Stork

Release 0.17.0

May 07, 2021

Contents

`Stork` is an ISC project with the aim of delivering a use and monitoring dashboard for *ISC Kea DHCP*, and eventually for *ISC BIND 9*. It is the spiritual successor of the earlier projects *Kittiwake* and *Anterius*.

This is the reference guide for Stork version 0.17.0. Links to the most up-to-date version of this document, along with other documents for Stork, can be found on ISC's Stork project homepage or at Read the Docs .

CHAPTER 1

Overview

1.1 Goals

The goals of the Stork project are as follows:

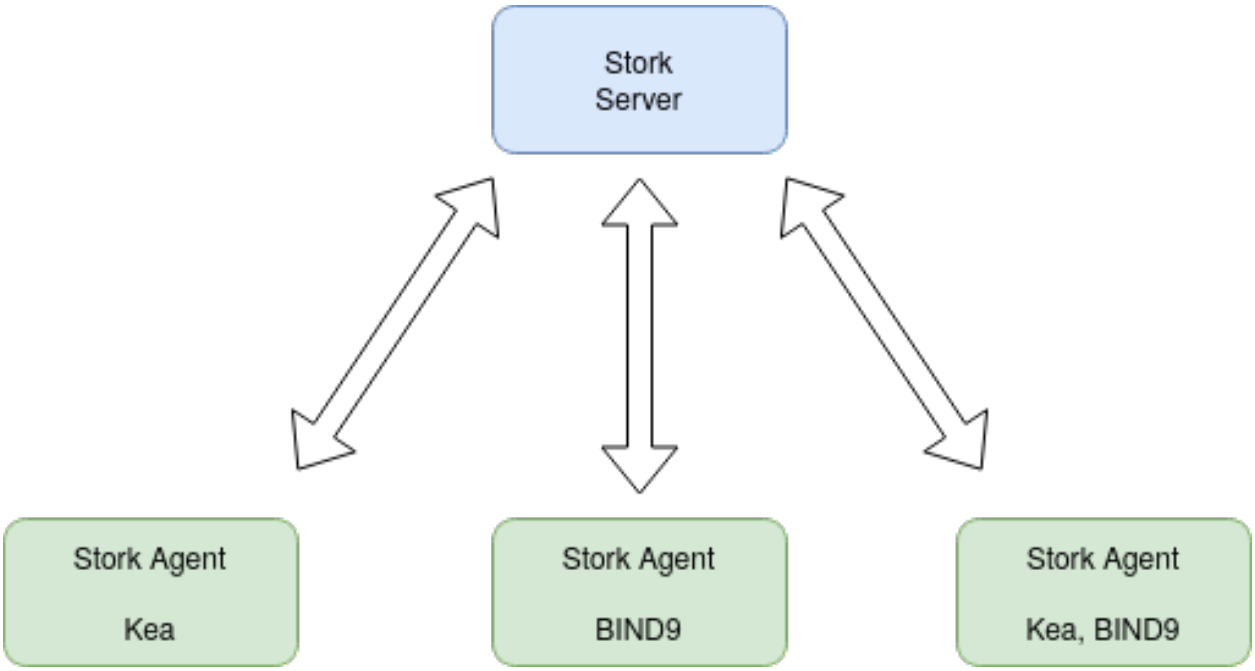
- to provide monitoring and insight into *ISC Kea DHCP* operations, and to eventually add them for *ISC BIND 9*
- to provide alerting mechanisms that indicate failures, fault conditions, and other unwanted events in *ISC Kea DHCP*, and eventually *ISC BIND 9*, services
- to permit easier troubleshooting of these services

1.2 Architecture

Stork is comprised of two components: the `Stork Server` and the `Stork Agent`.

The `Stork Agent` is installed along with *Kea DHCP* or *BIND 9* and interacts directly with those services. There may be many agents deployed in a network, one per machine.

The `Stork Server` is installed on a stand-alone machine. It connects to any indicated agents and indirectly (via those agents) interacts with the *Kea DHCP* and *BIND 9* services. It provides an integrated, centralized front end for interacting with these services. Only one `Stork Server` is deployed in a network.


Stork
Server
Stork Agent
Kea
Stork Agent
BIND9
Stork Agent
Kea, BIND9

CHAPTER 2

Installation

Stork can be installed from pre-built packages or from sources. The following sections describe both methods. Unless there's a good reason to compile from sources, installing from native deb or RPM packages is easier and faster.

2.1 Supported Systems

Stork is tested on the following systems:

- Ubuntu 18.04 and 20.04
- Fedora 31 and 32
- CentOS 7
- MacOS 10.15*

Note that MacOS is not and will not be officially supported. Many developers on ISC's team use Macs, so the goal is to keep Stork buildable on this platform.

The Stork server and agents are written in the Go language; the server uses a PostgreSQL database. In principle, the software can be run on any POSIX system that has a Go compiler and PostgreSQL. It is likely the software can also be built on other modern systems, but for the time being ISC's testing capabilities are modest. We encourage users to try running Stork on other OSes not on this list and report their findings to ISC.

2.2 Installation Prerequisites

The `Stork Agent` does not require any specific dependencies to run. It can be run immediately after installation.

Stork uses the *status-get* command to communicate with Kea, and therefore only works with a version of Kea that supports *status-get*, which was introduced in Kea 1.7.3 and backported to 1.6.3.

Stork requires the premium `Host Commands (host_cmds)` hooks library to be loaded by the Kea instance to retrieve host reservations stored in an external database. Stork does work without the Host Commands hooks library, but

will not be able to display host reservations. Stork can retrieve host reservations stored locally in the Kea configuration without any additional hooks libraries.

Stork requires the open source `Stat Commands (stat_cmds)` hooks library to be loaded by the Kea instance to retrieve lease statistics. Stork does work without the Stat Commands hooks library, but will not be able to show pool utilization and other statistics.

Stork uses Go implementation for handling TLS connections, certificates and keys. The secrets are stored in the PostgreSQL database, in the *secret* table.

For the `Stork Server`, a PostgreSQL database (https://www.postgresql.org/) version 11 or later is required. It may work with PostgreSQL 10, but this has not been tested. The general installation procedure for PostgreSQL is OS-specific and is not included here. However, please note that Stork uses pgcrypto extensions, which often come in a separate package. For example, a postgresql-crypto package is required on Fedora and postgresql12-contrib is needed on RHEL and CentOS.

These instructions prepare a database for use with the `Stork Server`, with the *stork* database user and *stork* password. Next, a database called *stork* is created and the *pgcrypto* extension is enabled in the database.

First, connect to PostgreSQL using *psql* and the *postgres* administration user. Depending on the system's configuration, this may require switching to the user *postgres* first, using the *su postgres* command.

```
$ psql postgres
psql (11.5)
Type "help" for help.
postgres=#
```

Then, prepare the database:

```
postgres=# CREATE USER stork WITH PASSWORD 'stork';
CREATE ROLE
postgres=# CREATE DATABASE stork;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON DATABASE stork TO stork;
GRANT
postgres=# \c stork
You are now connected to database "stork" as user "thomson".
stork=# create extension pgcrypto;
CREATE EXTENSION
```

Note: Make sure the actual password is stronger than 'stork', which is trivial to guess. Using default passwords is a security risk. Stork puts no restrictions on the characters used in the database passwords nor on their length. In particular, it accepts passwords containing spaces, quotes, double quotes, and other special characters.

2.3 Installing from Packages

Stork packages are stored in repositories located on the Cloudsmith service: https://cloudsmith.io/~isc/repos/stork/packages/. Both Debian/Ubuntu and RPM packages may be found there.

Detailed instructions for setting up the operating system to use this repository are available under the *Set Me Up* button on the Cloudsmith repository page.

It is possible to install both `Stork Agent` and `Stork Server` on the same machine. It is useful in small deployments with a single monitored machine to avoid setting up a dedicated system for the Stork Server. In those cases,

however, an operator must consider the possible impact of the Stork Server service on other services running on the same machine.

2.3.1 Installing the Stork Server

2.3.1.1 Installing on Debian/Ubuntu

The first step for both Debian and Ubuntu is:

```
$ curl -1sLf 'https://dl.cloudsmith.io/public/isc/stork/cfg/setup/bash.deb.sh' | sudo␣
↪bash
```

Next, install the `Stork Server` package:

```
$ sudo apt install isc-stork-server
```

2.3.1.2 Installing on CentOS/RHEL/Fedora

The first step for RPM-based distributions is:

```
$ curl -1sLf 'https://dl.cloudsmith.io/public/isc/stork/cfg/setup/bash.rpm.sh' | sudo␣
↪bash
```

Next, install the `Stork Server` package:

```
$ sudo dnf install isc-stork-server
```

If `dnf` is not available, `yum` can be used instead:

```
$ sudo yum install isc-stork-server
```

2.3.1.3 Setup

The following steps are common for Debian-based and RPM-based distributions using *systemd*.

Configure `Stork Server` settings in `/etc/stork/server.env`. The following settings are required for the database connection:

- STORK_DATABASE_HOST - the address of a PostgreSQL database; default is *localhost*
- STORK_DATABASE_PORT - the port of a PostgreSQL database; default is *5432*
- STORK_DATABASE_NAME - the name of a database; default is *stork*
- STORK_DATABASE_USER_NAME - the username for connecting to the database; default is *stork*
- STORK_DATABASE_PASSWORD - the password for the username connecting to the database

---

Note: All of the database connection settings have default values, but we strongly recommend protecting the database with a non-default and hard-to-guess password in the production environment. The *STORK_DATABASE_PASSWORD* setting must be adjusted accordingly.

The remaining settings pertain to the server's REST API configuration:

- STORK_REST_HOST - IP address on which the server listens

- STORK_REST_PORT - port number on which the server listens; default is *8080*
- STORK_REST_TLS_CERTIFICATE - a file with a certificate to use for secure connections
- STORK_REST_TLS_PRIVATE_KEY - a file with a private key to use for secure connections
- STORK_REST_TLS_CA_CERTIFICATE - a certificate authority file used for mutual TLS authentication
- STORK_REST_STATIC_FILES_DIR - a directory with static files served in the UI

With the settings in place, the `Stork Server` service can now be enabled and started:

```
$ sudo systemctl enable isc-stork-server
$ sudo systemctl start isc-stork-server
```

To check the status:

```
$ sudo systemctl status isc-stork-server
```

Note: By default, the `Stork Server` web service is exposed on port 8080 and can be tested using web browser at http://localhost:8080. To use a different IP address or port, please set the *STORK_REST_HOST* and *STORK_REST_PORT* variables in the `/etc/stork/stork.env` file.

The `Stork Server` can be configured to run behind an HTTP reverse proxy using *Nginx* or *Apache*. The `Stork Server` package contains an example configuration file for *Nginx*, in */usr/share/stork/examples/nginx-stork.conf*.

2.3.2 Installing the Stork Agent

There are two ways to install packaged `Stork Agent` on a monitored machine. The first method is to use the Cloudsmith repository like in the case of the `Stork Server` installation. The second method is to use an installation script provided by the `Stork Server` which downloads the agent packages embedded in the server package. The second installation method is supported since the Stork 0.15.0 release. The preferred installation method depends on the selected agent registration type. Supported registration methods are described in the *Securing Connections Between Stork Server and Stork Agents*.

2.3.2.1 Agent Configuration Settings

The following are the `Stork Agent` configuration settings available in the `/etc/stork/agent.env` after installing the package.

The general settings:

- STORK_AGENT_ADDRESS - the IP address of the network interface which `Stork Agent` should use to receive the connections from the server; default is *0.0.0.0* (i.e. listen on all interfaces)
- STORK_AGENT_PORT - the port number the agent should use to receive the connections from the server; default is *8080*
- STORK_AGENT_LISTEN_STORK_ONLY - enable Stork functionality only, i.e. disable Prometheus exporters; default is false
- STORK_AGENT_LISTEN_PROMETHEUS_ONLY - enable Prometheus exporters only, i.e. disable Stork functionality; default is false

The following settings are specific to the Prometheus exporters:

- STORK_AGENT_PROMETHEUS_KEA_EXPORTER_ADDRESS - the IP address or hostname the agent should use to receive the connections from Prometheus fetching Kea statistics; default is *0.0.0.0*

- STORK_AGENT_PROMETHEUS_KEA_EXPORTER_PORT - the port the agent should use to receive the connections from Prometheus fetching Kea statistics; default is *9547*
- STORK_AGENT_PROMETHEUS_KEA_EXPORTER_INTERVAL - specifies how often the agent collects stats from Kea, in seconds; default is *10*
- STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_ADDRESS - the IP address or hostname the agent should use to receive the connections from Prometheus fetching BIND9 statistics; default is *0.0.0.0* to listen on for incoming Prometheus connection; default is *0.0.0.0*
- STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_PORT - the port the agent should use to receive the connections from Prometheus fetching BIND9 statistics; default is *9119*
- STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_INTERVAL - specifies how often the agent collects stats from BIND9, in seconds; default is *10*

The last setting is used only when `Stork Agents` register in the `Stork Server` using agent token:

- STORK_AGENT_SERVER_URL - Stork Server URL used by the agent to send REST commands to the server during agent registration

2.3.2.2 Securing Connections Between Stork Server and Stork Agents

Connections between the server and the agents are secured using standard cryptography solutions, i.e. PKI and TLS.

The server generates the required keys and certificates during its first startup. They are used to establish safe, encrypted connections between the server and the agents with authentication of both ends of these connections. The agents use the keys and certificates generated by the server to create agent-side keys and certificates during the agents' registration procedure described in the next sections. The private key and CSR certificate generated by an agent and signed by the server are used for authentication and connection encryption.

An agent can be registered in the server using one of the two supported methods:

1. using agent token,
2. using server token.

In the first case, an agent generates a token and passes it to the server requesting registration. The server associates the token with the particular agent. A Stork super admin must approve the registration request in the web UI, ensuring that the token displayed in the UI matches the agent's token in the logs. The `Stork Agent` is typically installed from the Cloudsmith repository when this registration method is used.

In the second registration method, a server generates a token common for all new registrations. The super admin must copy the token from the UI and paste it into the agent's terminal during the interactive agent registration procedure. This registration method does not require any additional approval of the agent's registration request in the web UI. If the pasted server token is correct, the agent should be authorized in the UI when the interactive registration completes. The `Stork Agent` is typically installed using a script that downloads the agent packages embedded in the server when this registration method is used.

The applicability of the two methods is described in *Registration Methods Summary*.

The installation and registration process using both methods are described in the subsequent sections.

2.3.2.3 Installation from Cloudsmith and Registration with an Agent Token

This section describes installing an agent from the Cloudsmith repository and performing the agent's registration using an agent token.

The Stork Agent installation steps are similar to the Stork Server installation steps described in *Installing on Debian/Ubuntu* and *Installing on CentOS/RHEL/Fedora*. Use one of the following commands depending on your Linux distribution:

```
$ sudo apt install isc-stork-agent
```

```
$ sudo dnf install isc-stork-agent
```

in place of the commands installing the server.

Next, specify the required settings in the /etc/stork/agent.env file. The STORK_SERVER_URL should be the URL on which the server receives the REST connections, e.g. http://stork-server.example.org:8080. The STORK_AGENT_ADDRESS should point to the agent's address (or name), e.g. stork-agent.example.org. Finally, a non-default agent port can be specified with the STORK_AGENT_PORT.

Note: Even though the examples provided in this documentation use the http scheme, we highly recommend using secure protocols in the production environments. We use http in the examples because it usually makes it easier to start testing the software and eliminate all issues unrelated to the use of https before it is enabled.

Start the agent service:

```
$ sudo systemctl enable isc-stork-agent
$ sudo systemctl start isc-stork-agent
```

To check the status:

```
$ sudo systemctl status isc-stork-agent
```

You should expect the following log messages when the agent successfully sends the registration request to the server:

```
machine registered
stored agent signed cert and CA cert
registration completed successfully
```

A server administrator must approve the registration request via the web UI before the machine can be monitored. Visit the Services -> Machines page. Click the Unauthorized button located above the list of machines on the right side. This list contains all machines pending registration approval. Before authorizing the machine, ensure that the agent token displayed on this list is the same as the agent token in the agent's logs or the /var/lib/stork-agent/tokens/agent-token.txt file. If they match, click on the Action button and select Authorize. The machine should now be visible on the list of authorized machines.

2.3.2.4 Installation with a Script and Registration with a Server Token

This section describes installing an agent using a script and packages downloaded from the Stork Server and performing the agent's registration using a server token.

Open Stork in the web browser and log in as a user from the super admin group. Select Services and then Machines from the menu. Click on the How to Install Agent on New Machine button to display the agent installation instructions. Copy-paste the commands from the displayed window into the terminal on the machine where the agent is installed. These commands are also provided here for convenience:

```
$ wget http://stork.example.org:8080/stork-install-agent.sh
$ chmod a+x stork-install-agent.sh
$ sudo ./stork-install-agent.sh
```

Please note that this document provides an example URL of the `Stork Server` and it must be replaced with a server URL used in the particular deployment.

The script downloads an OS specific agent package from the `Stork Server` (deb or RPM), installs the package, and starts the agent's registration procedure.

In the agent machine's terminal, a prompt for a server token is presented:

```
>>>> Server access token (optional):
```

The server token is available for a super admin user after clicking on the `How to Install Agent on New Machine` button in the `Services -> Machines`. Copy the server token from the dialog box and paste it in the prompt displayed on the agent machine.

The following prompt appears next:

```
>>>> IP address or FQDN of the host with Stork Agent (the Stork Server will use it to␣
↪connect to the Stork Agent):
```

Specify an IP address or FQDN which the server should use to reach out to an agent via the secure gRPC channel.

When asked for the port:

```
>>>> Port number that Stork Agent will use to listen on [8080]:
```

specify the port number for the gRPC connections, or hit Enter if the default port 8080 matches your settings.

If the registration is successful, the following messages are displayed:

```
machine ping over TLS: OK
registration completed successfully
```

Unlike the *Installation from Cloudsmith and Registration with an Agent Token*, this registration method does not require approval via the web UI. The machine should be already listed among the authorized machines.

2.3.2.5 Installation with a Script and Registration with an Agent Token

This section describes installing an agent using a script and packages downloaded from the `Stork Server` and performing the agent's registration using an agent token. It is an interactive procedure alternative to the procedure described in *Installation from Cloudsmith and Registration with an Agent Token*.

Start the interactive registration procedure following the steps in the *Installation with a Script and Registration with a Server Token*.

In the agent machine's terminal, a prompt for a server token is presented:

```
>>>> Server access token (optional):
```

Because this registration method does not use the server token, do not type anything in this prompt. Hit Enter to move on.

The following prompt appears next:

```
>>>> IP address or FQDN of the host with Stork Agent (the Stork Server will use it to␣
↪connect to the Stork Agent):
```

Specify an IP address or FQDN which the server should use to reach out to an agent via the secure gRPC channel.

When asked for the port:

```
>>>> Port number that Stork Agent will use to listen on [8080]:
```

specify the port number for the gRPC connections, or hit Enter if the default port 8080 matches your settings.

You should expect the following log messages when the agent successfully sends the registration request to the server:

```
machine registered
stored agent signed cert and CA cert
registration completed successfully
```

Similar to *Installation from Cloudsmith and Registration with an Agent Token*, the agent's registration request must be approved in the UI to start monitoring the newly registered machine.

2.3.2.6 Installation from Cloudsmith and Registration with a Server Token

This section describes installing an agent from the Cloudsmith repository and performing the agent's registration using a server token. It is an alternative to the procedure described in *Installation with a Script and Registration with a Server Token*.

The `Stork Agent` installation steps are similar to the `Stork Server` installation steps described in *Installing on Debian/Ubuntu* and *Installing on CentOS/RHEL/Fedora*. Use one of the following commands depending on your Linux distribution:

```
$ sudo apt install isc-stork-agent
```

```
$ sudo dnf install isc-stork-agent
```

in place of the commands installing the server.

Start the agent service:

```
$ sudo systemctl enable isc-stork-agent
$ sudo systemctl start isc-stork-agent
```

To check the status:

```
$ sudo systemctl status isc-stork-agent
```

Start the interactive registration procedure with the following command:

```
$ su stork-agent -s /bin/sh -c 'stork-agent register -u http://stork.example.org'
```

where the last parameter should be the appropriate Stork server's URL.

Follow the same registration steps as described in the *Installation with a Script and Registration with a Server Token*.

2.3.2.7 Registration Methods Summary

Stork supports two different agents' registration methods described above. Both methods can be used interchangeably, and it is often a matter of preference which one the administrator selects. However, it is worth mentioning that the agent token registration may be more suitable in some situations. This method requires a server URL, agent address (or name), and agent port as registration settings. If they are known upfront, it is possible to prepare a system (or container) image with the agent offline. After starting the image, the agent will send the registration request to the server and await authorization in the web UI.

The agent registration with the server token is always manual. It requires copying the token from the web UI, logging into the agent, and pasting the token. Therefore, the registration using the server token is not appropriate when it is impossible or awkward to access the machine's terminal, e.g. in Docker. On the other hand, the registration using the server token is more straightforward because it does not require unauthorized agents' approval via the web UI.

If the server token leaks, it poses a risk that rogue agents register. In that case, the administrator should regenerate the token to prevent the uncontrolled registration of new agents. Regeneration of the token does not affect already registered agents. The new token must be used for the new registrations.

The server token can be regenerated in the `How to Install Agent on New Machine` dialog box available after entering the `Services -> Machines` page.

2.3.2.8 Agent Setup Summary

After successful agent setup, the agent periodically tries to detect installed Kea DHCP or BIND 9 services on the system. If it finds them, they are reported to the `Stork Server` when it connects to the agent.

Further configuration and usage of the `Stork Server` and the `Stork Agent` are described in the *Using Stork* chapter.

2.3.3 Upgrading

Due to the new security model introduced with TLS in Stork 0.15.0 release, upgrades from versions 0.14.0 and earlier require registering the agents from scratch.

Server upgrade procedure looks the same as the installation procedure.

First, install the new packages on the server. Installation scripts in deb/RPM package will perform the required database and other migrations.

2.4 Installing From Sources

2.4.1 Compilation Prerequisites

Usually, it is more convenient to install Stork using native packages. See *Supported Systems* and *Installing from Packages* for details regarding supported systems. However, the sources can also be built separately.

The dependencies that need to be installed to build `Stork` sources are:

- Rake
- Java Runtime Environment (only if building natively, not using Docker)
- Docker (only if running in containers; this is needed to build the demo)

Other dependencies are installed automatically in a local directory by Rake tasks. This does not require root privileges. If the demo environment will be run, Docker is needed but not Java (Docker will install Java within a container).

For details about the environment, please see the Stork wiki at https://gitlab.isc.org/isc-projects/stork/-/wikis/Install .

2.4.2 Download Sources

The Stork sources are available on the ISC GitLab instance: https://gitlab.isc.org/isc-projects/stork.

To get the latest sources invoke:

```
$ git clone https://gitlab.isc.org/isc-projects/stork
```

2.4.3 Building

There are several components of `Stork`:

- `Stork Agent` - this is the binary *stork-agent*, written in Go
- `Stork Server` - this is comprised of two parts: - *backend service* - written in Go - *frontend* - an *Angular* application written in Typescript

All components can be built using the following command:

```
$ rake build_all
```

The agent component is installed using this command:

```
$ rake install_agent
```

and the server component with this command:

```
$ rake install_server
```

By default, all components are installed to the *root* folder in the current directory; however, this is not useful for installation in a production environment. It can be customized via the `DESTDIR` variable, e.g.:

```
$ sudo rake install_server DESTDIR=/usr
```

2.5 Database Migration Tool (optional)

Optional step: to initialize the database directly, the migrations tool must be built and used to initialize and upgrade the database to the latest schema. However, this is completely optional, as the database migration is triggered automatically upon server startup. This is only useful if for some reason it is desirable to set up the database but not yet run the server. In most cases this step can be skipped.

```
$ rake build_migrations
$ backend/cmd/stork-db-migrate/stork-db-migrate init
$ backend/cmd/stork-db-migrate/stork-db-migrate up
```

The up and down commands have an optional *-t* parameter that specifies the desired schema version. This is only useful when debugging database migrations.

```
$ # migrate up version 25
$ backend/cmd/stork-db-migrate/stork-db-migrate up -t 25
$ # migrate down back to version 17
$ backend/cmd/stork-db-migrate/stork-db-migrate down -t 17
```

Note that the server requires the latest database version to run, always runs the migration on its own, and will refuse to start if the migration fails for any reason. The migration tool is mostly useful for debugging problems with migration or migrating the database without actually running the service. For complete reference, see the manual page here: *stork-db-migrate - Stork database migration tool*.

To debug migrations, another useful feature is SQL tracing using the *–db-trace-queries* parameter. It takes either “all” (trace all SQL operations, including migrations and run-time) or “run” (just trace run-time operations, skip migrations).

If specified without any parameters, "all" is assumed. With it enabled, *stork-db-migrate* prints out all its SQL queries on stderr. For example, these commands can be used to generate an SQL script that updates the schema. Note that for some migrations, the steps are dependent on the contents of the database, so this is not a universal Stork schema. This parameter is also supported by the `Stork Server`.

```
$ backend/cmd/stork-db-migrate/stork-db-migrate down -t 0
$ backend/cmd/stork-db-migrate/stork-db-migrate up --db-trace-queries 2> stork-schema.
↪txt
```

2.6 Integration With Prometheus and Grafana

Stork can optionally be integrated with Prometheus, an open source monitoring and alerting toolkit, and Grafana, an easy-to-view analytics platform for querying, visualization, and alerting. Grafana requires external data storage. Prometheus is currently the only environment supported by both Stork and Grafana. It is possible to use Prometheus without Grafana, but using Grafana requires Prometheus.

2.6.1 Prometheus Integration

The Stork agent, by default, makes the Kea (and eventually, BIND 9) statistics available in a format understandable by Prometheus (it works as a Prometheus exporter, in Prometheus nomenclature). If Prometheus server is available, it can be configured to monitor Stork agents. To enable Stork agent monitoring, the `prometheus.yml` (which is typically stored in /etc/prometheus/, but this may vary depending on the installation) must be edited to add the following entries there:

```
# statistics from Kea
- job_name: 'kea'
  static_configs:
    - targets: ['agent-kea.example.org:9547', 'agent-kea6.example.org:9547', ... ]

# statistics from bind9
- job_name: 'bind9'
  static_configs:
    - targets: ['agent-bind9.example.org:9119', 'another-bind9.example.org:9119', ...␣
↪]
```

By default, the Stork agent exports Kea data on TCP port 9547 (and BIND 9 data on TCP port 9119). This can be configured using command-line parameters, or the Prometheus export can be disabled altogether. For details, see the stork-agent manual page at *stork-agent - Stork agent that monitors BIND 9 and Kea services*.

After restarting, the Prometheus web interface can be used to inspect whether statistics are exported properly. Kea statistics use the `kea_` prefix (e.g. kea_dhcp4_addresses_assigned_total); BIND 9 statistics will eventually use the `bind_` prefix (e.g. bind_incoming_queries_tcp).

2.6.2 Grafana Integration

Stork provides several Grafana templates that can easily be imported. Those are available in the `grafana/` directory of the Stork source code. The currently available templates are *bind9-resolver.json* and *kea-dhcp4.json*. Grafana integration requires three steps:

1. Prometheus must be added as a data source. This can be done in several ways, including via the user interface to edit the Grafana configuration files. This is the easiest method; for details, see the Grafana documentation about Prometheus integration. Using the Grafana user interface, select Configuration, select Data Sources, click "Add data source," and choose Prometheus, and then specify the necessary parameters to connect to the Prometheus instance.

In test environments, the only really necessary parameter is the URL, but authentication is also desirable in most production deployments.

2. Import the existing dashboard. In the Grafana UI, click Dashboards, then Manage, then Import, and select one of the templates, e.g. *kea-dhcp4.json*. Make sure to select the Prometheus data source added in the previous step. Once imported, the dashboard can be tweaked as needed.

3. Once Grafana is configured, go to the Stork user interface, log in as super-admin, click Settings in the Configuration menu, and then add the URLs to Grafana and Prometheus that point to the installations. Once this is done, Stork will be able to show links for subnets leading to specific subnets.

Alternatively, a Prometheus data source can be added by editing *datasource.yaml* (typically stored in */etc/grafana*, but this may vary depending on the installation) and adding entries similar to this one:

```
datasources:
- name: Stork-Prometheus instance
  type: prometheus
  access: proxy
  url: http://prometheus.example.org:9090
  isDefault: true
  editable: false
```

Also, the Grafana dashboard files can be copied to */var/lib/grafana/dashboards/* (again, this may vary depending on the installation).

Example dashboards with some live data can be seen in the Stork screenshots gallery .

CHAPTER 3

Using Stork

This section describes how to use the features available in `Stork`. To connect to `Stork`, use a web browser and connect to port 8080. If Stork is running on a localhost, it can be reached by navigating to http://localhost:8080.

3.1 Managing Users

A default administrator account is created upon initial installation of Stork. It can be used to sign in to the system via the web UI, with the username `admin` and password `admin`.

To see a list of existing users, click on the `Configuration` menu and choose `Users` . There will be at least one user, `admin`.

To add a new user, click `Create User Account`. A new tab opens to specify the new account parameters. Some fields have specific restrictions:

- Username can consist of only letters, numbers, and an underscore (_).
- The e-mail field is optional, but if specified, it must be a well-formed e-mail address.
- The firstname and lastname fields are mandatory.
- The password must only contain letters, digits, @, ., !, +, or -, and must be at least eight characters long.

Currently, users are associated with one of the two predefined groups (roles), i.e., `super-admin` or `admin`, which must be selected when the user account is created. Both types of users can view Stork status screens, edit interval and reporting configuration settings, and add/remove machines for monitoring. `super-admin` users can also create and manage user accounts.

Once the new user account information has been specified and all requirements are met, the `Save` button becomes active and the new account can be enabled.

3.2 Changing a User Password

An initial password is assigned by the administrator when a user account is created. Each user should change the password when first logging into the system. To change the password, click on the `Profile` menu and choose `Settings` to display the user profile information. Click on `Change password` in the menu bar on the left and specify the current password in the first input box. The new password must be entered and confirmed in the second and third input boxes, and must meet the password requirements specified in the previous section. When all entered data is valid, the `Save` button is activated to change the password.

3.3 Configuration Settings

It is possible to control some of the Stork configuration settings from the web UI. Click on the `Configuration` menu and choose `Settings`. There are two classes of settings available: Intervals and Grafana & Prometheus.

Intervals settings specify the configuration of "pullers." A puller is a mechanism in Stork which triggers a specific action at the specified interval. Each puller has its own specific action and interval. The puller interval is specified in seconds and designates a time period between the completion of the previously invoked action and the beginning of the next invocation of this action. For example, if the Kea Hosts Puller Interval is set to 10 seconds and it takes five seconds to pull the hosts information, the time period between the starts of the two consecutive attempts to pull the hosts information will be 15 seconds.

The pull time varies between deployments and depends on the amount of information pulled, network congestion, and other factors. The interval setting guarantees that there is a constant idle time between any consecutive attempts.

The Grafana & Prometheus settings currently allow for specifying the URLs of the Prometheus and Grafana instances used with Stork.

3.4 Connecting and Monitoring Machines

3.4.1 Monitoring a Machine

Monitoring of registered machines is accomplished via the `Services` menu, under `Machines`. A list of currently registered machines is displayed, with multiple pages available if needed.

A filtering mechanism that acts as an omnibox is available. Via a typed string, Stork can search for an address, agent version, hostname, OS, platform, OS version, kernel version, kernel architecture, virtualization system, or host-id field.

The state of a machine can be inspected by clicking its hostname; a new tab opens with the machine's details. Multiple tabs can be open at the same time, and clicking Refresh updates the available information.

The machine state can also be refreshed via the `Action` menu. On the `Machines` list, each machine has its own menu; click on the triple-lines button at the right side and choose the Refresh option.

3.4.2 Deleting a Machine

To stop monitoring a machine, go to the `Machines` list, find the machine to stop monitoring, click on the triple-lines button at the right side, and choose `Delete`. This will terminate the connection between the Stork server and the agent running on the machine, and the server will no longer monitor that machine; however, the Stork agent process will continue running. Complete shutdown of a Stork agent process must be done manually, e.g. by connecting to the machine using ssh and stopping the agent there. The preferred way to achieve that is to issue the `killall stork-agent` command.

3.5 Monitoring Applications

3.5.1 Application Status

Kea DHCP applications discovered on connected machines are listed via the top-level menu bar, under `Services`. The list view includes the application version, application status, and some machine details. The `Action` button is also available, to refresh the information about the application.

The application status displays a list of daemons belonging to the application. Several daemons may be presented in the application status column, typically: DHCPv4, DHCPv6, DDNS, and CA (Kea Control Agent).

Eventually, when support for BIND 9 is added, the Stork agent will look for `named` in the process list and parse the configuration file that is given with the `-c` argument. If the `named` process is started without a specific configuration file, the Stork agent will default to `/etc/bind/named.conf`.

Stork uses `rndc` to retrieve the application status. It looks for the `controls` statement in the configuration file, and uses the first listed control point for monitoring the application.

Furthermore, the Stork agent can be used as a Prometheus exporter. Stork is able to do so if `named` is built with `json-c` because it gathers statistics via the JSON statistics API. The `named.conf` file must have a `statistics-channel` configured; the exporter queries the first listed channel. Stork is able to export the most metrics if `zone-statistics` is set to `full` in the `named.conf` configuration.

For Kea, the listed daemons are those that Stork finds in the Control Agent (CA) configuration file. A warning sign is displayed for any daemons from the CA configuration file that are not running. When the Kea installation is simply using the default CA configuration file, which includes configuration of daemons that are never intended to be launched, it is recommended to remove (or comment out) those configurations to eliminate unwanted warnings from Stork about inactive daemons.

3.5.2 Friendly App Names

Every app connected to Stork is assigned a default name. For example, if a Kea app runs on the machine `abc.example.org`, this app's default name will be `kea@abc.example.org`. Similarly, if a BIND9 app runs on the machine with address `192.0.2.3`, the resulting app name will be `bind9@192.0.2.3`. If multiple apps of a given type run on the same machine, a postfix with a unique identifier is appended to the duplicated names, e.g. `bind9@192.0.2.3%56`.

The default app names are unique so that the user can distinguish them in the dashboard, apps list, events panel, and other views. However, the default names may become lengthy when machines names consist of fully qualified domain names. When machines' IP addresses are used instead of FQDNs, the app names are less meaningful for someone not familiar with addressing in the managed network. In these cases, users may prefer replacing the default app names with more descriptive ones.

Suppose there are two DHCP servers in the network, one on the first floor, second on the second floor of the building. A user may assign `Floor 1 DHCP` and `Floor 2 DHCP` names to the respective DHCP servers in this case. The new names need not have the same pattern as the default names and may contain whitespace. The @ character is not required, but if it is present, the part of the name following this character (and before an optional % character) must be an address or name of the machine monitored in Stork. The following names: `dhcp-server@floor1%123` and `dhcp-server@floor1`, are invalid unless `floor1` is a monitored machine's name. The special notation using two consecutive @ characters can be used to suppress this check. The `dhcp-server@@floor1` is a valid name even if `floor1` is not a machine's name. In this case, `floor1` can be a physical location of the DHCP server in a building.

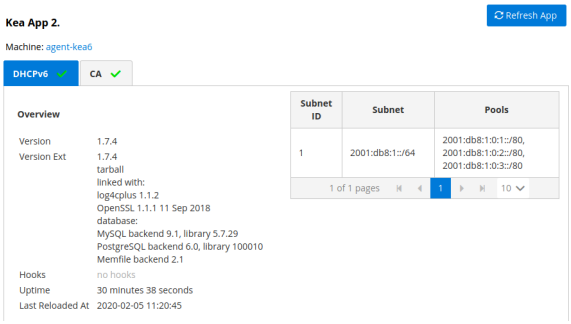
To modify an app's name, navigate to the selected app's view. For example, select `Services` from the top menu bar and then click `Kea Apps`. Select an app from the presented apps list. Locate and click the pencil icon next to the app name in the app view. In the displayed dialog box, type the new app name. If the specified name is valid, the `Rename`

button is enabled. Click this button to submit the new name. The `Rename` button is disabled if the name is invalid. In this case, a hint is displayed informing about issues with the new name.

3.5.3 IPv4 and IPv6 Subnets per Kea Application

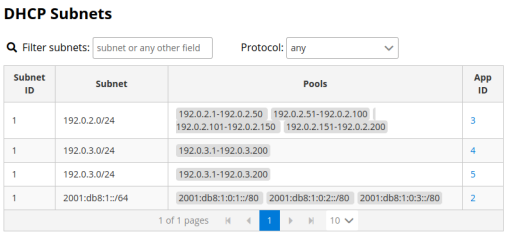
One of the primary configuration aspects of any network is the layout of IP addressing. This is represented in Kea with IPv4 and IPv6 subnets. Each subnet represents addresses used on a physical link. Typically, certain parts of each subnet ("pools") are delegated to the DHCP server to manage. Stork is able to display this information.

One way to inspect the subnets and pools within Kea is by looking at each Kea application to get an overview of what configurations a specific Kea application is serving. A list of configured subnets on that specific Kea application is displayed. The following picture shows a simple view of the Kea DHCPv6 server running with a single subnet, with three pools configured in it.



3.5.4 IPv4 and IPv6 Subnets in the Whole Network

It is convenient to see the complete overview of all subnets configured in the network that are being monitored by Stork. Once at least one machine with the Kea application running is added to Stork, click on the `DHCP` menu and choose `Subnets` to see all available subnets. The view shows all IPv4 and IPv6 subnets with the address pools and links to the applications that are providing them. An example view of all subnets in the network is presented in the figure below.



Stork provides filtering capabilities; it is possible to choose whether to see IPv4 only, IPv6 only, or both. There is also an omnisearch box available where users can type a search string. Note that for strings of four characters or more, the filtering takes place automatically, while shorter strings require the user to hit Enter. For example, in the above example it is possible to show only the first (192.0.2.0/24) subnet by searching for the *0.2* string. One can also search for specific pools, and easily filter the subnet with a specific pool, by searching for part of the pool ranges, e.g. *3.200*.

Stork displays pool utilization for each subnet, with the absolute number of addresses allocated and usage percentage. There are two thresholds: 80% (warning; the pool utilization bar turns orange) and 90% (critical; the pool utilization bar turns red).

3.5.5 IPv4 and IPv6 Networks

Kea uses the concept of a shared network, which is essentially a stack of subnets deployed on the same physical link. Stork retrieves information about shared networks and aggregates it across all configured Kea servers. The `Shared Networks` view allows for the inspection of networks and the subnets that belong in them. Pool utilization is shown for each subnet.

3.5.6 Host Reservations

Kea DHCP servers can be configured to assign static resources or parameters to the DHCP clients communicating with the servers. Most commonly these resources are the IP addresses or delegated prefixes. However, Kea also allows for assigning hostnames, PXE boot parameters, client classes, DHCP options, and other parameters. The mechanism by which a given set of resources and/or parameters is associated with a given DHCP client is called "host reservations."

A host reservation consists of one or more DHCP identifiers used to associate the reservation with a client, e.g. MAC address, DUID, or client identifier; and a collection of resources and/or parameters to be returned to the client if the client's DHCP message is associated with the host reservation by one of the identifiers. Stork can detect existing host reservations specified both in the configuration files of the monitored Kea servers and in the host database backends accessed via the Kea Host Commands premium hooks library. At present, Stork provides no means to update or delete host reservations.

All reservations detected by Stork can be listed by selecting the `DHCP` menu option and then selecting `Hosts`.

The first column in the presented view displays one or more DHCP identifiers for each host in the format `hw-address=0a:1b:bd:43:5f:99`, where `hw-address` is the identifier type. In this case, the identifier type is the MAC address of the DHCP client for which the reservation has been specified. Supported identifier types are described in the following sections of the Kea Administrator Reference Manual (ARM): Host Reservation in DHCPv4 and Host Reservation in DHCPv6. If multiple identifiers are present for a reservation, the reservation is assigned when at least one of the identifiers matches the received DHCP packet.

The second column, `IP Reservations`, includes the static assignments of the IP addresses and/or delegated prefixes to the clients. There may be one or more IP reservations for each host.

The `Hostname` column contains an optional hostname reservation, i.e., the hostname assigned to the particular client by the DHCP servers via the Hostname or Client FQDN option.

The `Global/Subnet` column contains the prefixes of the subnets to which the reserved IP addresses and prefixes belong. If the reservation is global, i.e., is valid for all configured subnets of the given server, the word "global" is shown instead of the subnet prefix.

Finally, the `App Name` column includes one or more links to Kea applications configured to assign each reservation to the client. The number of applications is typically greater than one when Kea servers operate in the High Availability setup. In this case, each of the HA peers uses the same configuration and may allocate IP addresses and delegated prefixes to the same set of clients, including static assignments via host reservations. If HA peers are configured correctly, the reservations they share will have two links in the `App Name` column. Next to each link there is a little label indicating whether the host reservation for the given server has been specified in its configuration file or a host database (via the Host Commands premium hooks library).

The `Filter hosts` input box is located above the Hosts table. It allows the hosts to be filtered by identifier types, identifier values, IP reservations, and hostnames, and by globality, i.e., `is:global` and `not:global`. When filtering by DHCP identifier values, it is not necessary to use colons between the pairs of hexadecimal digits. For example, the reservation `hw-address=0a:1b:bd:43:5f:99` will be found whether the filtering text is `1b:bd:43` or `1bbd43`.

3.5.7 Sources of Host Reservations

There are two ways to configure the Kea servers to use host reservations. First, the host reservations can be specified within the Kea configuration files; see Host Reservation in DHCPv4 for details. The other way is to use a host database backend, as described in Storing Host Reservations in MySQL, PostgreSQL, or Cassandra. The second solution requires the given Kea server to be configured to use the `host_cmds` premium hooks library. This library implements control commands used to store and fetch the host reservations from the host database which the Kea server is connected to. If the `host_cmds` hooks library is not loaded, Stork only presents the reservations specified within the Kea configuration files.

Stork periodically fetches the reservations from the host database backends and updates them in the local database. The default interval at which Stork refreshes host reservation information is set to 60 seconds. This means that an update in the host reservation database will not be visible in Stork until up to 60 seconds after it was applied. This interval is currently not configurable.

Note: The list of host reservations must be manually refreshed by reloading the browser page to see the most recent updates fetched from the Kea servers.

3.5.8 Leases Search

Stork has a utility to search DHCP leases on monitored Kea servers. It is helpful for troubleshooting issues with a particular IP address or delegated prefix. It is also helpful in resolving lease allocation issues for certain DHCP clients. The search mechanism utilizes Kea control commands to find leases on the monitored servers. An operator must ensure that Kea servers on which he intends to search the leases have the lease_cmds hooks library loaded. Stork does not search leases on the Kea instances without this library.

The leases search is available via the `DHCP -> Leases Search` menu. Type one of the searched lease properties in the search box:

- IPv4 address, e.g. `192.0.2.3`
- IPv6 address or delegated prefix without prefix length, `2001:db8::1`
- MAC address, e.g. `01:02:03:04:05:06`
- DHCPv4 Client Identifier, e.g. `01:02:03:04`
- DHCPv6 DUID, e.g. `00:02:00:00:00:04:05:06:07`
- Hostname, e.g. `myhost.example.org`

All identifier types can also be specified using the notation with spaces, e.g. 01 02 03 04 05 06, or the notation with hexadecimal digits only, e.g. 010203040506.

To search all declined leases, type `state:declined`. Beware that this query may return a large result if there are many declined leases, and in this case, the query processing time may also increase.

Searching using partial text is currently unsupported. For example: searching by partial IPv4 address `192.0.2` is not accepted by the search box. Partial MAC address `01:02:03` is accepted but will return no results. Specify the complete MAC address instead, e.g. `01:02:03:04:05:06`. Searching leases in states other than declined is also unsupported. For example, the text `state:expired-reclaimed` is not accepted by the search box.

The search utility automatically recognizes the specified lease type property and communicates with the Kea servers to find leases using appropriate commands. Each search attempt may result in several commands to multiple Kea servers. Therefore, it may take several seconds or more before Stork displays the search results. Suppose some Kea servers are unavailable or return an error. In that case, Stork shows leases found on the servers which returned success status, and displays a warning message containing the list of Kea servers that returned an error.

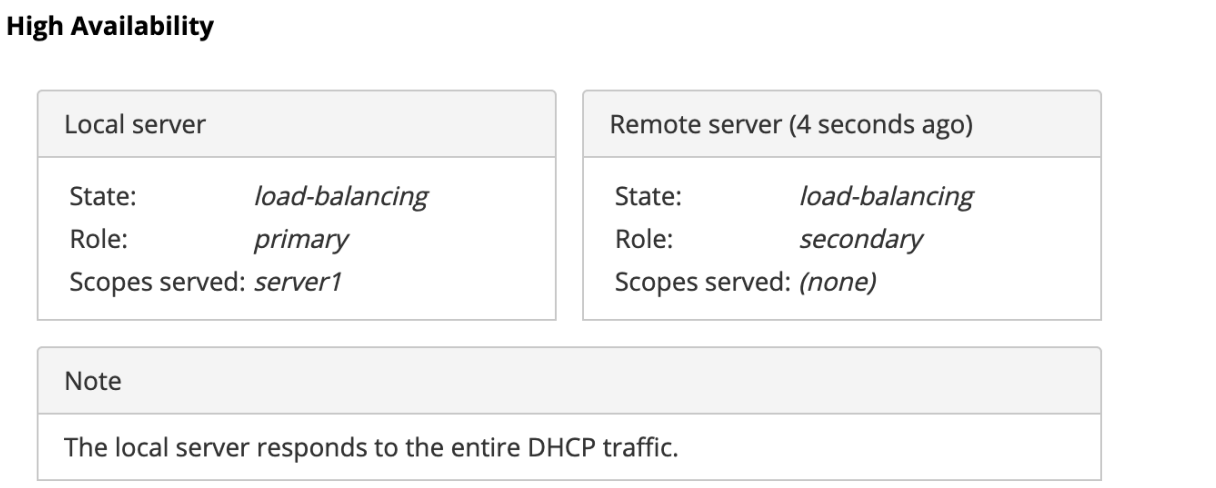
If the same lease is found on two or more Kea servers, the results list contains all that lease occurrences. For example, if there is a pair of servers cooperating via HA hooks library, the servers exchange the lease information, and each of them maintains a copy of the lease database. In that case, the lease search on these servers typically returns two occurrences of the same lease.

To display the detailed lease information click the expand button (>) in the first column for the selected lease.

3.5.9 Kea High Availability Status

When viewing the details of the Kea application for which High Availability (HA) is enabled (via the `libdhcp_ha.so` hooks library), the High Availability live status is presented and periodically refreshed for the DHCPv4 and/or DHCPv6 daemon configured as primary or secondary/standby server. The status is not displayed for the server configured as an HA backup. See the High Availability section in the Kea ARM for details about the roles of the servers within the HA setup.

The following picture shows a typical High Availability status view displayed in the Stork UI.



The **local** server is the DHCP server (daemon) belonging to the application for which the status is displayed; the **remote** server is its active HA partner. The remote server belongs to a different application running on a different machine, and this machine may or may not be monitored by Stork. The statuses of both the local and the remote servers are fetched by sending the status-get command to the Kea server whose details are displayed (the local server). In the load-balancing and hot-standby modes, the local server periodically checks the status of its partner by sending it the `ha-heartbeat` command. Therefore, this information is not always up-to-date; its age depends on the heartbeat command interval (typically 10 seconds). The status of the remote server includes the age of the data displayed.

The status information contains the role, state, and scopes served by each HA partner. In the usual HA case, both servers are in load-balancing state, which means that both are serving DHCP clients and there is no failure. If the remote server crashes, the local server transitions to the partner-down state, which will be reflected in this view. If the local server crashes, this will manifest itself as a communication problem between Stork and the server.

As of the Stork 0.8.0 release, the High Availability view may also contain the information about the heartbeat status between the two servers and the information about the failover progress. This information is only available while monitoring Kea version 1.7.8 and later.

The failover progress information is only presented when one of the active servers has been unable to communicate with the partner via the heartbeat exchange for a time exceeding the `max-heartbeat-delay` threshold. If the server is configured to monitor the DHCP traffic directed to the partner, to verify that the partner is not responding

to this traffic before transitioning to the partner-down state, the number of unacked clients (clients which failed to get the lease), connecting clients (all clients currently trying to get the lease from the partner), and analyzed packets are displayed. The system administrator may use this information to diagnose why the failover transition has not taken place or when such a transition is likely to happen.

More about High Availability status information provided by Kea can be found in the Kea ARM.

3.5.10 Viewing the Kea Log

Stork offers a simple log-viewing mechanism to diagnose issues with monitored applications.

Note: As of the Stork 0.10 release, this mechanism only supports viewing Kea log files; viewing BIND 9 logs is not yet supported. Monitoring other logging locations such as: stdout, stderr or syslog is also not supported.

Kea can be configured to log into multiple destinations. Different types of log messages may be output into different log files, syslog, stdout, or stderr. The list of log destinations used by the Kea application is available on the `Kea App` page. Click on the Kea app to view its logs. Next, select the Kea daemon by clicking on one of the tabs, e.g. the `DHCPv4` tab. Scroll down to the `Loggers` section.

This section contains a table with a list of configured loggers for the selected daemon. For each configured logger, the logger's name, logging severity, and output location are presented. The possible output locations are: log file, stdout, stderr, or syslog. It is only possible to view the logs output to the log files. Therefore, for each log file there is a link which leads to the log viewer showing the selected file's contents. The loggers which output to the stdout, stderr, and syslog are also listed, but links to the log viewer are not available for them.

Clicking on the selected log file navigates to its log viewer. By default, the viewer displays the tail of the log file, up to 4000 characters. Depending on the network latency and the size of the log file, it may take several seconds or more before the log contents are fetched and displayed.

The log viewer title bar comprises three buttons. The button with the refresh icon triggers log data fetch without modifying the size of the presented data. Clicking on the + button extends the size of the viewed log tail by 4000 characters and refreshes the data in the log viewer. Conversely, clicking on the − button reduces the amount of presented data by 4000 characters. Every time any of these buttons is clicked, the viewer discards the currently presented data and displays the latest part of the log file tail.

Please keep in mind that extending the size of the viewed log tail may cause slowness of the log viewer and network congestion as the amount of data fetched from the monitored machine increases.

3.6 Dashboard

The main Stork page presents a dashboard. It contains a panel with information about DHCP and a panel with events observed or noticed by the Stork server.

3.6.1 DHCP Panel

The DHCP panel includes two sections: one for DHCPv4 and one for DHCPv6. Each section contains three kinds of information:

- a list of up to five subnets with the highest pool utilization
- a list of up to five shared networks with the highest pool utilization
- statistics about DHCP

3.6.2 Events Panel

The Events panel presents the list of the most recent events captured by the Stork server. There are three event urgency levels: info, warning and error. Events pertaining to the particular entities, e.g. machines or applications, provide a link to a web page containing the information about the given object.

3.7 Events Page

The Events page presents a list of all events. It allows events to be filtered by:

- urgency level
- machine
- application type (Kea, BIND 9)
- daemon type (DHCPv4, DHCPv6, named, etc.)
- the user who caused given event (available only to users in the `super-admin` group)

CHAPTER 4

Backend API

The Stork agent provides a REST API, generated using Swagger. Source YAML files are stored in the *api/* directory in the source files. To view the REST API documentation, open the Stork interface, click Help and choose `Stork API Docs (SwaggerUI)` or `Stork API Docs (Redoc)`.

CHAPTER 5

Developer's Guide

Note: ISC acknowledges that users and developers have different needs, so the user and developer documents should eventually be separated. However, since the project is still in its early stages, this section is kept in the Stork ARM for convenience.

5.1 Rakefile

Rakefile is a script for performing many development tasks, like building source code, running linters, running unit tests, and running Stork services directly or in Docker containers.

There are several other Rake targets. For a complete list of available tasks, use *rake -T*. Also see the Stork wiki for detailed instructions.

5.2 Generating Documentation

To generate documentation, simply type `rake doc`. Sphinx and rtd-theme must be installed. The generated documentation will be available in the `doc/singlehtml` directory.

5.3 Setting Up the Development Environment

The following steps install Stork and its dependencies natively, i.e., on the host machine, rather than using Docker images.

First, PostgreSQL must be installed. This is OS-specific, so please follow the instructions from the *Installation* chapter.

Once the database environment is set up, the next step is to build all the tools. Note that the first command below downloads some missing dependencies and installs them in a local directory. This is done only once and is not needed for future rebuilds, although it is safe to rerun the command.

```
$ rake build_backend
$ rake build_ui
```

The environment should be ready to run. Open three consoles and run the following three commands, one in each console:

```
$ rake run_server
```

```
$ rake serve_ui
```

```
$ rake run_agent
```

Once all three processes are running, connect to http://localhost:8080 via a web browser. See *Using Stork* for information on initial password creation or addition of new machines to the server.

The *run_agent* runs the agent directly on the current operating system, natively; the exposed port of the agent is 8888.

There are other Rake tasks for running preconfigured agents in Docker containers. They are exposed to the host on specific ports.

When these agents are added as machines in the `Stork Server` UI, both a localhost address and a port specific to a given container must be specified. The list of containers can be found in the *Docker Containers for Development* section.

5.3.1 Installing Git Hooks

There is a simple git hook that inserts the issue number in the commit message automatically; to use it, go to the `utils` directory and run the `git-hooks-install` script. It copies the necessary file to the `.git/hooks` directory.

5.4 Agent API

The connection between the Stork server and the agents is established using gRPC over http/2. The agent API definition is kept in the `backend/api/agent.proto` file. For debugging purposes, it is possible to connect to the agent using the grpcurl tool. For example, a list of currently provided gRPC calls may be retrieved with this command:

```
$ grpcurl -plaintext -proto backend/api/agent.proto localhost:8888 describe
agentapi.Agent is a service:
service Agent {
  rpc detectServices ( .agentapi.DetectServicesReq ) returns ( .agentapi.
↪DetectServicesRsp );
  rpc getState ( .agentapi.GetStateReq ) returns ( .agentapi.GetStateRsp );
  rpc restartKea ( .agentapi.RestartKeaReq ) returns ( .agentapi.RestartKeaRsp );
}
```

Specific gRPC calls can also be made. For example, to get the machine state, use the following command:

```
$ grpcurl -plaintext -proto backend/api/agent.proto localhost:8888 agentapi.Agent.
↪getState
{
  "agentVersion": "0.1.0",
  "hostname": "copernicus",
  "cpus": "8",
  "cpusLoad": "1.68 1.46 1.28",
```

(continues on next page)

(continued from previous page)

```
  "memory": "16",
  "usedMemory": "59",
  "uptime": "2",
  "os": "darwin",
  "platform": "darwin",
  "platformFamily": "Standalone Workstation",
  "platformVersion": "10.14.6",
  "kernelVersion": "18.7.0",
  "kernelArch": "x86_64",
  "hostID": "c41337a1-0ec3-3896-a954-a1f85e849d53"
}
```

5.5 REST API

The primary user of the REST API is the Stork UI in a web browser. The definition of the REST API is located in the `api` folder and is described in Swagger 2.0 format.

The description in Swagger is split into multiple files. Two files comprise a tag group:

- *-paths.yaml - defines URLs
- *-defs.yaml - contains entity definitions

All these files are combined by the `yamlinc` tool into a single Swagger file, `swagger.yaml`. Then `swagger.yaml` generates code for:

- the UI fronted by swagger-codegen
- the backend in Go lang by go-swagger

All these steps are accomplished by Rakefile.

5.6 Backend Unit Tests

There are unit tests for the Stork agent and server backends, written in Go. They can be run using Rake:

```
$ rake unittest_backend
```

This requires preparing a database in PostgreSQL. One way to avoid doing this manually is by using a Docker container with PostgreSQL, which is automatically created when running the following Rake task:

```
$ rake unittest_backend_db
```

This task spawns a container with PostgreSQL in the background which then runs unit tests. When the tests are completed, the database is shut down and removed.

5.6.1 Unit Tests Database

When a Docker container with a database is not used for unit tests, the PostgreSQL server must be started and the following role must be created:

```
postgres=# CREATE USER storktest WITH PASSWORD 'storktest';
CREATE ROLE
postgres=# ALTER ROLE storktest SUPERUSER;
ALTER ROLE
```

To point unit tests to a specific Stork database, set the `POSTGRES_ADDR` environment variable, e.g.:

```
$ rake unittest_backend POSTGRES_ADDR=host:port
```

By default it points to `localhost:5432`.

Similarly, if the database setup requires a password other than the default `storktest`, the PGPASSWORD variable can be used by issuing the following command:

```
$ rake unittest_backend PGPASSWORD=secret123
```

Note that there is no need to create the `storktest` database itself; it is created and destroyed by the Rakefile task.

5.6.2 Unit Tests Coverage

A coverage report is presented once the tests have executed. If coverage of any module is below a threshold of 35%, an error is raised.

5.6.3 Benchmarks

Benchmarks are part of the backend unit tests. They are implemented using the golang "testing" library and they test performance-sensitive parts of the backend. Unlike unit tests, the benchmarks do not return pass/fail status. They measure average execution time of functions and print the results to the console.

In order to run unit tests with benchmarks, the `benchmark` environment variable must be specified as follows:

```
$ rake unittest_backend benchmark=true
```

This command runs all unit tests and all benchmarks. Running benchmarks without unit tests is possible using the combination of the `benchmark` and `test` environment variables:

```
$ rake unittest_backend benchmark=true test=Bench
```

Benchmarks are useful to test the performance of complex functions and find bottlenecks. When working on improving the performance of a function, examining a benchmark result before and after the changes is a good practice to ensure that the goals of the changes are achieved.

Similarly, adding a new logic to a function often causes performance degradation, and careful examination of the benchmark result drop for that function may be a driver for improving efficiency of the new code.

5.6.4 Short Testing Mode

It is possible to filter out long running unit tests. Set the `short` variable to *true* on the command line:

```
$ rake unittest_backend short=true
```

5.7 Web UI Unit Tests

Stork offers web UI tests, to take advantage of the unit-tests generated automatically by Angular. The simplest way to run these tests is by using Rake tasks:

```
rake build_ui
rake ng_test
```

The tests require the Chromium (on Linux) or Chrome (on Mac) browser. The *rake ng_test* task attempts to locate the browser binary and launch it automatically. If the browser binary is not found in the default location, the Rake task returns an error. It is possible to set the location manually by setting the *CHROME_BIN* environment variable; for example:

```
export CHROME_BIN=/usr/local/bin/chromium-browser
rake ng_test
```

By default, the tests launch the browser in headless mode, in which test results and any possible errors are printed in the console. However, in some situations it is useful to run the browser in non-headless mode because it provides debugging features in Chrome's graphical interface. It also allows for selectively running the tests. Run the tests in non-headless mode using the *debug* variable appended to the *rake* command:

```
rake ng_test debug=true
```

That command causes a new browser window to open; the tests run there automatically.

The tests are run in random order by default, which can make it difficult to chase the individual errors. To make debugging easier by always running the tests in the same order, click Debug in the new Chrome window, then click Options and unset the "run tests in random order" button. A specific test can be run by clicking on its name.

When adding a new component or service with *ng generate component|service ...*, the Angular framework adds a .spec.ts file with boilerplate code. In most cases, the first step in running those tests is to add the necessary Stork imports. If in doubt, refer to the commits on https://gitlab.isc.org/isc-projects/stork/-/merge_requests/97. There are many examples of ways to fix failing tests.

5.8 System Tests

System tests for Stork are designed to test the software in a distributed environment. They allow for testing several Stork servers and agents running at the same time in one test case, inside `LXD` containers. It is possible to set up Kea (and eventually, BIND 9) services along with Stork agents. The framework enables experimenting in containers so custom Kea configurations can be deployed or specific Kea daemons can be stopped.

The tests can use the Stork server REST API directly or the Stork web UI via Selenium.

5.8.1 Dependencies

System tests require:

- Linux operating system (preferably Ubuntu or Fedora)
- Python 3
- `LXD` containers (https://linuxcontainers.org/lxd/introduction)

5.8.2 LXD Installation

The easiest way to install LXD is to use snap. First, install snap.

On Fedora:

```
$ sudo dnf install snapd
```

On Ubuntu:

```
$ sudo apt install snapd
```

Then install LXD:

```
$ sudo snap install lxd
```

And then add the user to lxd group:

```
$ sudo usermod -a -G lxd $USER
```

Now log in again to make the user's presence in lxd group visible in the shell session.

After installing LXD, it requires initialization. Run:

```
$ lxd init
```

and then for each question press **Enter**, i.e., use the default values:

```
Would you like to use LXD clustering? (yes/no) [default=no]: **Enter**
Do you want to configure a new storage pool? (yes/no) [default=yes]: **Enter**
Name of the new storage pool [default=default]: **Enter**
Name of the storage backend to use (dir, btrfs) [default=btrfs]: **Enter**
Would you like to create a new btrfs subvolume under /var/snap/lxd/common/lxd? (yes/
↪no) [default=yes]: **Enter**
Would you like to connect to a MAAS server? (yes/no) [default=no]:  **Enter**
Would you like to create a new local network bridge? (yes/no) [default=yes]: ␣
↪**Enter**
What should the new bridge be called? [default=lxdbr0]:  **Enter**
What IPv4 address should be used? (CIDR subnet notation, "auto" or "none")␣
↪[default=auto]:  **Enter**
What IPv6 address should be used? (CIDR subnet notation, "auto" or "none")␣
↪[default=auto]:  **Enter**
Would you like LXD to be available over the network? (yes/no) [default=no]:  **Enter**
Would you like stale cached images to be updated automatically? (yes/no)␣
↪[default=yes]  **Enter**
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]: ␣
↪**Enter**
```

More details can be found at: https://linuxcontainers.org/lxd/getting-started-cli/

The subvolume is stored in /var/snap/lxd/common/lxd, and is used to store images and containers. If the space is exhausted, it is not possible to create new containers. This is not connected with total disk space but rather with the space in this subvolume. To free space, remove stale images or stopped containers. Basic usage of LXD is presented at: https://linuxcontainers.org/lxd/getting-started-cli/#lxd-client

5.8.3 Running System Tests

After preparing all the dependencies, it is possible to start tests. But first, the RPM and deb Stork packages need to be prepared. This can be done with this Rake task:

```
$ rake build_pkgs_in_docker
```

When using packages, the tests can be invoked by the following Rake task:

```
$ rake system_tests
```

This command first prepares the Python virtual environment (`venv`) where `pytest` and other Python dependencies are installed. `pytest` is a Python testing framework that is used in Stork system tests.

At the end of the logs are listed test cases with their result status.

The tests can be invoked directly using `pytest`, but first the directory must be changed to `tests/system`:

```
$ cd tests/system
$ ./venv/bin/pytest --tb=long -l -r ap -s tests.py
```

The switches passed to `pytest` are:

- `--tb=long`: in case of failures, present long format of traceback
- `-l`: show values of local variables in tracebacks
- `-r ap`: at the end of execution, print a report that includes (p)assed and (a)ll except passed (p)

To run a particular test case, add it just after `test.py`:

```
$ ./venv/bin/pytest --tb=long -l -r ap -s tests.py::test_users_management[centos/7-
↪ubuntu/18.04]
```

To get a list of tests without actually running them, the following command can be used:

```
$ ./venv/bin/pytest --collect-only tests.py
```

The names of all available tests are printed as <*Function name_of_the_test*>.

A single test case can be run using a `rake` task with the test variable set to the test name:

```
$ rake system_tests test=tests.py::test_users_management[centos/7-ubuntu/18.04]
```

5.8.4 Developing System Tests

System tests are defined in tests.py and other files that start with *test_*. There are two other files that define the framework for Stork system tests:

- conftest.py - defines hooks for `pytests`
- containers.py - handles LXD containers: starting/stopping; communication, such as invoking commands; uploading/downloading files; installing and preparing Stork Agent/Server and Kea; and other dependencies that they require.

Most tests are constructed as follows:

```
@pytest.mark.parametrize("agent, server", SUPPORTED_DISTROS)
def test_machines(agent, server):
    # login to stork server
    r = server.api_post('/sessions',
                        json=dict(useremail='admin', userpassword='admin'),
                        expected_status=200)
    assert r.json()['login'] == 'admin'

    # add machine
    machine = dict(
        address=agent.mgmt_ip,
        agentPort=8080)
    r = server.api_post('/machines', json=machine, expected_status=200)
    assert r.json()['address'] == agent.mgmt_ip

    # wait for application discovery by Stork Agent
    for i in range(20):
        r = server.api_get('/machines')
        data = r.json()
        if len(data['items']) == 1 and \
           len(data['items'][0]['apps'][0]['details']['daemons']) > 1:
            break
        time.sleep(2)

    # check discovered application by Stork Agent
    m = data['items'][0]
    assert m['apps'][0]['version'] == '1.7.3'
```

It may be useful to explain each part of this code.

```
@pytest.mark.parametrize("agent, server", SUPPORTED_DISTROS)
```

This indicates that the test is parameterized: there will be one or more instances of this test in execution for each set of parameters.

The constant `SUPPORTED_DISTROS` defines two sets of operating systems for testing:

```
SUPPORTED_DISTROS = [
    ('ubuntu/18.04', 'centos/7'),
    ('centos/7', 'ubuntu/18.04')
]
```

The first set indicates that for the Stork agent `Ubuntu 18.04` should be used in the LXD container, and for the Stork server `CentOS 7`. The second set is the opposite of the first one.

The next line:

```
def test_machines(agent, server):
```

defines the test function. Normally, the agent and server argument would get the text values `'ubuntu/18.04'` and `'centos/7'`, but a hook exists in the `pytest_pyfunc_call()` function of `conftest.py` that intercepts these arguments and uses them to spin up LXD containers with the indicated operating systems. This hook also collects Stork logs from these containers at the end of the test and stores them in the `test-results` folder for later analysis if needed.

Instead of text values, the hook replaces the arguments with references to actual LXC container objects, so that the test can interact directly with them. Besides substituting the `agent` and `server` arguments, the hook intercepts any

argument that starts with `agent` or `server`. This allows multiple agents in the test, e.g. `agent1`, `agent_kea`, or `agent_bind9`.

Next, log into the Stork server using its REST API:

```
# login to stork server
r = server.api_post('/sessions',
                    json=dict(useremail='admin', userpassword='admin'),
                    expected_status=200)
assert r.json()['login'] == 'admin'
```

Then, add a machine with a Stork agent to the Stork server:

```
# add machine
machine = dict(
    address=agent.mgmt_ip,
    agentPort=8080)
r = server.api_post('/machines', json=machine, expected_status=200)
assert r.json()['address'] == agent.mgmt_ip
```

A check then verifies the returned address of the machine.

After a few seconds, the Stork agent detects the Kea application and reports it to the Stork server. The server is periodically polled for updated information about the Kea application.

```
# wait for application discovery by Stork Agent
for i in range(20):
    r = server.api_get('/machines')
    data = r.json()
    if len(data['items']) == 1 and \
       len(data['items'][0]['apps'][0]['details']['daemons']) > 1:
        break
    time.sleep(2)
```

Finally, the returned data about Kea can be verified:

```
# check discovered application by Stork Agent
m = data['items'][0]
assert m['apps'][0]['version'] == '1.7.3'
```

5.9 Docker Containers for Development

To ease development, there are several Docker containers available. These containers are used in the Stork demo and are fully described in the *Demo* chapter.

The following `Rake` tasks start these containers.

Table 1: Rake tasks for managing development containers.

Rake Task	Description
`rake build_kea_container`	Build a container *agent-kea* with a Stork agent and Kea with DHCPv4.
`rake run_kea_container`	Start an *agent-kea* container. Published port is 8888.

Continued on next page

Table 1 – continued from previous page

Rake Task	Description
`rake build_kea6_container`	Build an *agent-kea6* container with a Stork agent and Kea with DHCPv6.
`rake run_kea6_container`	Start an *agent-kea6* container. Published port is 8886.
`rake build_kea_ha_containers`	Build two containers, *agent-kea-ha1* and *agent-kea-ha2*, that are configured to work together in *High Availability* mode, with Stork agents, and Kea with DHCPv4.
`rake run_kea_ha_containers`	Start the *agent-kea-ha1* and *agent-kea-ha2* containers. Published ports are 8881 and 8882.
`rake build_kea_hosts_container`	Build an *agent-kea-hosts* container with a Stork agent and Kea with DHCPv4 with host reservations stored in a database. This requires **premium** features.
`rake run_kea_hosts_container`	Start the *agent-kea-hosts* container. This requires **premium** features.
`rake build_bind9_container`	Build an *agent-bind9* container with a Stork agent and BIND 9.
`rake run_bind9_container`	Start an *agent-bind9* container. Published port is 9999.

5.10 Packaging

There are scripts for packaging the binary form of Stork. There are two supported formats: RPM and deb.

The RPM package is built on the latest CentOS version. The deb package is built on the latest Ubuntu LTS.

There are two packages built for each system: a server and an agent.

Rake tasks can perform the entire build procedure in a Docker container: *build_rpms_in_docker* and *build_debs_in_docker*. It is also possible to build packages directly in the current operating system; this is provided by the *deb_agent*, *rpm_agent*, *deb_server*, and *rpm_server* Rake tasks.

Internally, these packages are built by FPM (https://fpm.readthedocs.io/). The containers that are used to build packages are prebuilt with all dependencies required, using the *build_fpm_containers* Rake task. The definitions of these containers are placed in *docker/pkgs/centos-8.txt* and *docker/pkgs/ubuntu-18-04.txt*.

CHAPTER 6

Demo

A sample installation of `Stork` can be used to demonstrate `Stork` capabilities, and can also be used for its development.

The demo installation uses *Docker* and *Docker Compose* to set up all *Stork* services. It contains:

- Stork Server
- Stork Agent with Kea DHCPv4
- Stork Agent with Kea DHCPv6
- Stork Agent with Kea HA-1 (high availability server 1)
- Stork Agent with Kea HA-2 (high availability server 2)
- Stork Agent with BIND 9
- Stork Environment Simulator
- PostgreSQL database
- Prometheus & Grafana

These services allow observation of many Stork features.

6.1 Requirements

Running the `Stork Demo` requires the same dependencies as building Stork, which are described in the *Installing From Sources* chapter.

Besides the standard dependencies, the `Stork Demo` requires:

- Docker
- Docker Compose

For details, please see the Stork wiki at https://gitlab.isc.org/isc-projects/stork/-/wikis/Processes/development-Environment

6.2 Setup Steps

The following command retrieves all required software (go, goswagger, nodejs, Angular dependencies, etc.) to the local directory. No root password is necessary. It then prepares Docker images and starts them.

```
$ rake docker_up
```

Once the build process finishes, the Stork UI is available at http://localhost:8080/. Use any browser to connect.

6.2.1 Premium Features

It is possible to run the demo with premium features enabled in Kea apps. It requires starting the demo with an access token to the Kea premium repositories. Access tokens are provided to ISC's paying customers and can be found on https://cloudsmith.io/~isc/repos/kea-1-7-prv/setup/#tab-formats-deb. The token can be found inside this URL on that page: `https://dl.cloudsmith.io/${ACCESS_TOKEN}/isc/kea-1-7-prv/cfg/setup/bash.deb.sh`. This web page and the token are available only to paying customers of ISC.

```
$ rake docker_up cs_repo_access_token=<access token>
```

6.3 Demo Containers

The setup procedure creates several Docker containers. Their definition is stored in the `docker-compose.yaml` file in the Stork source code repository.

These containers have Stork production services and components:

server This container is essential. It runs the Stork server, which interacts with all the agents and the database and exposes the API. Without it, Stork is not able to function.

webui This container is essential in most circumstances. It provides the front-end web interface. It is potentially unnecessary with the custom development of a Stork API client.

agent-bind9 This container runs a BIND 9 server. With this container, the agent can be added as a machine and Stork will begin monitoring its BIND 9 service.

agent-bind9-2 This container also runs a BIND 9 server, for the purpose of experimenting with two different DNS servers.

agent-kea This container runs a Kea DHCPv4 server. With this container, the agent can be added as a machine and Stork will begin monitoring its Kea DHCPv4 service.

agent-kea6 This container runs a Kea DHCPv6 server.

agent-kea-ha1 and agent-kea-ha2 These two containers should, in general, be run together. They each have a Kea DHCPv4 server instance configured in an HA pair. With both running and registered as machines in Stork, users can observe certain HA mechanisms, such as one partner taking over the traffic if the other partner becomes unavailable.

agent-kea-many-subnets This container runs an agent with a Kea DHCPv4 server that has many subnets defined in its configuration (about 7000).

These are containers with 3rd-party services that are required by Stork:

postgres This container is essential. It runs the PostgreSQL database that is used by the Stork server. Without it, the Stork server produces error messages about an unavailable database.

prometheus Prometheus, a monitoring solution (https://prometheus.io/), uses this container to monitor applications. It is preconfigured to monitor Kea and BIND 9 containers.

grafana This is a container with Grafana (https://grafana.com/), a dashboard for Prometheus. It is preconfigured to pull data from a Prometheus container and show Stork dashboards.

There is also a supporting container:

simulator Stork Environment Simulator is a web application that can run DHCP traffic using `perfdhcp` (useful to observe non-zero statistics coming from Kea), run DNS traffic using `dig` and `flamethrower` (useful to observe non-zero statistics coming from BIND 9), and start and stop any service in any other container (useful to simulate, for example, a Kea crash).

Note: The containers running the Kea and BIND 9 applications are for demonstration purposes only. They allow users to quickly start experimenting with Stork without having to manually deploy Kea and/or BIND 9 instances.

The PostgreSQL database schema is automatically migrated to the latest version required by the Stork server process.

The setup procedure assumes those images are fully under Stork's control. Any existing images are overwritten.

6.4 Initialization

`Stork Server` requires some initial information:

1. Go to http://localhost:8080/machines/all
2. Add new machines (leave the default port):
    1. agent-kea
    2. agent-kea6
    3. agent-kea-ha1
    4. agent-kea-ha2
    5. agent-bind9
    6. agent-bind9-2

6.5 Stork Environment Simulator

Stork Environment Simulator allows:

- sending DHCP traffic to Kea applications
- sending DNS requests to BIND 9 applications
- stopping and starting Stork Agents, and the Kea and BIND 9 daemons

Stork Environment Simulator allows DHCP traffic to be sent to selected subnets pre-configured in Kea instances, with a limitation: it is possible to send traffic to only one subnet from a given shared network.

Stork Environment Simulator also allows sending DNS traffic to selected DNS servers.

Stork Environment Simulator can add all the machines available in the demo setup. It can stop and start selected Stork Agents, and the Kea and BIND 9 applications. This is useful to simulate communication problems between applications, Stork Agents, and the Stork Server.

The Stork Environment Simulator can be found at: http://localhost:5000/ .

For development purposes, the simulator can be started directly with the command:

```
$ rake run_sim
```

6.6 Prometheus

The Prometheus instance is preconfigured and pulls statistics from:

- node exporters: agent-kea:9100, agent-bind9:9100, agent-bind9:9100
- Kea exporters embedded in stork-agent: agent-kea:9547, agent-kea6:9547, agent-kea-ha1:9547, agent-kea-ha2:9547
- BIND exporters embedded in stork-agent: agent-bind9:9119, agent-bind9-2:9119

The Prometheus web page can be found at: http://localhost:9090/ .

6.7 Grafana

The Grafana instance is also preconfigured. It pulls data from Prometheus and loads dashboards from the Stork repository, in the Grafana folder.

The Grafana web page can be found at: http://localhost:3000/ .

CHAPTER 7

Manual Pages

7.1 stork-server - Main Stork server

7.1.1 Synopsis

stork-server

7.1.2 Description

`stork-server` provides the main Stork server capabilities. In every Stork deployment, there should be exactly one Stork server.

7.1.3 Arguments

`stork-server` takes the following arguments:

-h or --help the list of available parameters.

-v or --version the `stork-server` version.

-d or --db-name= the name of the database to connect to. (default: stork) [$STORK_DATABASE_NAME]

-u or --db-user the user name to be used for database connections. (default: stork) [$STORK_DATABASE_USER_NAME]

--db-host the name of the host where the database is available. (default: localhost) [$STORK_DATABASE_HOST]

-p or --db-port the port on which the database is available. (default: 5432) [$STORK_DATABASE_PORT]

--db-trace-queries= enable tracing SQL queries: run (only runtime, without migrations), all (migrations and run-time)). `all` is the default and covers both migrations and `run-time.enable` tracing SQL queries. [$STORK_DATABASE_TRACE]

--rest-cleanup-timeout the period to wait before killing idle connections. (default: 10s)

--rest-graceful-timeout the period to wait before shutting down the server. (default: 15s)

--rest-max-header-size the maximum number of bytes the server reads parsing the request header's keys and values, including the request line. It does not limit the size of the request body. (default: 1MiB)

--rest-host the IP to listen on for connections over the REST API. [$STORK_REST_HOST]

--rest-port the port to listen on for connections over the REST API. (default: 8080) [$STORK_REST_PORT]

--rest-listen-limit the maximum number of outstanding requests.

--rest-keep-alive the TCP keep-alive timeout on accepted connections. It prunes dead TCP connections (e.g. closing laptop mid-download). (default: 3m)

--rest-read-timeout the maximum duration before timing out a read of the request. (default: 30s)

--rest-write-timeout the maximum duration before timing out a write of the response. (default: 60s)

--rest-tls-certificate the certificate to use for secure connections. [$STORK_REST_TLS_CERTIFICATE]

--rest-tls-key the private key to use for secure connections. [$STORK_REST_TLS_PRIVATE_KEY]

--rest-tls-ca the certificate authority file to be used with a mutual TLS authority. [$STORK_REST_TLS_CA_CERTIFICATE]

--rest-static-files-dir the directory with static files for the UI. [$STORK_REST_STATIC_FILES_DIR]

Note that there is no argument for database password, as the command-line arguments can sometimes be seen by other users. It can be passed using the STORK_DATABASE_PASSWORD variable.

7.1.4 Mailing Lists and Support

There are public mailing lists available for the Stork project. **stork-users** (stork-users at lists.isc.org) is intended for Stork users. **stork-dev** (stork-dev at lists.isc.org) is intended for Stork developers, prospective contributors, and other advanced users. The lists are available at https://lists.isc.org. The community provides best-effort support on both of those lists.

Once stork becomes more mature, ISC will provide professional support for Stork services.

7.1.5 History

`stork-server` was first coded in November 2019 by Michal Nowikowski and Marcin Siodelski.

7.1.6 See Also

`stork-agent(8)`

7.2 stork-agent - Stork agent that monitors BIND 9 and Kea services

7.2.1 Synopsis

stork-agent [**–host**] [**–port**]

7.2.2 Description

The `stork-agent` is a small tool that operates on systems that are running BIND 9 and Kea services. The Stork server connects to the Stork agent and uses it to monitor services remotely.

7.2.3 Arguments

Stork does not use an explicit configuration file. Instead, its behavior can be controlled with command-line switches and/or variables. The Stork agent takes the following command-line switches. Equivalent environment variables are listed in square brackets, where applicable.

`--listen-stork-only` listen for commands from the Stork server only, but not for Prometheus requests. [$STORK_AGENT_LISTEN_STORK_ONLY]

`--listen-prometheus-only` listen for Prometheus requests only, but not for commands from the Stork server. [$STORK_AGENT_LISTEN_PROMETHEUS_ONLY]

`-v` or `--version` show software version.

`Stork Server` flags:

`--host=` the IP or hostname to listen on for incoming Stork server connections. [$STORK_AGENT_ADDRESS]

`--port=` the TCP port to listen on for incoming Stork server connections. (default: 8080) [$STORK_AGENT_PORT]

`Prometheus Kea Exporter` flags:

`--prometheus-kea-exporter-host=` the IP or hostname to listen on for incoming Prometheus connections. (default: 0.0.0.0) [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_ADDRESS]

`--prometheus-kea-exporter-port=` the port to listen on for incoming Prometheus connections. (default: 9547) [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_PORT]

`--prometheus-kea-exporter-interval=` how often the agent collects stats from Kea, in seconds. (default: 10) [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_INTERVAL]

`Prometheus BIND 9 Exporter` flags:

`--prometheus-bind9-exporter-host=` the IP or hostname to listen on for incoming Prometheus connections. (default: 0.0.0.0) [$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_ADDRESS]

`--prometheus-bind9-exporter-port=` the port to listen on for incoming Prometheus connections. (default: 9119) [$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_PORT]

`--prometheus-bind9-exporter-interval=` how often the agent collects stats from BIND 9, in seconds. (default: 10) [$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_INTERVAL]

`-h` or `--help` the list of available parameters.

7.2.4 Mailing Lists and Support

There are public mailing lists available for the Stork project. **stork-users** (stork-users at lists.isc.org) is intended for Stork users. **stork-dev** (stork-dev at lists.isc.org) is intended for Stork developers, prospective contributors, and other advanced users. The lists are available at https://lists.isc.org. The community provides best-effort support on both of those lists.

Once Stork becomes more mature, ISC will provide professional support for Stork services.

7.2.5 History

The `stork-agent` was first coded in November 2019 by Michal Nowikowski.

7.2.6 See Also

stork-server(8)

7.3 stork-db-migrate - Stork database migration tool

7.3.1 Synopsis

stork-db-migrate [options] command

7.3.2 Description

The `stork-db-migrate` tool is an option to assist with database schema migrations. Usually, there is no need to use this tool, as the Stork server always runs the migration scripts on startup. However, it may be useful for debugging and manual migrations.

7.3.3 Arguments

`stork-db-migrate` takes the following commands:

Available commands:

`down` Revert last migration (or use -t to migrate to a specific version)

`init` Create schema versioning table in the database

`reset` Revert all migrations

`set_version` Set database version without running migrations

`up` Run all available migrations (or use -t to migrate to a specific version)

`version` Print current migration version

Application Options:

-d, --db-name= the name of the database to connect to. (default: stork) [$STORK_DATABASE_NAME]

-u, --db-user= the user name to be used for database connections. (default: stork) [$STORK_DATABASE_USER_NAME]

--db-host= the name of the host where the database is available. (default: localhost) [$STORK_DATABASE_HOST]

-p, --db-port= the port on which the database is available. (default: 5432) [$STORK_DATABASE_PORT]

--db-trace-queries= enable tracing SQL queries: run (only runtime, without migrations), all (migrations and run-time)). `all` is the default and covers both migrations and `run-time.enable` tracing SQL queries. [$STORK_DATABASE_TRACE]

-h, --help show help message

Note that there is no argument for the database password, as the command-line arguments can sometimes be seen by other users. It can be passed using the STORK_DATABASE_PASSWORD variable.

7.3.4 Mailing Lists and Support

There are public mailing lists available for the Stork project. **stork-users** (stork-users at lists.isc.org) is intended for Stork users. **stork-dev** (stork-dev at lists.isc.org) is intended for Stork developers, prospective contributors, and other advanced users. The lists are available at https://lists.isc.org. The community provides best-effort support on both of those lists.

Once stork becomes more mature, ISC will provide professional support for Stork services.

7.3.5 History

The `stork-db-migrate` tool was first coded in October 2019 by Marcin Siodelski.

7.3.6 See Also

stork-agent(8), *stork-server(8)*

CHAPTER 8

Indices and tables

- genindex
- modindex
- search