targets = [
      { target = infrastructure.test1;
        container = "tomcat-webapplication-secondary";
      }
    ];
  };
  StaffService = {
    targets = [
      { target = infrastructure.test1;
        container = "tomcat-webapplication-primary";
      }
    ];
  };
  StaffTracker = {
    targets = [
      { target = infrastructure.test1;
        container = "tomcat-webapplication-secondary";
      }
    ];
  };
  ZipcodeService = {
    targets = [
      { target = infrastructure.test1;
        container = "tomcat-webapplication-primary";
      }
    ];
  };
  rooms = {
    targets = [
      { target = infrastructure.test1;
        container = "mysql-database-primary";
      }
    ];
  };
  staff = {
    targets = [
```

```
      { target = infrastructure.test1;
        container = "mysql-database-secondary";
      }
    ];
  };
  zipcodes = {
    targets = [
      { target = infrastructure.test1;
        container = "mysql-database-primary";
      }
    ];
  };
}
```

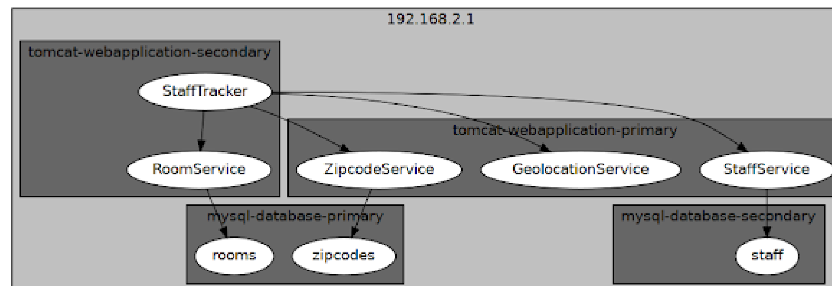
Compared to the previous distribution model, the above model uses a more verbose notation for mapping services.

As explained in an earlier blog post, in deployments in which only a single container is deployed, services are **automapped** to the container that has the same name as the service's type. When multiple instances exist, we need to manually specify the container where the service needs to be deployed to.

After deploying the system with the following command:

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

we will get a running system with the following deployment architecture:



Using the Disnix web service for executing remote deployment operations

By default, Disnix uses SSH to communicate to target machines in the network. Disnix has a modular architecture and is also capable of communicating to target machines by other means, for example via NixOps, the backdoor client, D-Bus, and directly executing tasks on a local machine.

There is also an external package: *DisnixWebService* that remotely exposes all deployment operations from a web service with a SOAP API.

To use the *DisnixWebService*, we must deploy a Java servlet container (such as Apache Tomcat) with the *DisnixWebService* application, configured in such a way that it can connect to the *disnix-service* over the D-Bus system bus.

The following processes model is an extension of the non-multi containers Staff Tracker example, with an Apache Tomcat service that bundles the *DisnixWebService*:

```
{ pkgs ? import <nixpkgs> { inherit system; }
, system ? builtins.currentSystem
, stateDir ? "/var"
, runtimeDir ? "${stateDir}/run"
, logDir ? "${stateDir}/log"
, spoolDir ? "${stateDir}/spool"
, cacheDir ? "${stateDir}/cache"
, tmpDir ? (if stateDir == "/var" then "/tmp" else "${stateDir}/tmp")
, forceDisableUserChange ? false
, processManager
}:

let
  ids = if builtins.pathExists ./ids-tomcat-mysql.nix then (import ./

  constructors = import ../../services-agnostic/constructors.nix {
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
  };

  containerProviderConstructors = import ../../service-containers-agn
    inherit pkgs stateDir runtimeDir logDir tmpDir cacheDir spoolDir
  };
in
rec {
  sshd = {
    pkg = constructors.sshd {
      extraSSHDConfig = ''
        UsePAM yes
      '';
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  dbus-daemon = {
```

```
    pkg = constructors.dbus-daemon {
      services = [ disnix-service ];
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  tomcat = containerProviderConstructors.disnixAppservingTomcat {
    commonLibs = [ "${pkgs.mysql_jdbc}/share/java/mysql-connector-jav
    webapps = [
      pkgs.tomcat9.webapps # Include the Tomcat example and managemen
    ];
    enableAJP = true;
    inherit dbus-daemon;

    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  apache = {
    pkg = constructors.basicAuthReverseProxyApache {
      dependency = tomcat;
      serverAdmin = "admin@localhost";
      targetProtocol = "ajp";
      portPropertyName = "ajpPort";

      authName = "DisnixWebService";
      authUserFile = pkgs.stdenv.mkDerivation {
        name = "htpasswd";
        buildInputs = [ pkgs.apacheHttpd ];
        buildCommand = ''
          htpasswd -cb ./htpasswd admin secret
          mv htpasswd $out
        '';
      };
      requireUser = "admin";
    };

    requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  mysql = containerProviderConstructors.mysql {
    properties.requiresUniqueIdsFor = [ "uids" "gids" ];
  };

  disnix-service = {
    pkg = constructors.disnix-service {
      inherit dbus-daemon;
      containerProviders = [ tomcat mysql ];
      authorizedUsers = [ tomcat.name ];
      dysnomiaProperties = {
        targetEPR = "http://$(hostname)/DisnixWebService/services/Dis
      };
    };

    requiresUniqueIdsFor = [ "gids" ];
  };
}
```

The above processes model contains the following changes:

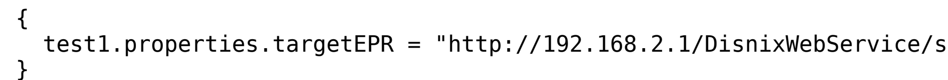
- The Apache Tomcat process instance is constructed with the *containerProviderConstructors.disnixAppservingTomcat* constructor function automatically deploying the *DisnixWebService* and providing the required configuration settings so that it can communicate with the *disnix-service* over the D-Bus system bus.

  Because the *DisnixWebService* requires the presence of the D-Bus system daemon, it is configured as a *dependency* for Apache Tomcat ensuring that it is started before Apache Tomcat.
- Connecting to the Apache Tomcat server including the *DisnixWebService* requires no authentication. To secure the web applications and the *DisnixWebService*, I have configured an *apache* reverse proxy that forwards connections to Apache Tomcat using the AJP protocol.

  Moreover, the reverse proxy protects incoming requests by using HTTP basic authentication requiring a username and password.

We can use the following bootstrap infrastructure model to discover the machine's configuration:

```
{
  test1.properties.targetEPR = "http://192.168.2.1/DisnixWebService/s
}
```



The difference between this bootstrap infrastructure model and the previous is that it uses a different connection property (*targetEPR*) that refers to the URL of the *DisnixWebService*.

By default, Disnix uses the *disnix-ssh-client* to communicate to target machines. To use a different client, we must set the following environment variables:

```
$ export DISNIX_CLIENT_INTERFACE=disnix-soap-client
$ export DISNIX_TARGET_PROPERTY=targetEPR
```

The above environment variables instruct Disnix to use the *disnix-soap-client* executable and the *targetEPR* property from the infrastructure model as a connection string.

To authenticate ourselves, we must set the following environment variables with a username and password:

```
$ export DISNIX_SOAP_CLIENT_USERNAME=admin
$ export DISNIX_SOAP_CLIENT_PASSWORD=secret
```

The following command makes it possible to discover the machine's configuration using the *disnix-soap-client* and *DisnixWebService*:

```
$ disnix-capture-infra infra-bootstrap.nix
{
  "test1" = {
    properties = {
      "hostname" = "192.168.2.1";
      "system" = "x86_64-linux";
      "targetEPR" = "http://192.168.2.1/DisnixWebService/services/Dis
    };
    containers = {
      echo = {
      };
      fileset = {
      };
      process = {
      };
      supervisord-program = {
        "supervisordTargetDir" = "/etc/supervisor/conf.d";
      };
      wrapper = {
      };
      tomcat-webapplication = {
        "tomcatPort" = "8080";
        "catalinaBaseDir" = "/var/tomcat";
        "ajpPort" = "8009";
      };
      mysql-database = {
        "mysqlPort" = "3306";
        "mysqlUsername" = "root";
        "mysqlPassword" = "";
        "mysqlSocket" = "/var/run/mysqld/mysqld.sock";
      };
    };
    "system" = "x86_64-linux";
  }
  ;
}
```

After capturing the full infrastructure model, we can deploy the system with *disnix-env* if desired, using the *disnix-soap-client* to carry out all necessary remote deployment operations.

Miscellaneous: using Docker containers as light-weight virtual machines

As explained earlier in this blog post, the Nix process management framework is only a partial infrastructure deployment solution -- you still need to somehow obtain physical or virtual machines with a software distribution running the Nix package manager.

In a blog post written some time ago, I have explained that Docker containers are not virtual machines or even light-weight virtual machines.

In my previous blog post, I have shown that we can also deploy mutable Docker multi-process containers in which process instances can be upgraded without stopping the container.

The deployment workflow for upgrading mutable containers, is very machine-like -- NixOS has a similar workflow that consists of updating the machine configuration (*/etc/nixos/configuration.nix*) and running a single command-line instruction to upgrade machine (*nixos-rebuild switch*).

We can actually start using containers as VMs by adding another ingredient in the mix -- we can also assign static IP addresses to Docker containers.

With the following Nix expression, we can create a Docker image for a mutable container, using any of the processes models shown previously as the "machine's configuration":

```
let
  pkgs = import <nixpkgs> {};

  createMutableMultiProcessImage = import ../nix-processmgmt/nixproc/
    inherit pkgs;
  };
in
createMutableMultiProcessImage {
  name = "disnix";
  tag = "test";
  contents = [ pkgs.mc pkgs.disnix ];
  exprFile = ./processes.nix;
  interactive = true;
  manpages = true;
  processManager = "supervisord";
}
```

The *exprFile* in the above Nix expression refers to a previously shown processes

model, and the *processManager* the desired process manager to use, such as supervisord.

With the following command, we can build the image with Nix and load it into Docker:

```
$ nix-build
$ docker load -i result
```

With the following command, we can create a network to which our containers (with IP addresses) should belong:

```
$ docker network create --subnet=192.168.2.0/8 disnixnetwork
```

The above command creates a subnet with a prefix: *192.168.2.0* and allocates an 8-bit block for host IP addresses.

We can create and start a Docker container named: *containervm* using our previously built image, and assign it an IP address:

```
$ docker run --network disnixnetwork --ip 192.168.2.1 \
  --name containervm disnix:test
```

By default, Disnix uses SSH to connect to remote machines. With the following commands we can create a public-private key pair and copy the public key to the container:

```
$ ssh-keygen -t ed25519 -f id_test -N ""

$ docker exec containervm mkdir -m0700 -p /root/.ssh
$ docker cp id_test.pub containervm:/root/.ssh/authorized_keys
$ docker exec containervm chmod 600 /root/.ssh/authorized_keys
$ docker exec containervm chown root:root /root/.ssh/authorized_keys
```

On the coordinator machine, that carries out the deployment, we must add the private key to the SSH agent and configure the *disnix-ssh-client* to connect to the *disnix-service*:

```
$ ssh-add id_test
$ export DISNIX_REMOTE_CLIENT=disnix-client
```

By executing all these steps, *containervm* can be (mostly) used as if it were a virtual

machine, including connecting to it with an IP address over SSH.

Conclusion

In this blog post, I have described how the Nix process management framework can be used as a partial infrastructure deployment solution for Disnix. It can be used both for deploying the *disnix-service* (to facilitate multi-user installations) as well as deploying container providers: services that manage the life-cycles of services deployed by Disnix.

Moreover, the Nix process management framework makes it possible to do these deployments on all kinds of software distributions that can use the Nix package manager, including NixOS, conventional Linux distributions and other operating systems, such as macOS and FreeBSD.

If I had developed this solution a couple of years ago, it would probably have saved me many hours of preparation work for my first demo in my NixCon 2015 talk in which I wanted demonstrate that it is possible to deploy services to a heterogeneous network that consists of a NixOS, Ubuntu and Windows machine. Back then, I had to do all the infrastructure deployment tasks manually.

I also have to admit (but this statement is mostly based on my personal preferences, not facts), is that I find the functional style that the framework uses is IMO far more intuitive than the NixOS module system for certain service configuration aspects, especially for configuring container services and exposing them with Disnix and Dysnomia:

- Because every process instance is constructed from a constructor function that makes all instance parameters explicit, you are guarded against common configuration errors such as undeclared dependencies.

  For example, the *DisnixWebService*-enabled Apache Tomcat service requires access to the *dbus-service* providing the system bus. Not having this service in the processes model, causes a missing function parameter error.
- Function parameters in the processes model make it more clear that a process depends on another process and what that relationship may be. For example, with the *containerProviders* parameter it becomes IMO really clear that the *disnix-service* uses them as potential deployment targets for services deployed by Disnix.

  In comparison, the implementations of the Disnix and Dysnomia NixOS modules are far more complicated and monolithic -- the Dysnomia module has to figure for all potential container services deployed as part of a NixOS

configuration, their properties, convert them to Dysnomia configuration files, and configure the systemd configuration for the *disnix-service* for proper activation ordering.

The *wants* parameter (used for activation ordering) is just a list of strings, not knowing whether it contains valid references to services that have been deployed already.

Availability

The constructor functions for the services as well as the deployment examples described in this blog post can be found in the Nix process management services repository.

Future work

Slowly more and more of my personal use cases are getting supported by the Nix process management framework.

Moreover, the services repository is steadily growing. To ensure that all the services that I have packaged so far do not break, I really need to focus my work on a service test solution.