Stork

Release 1.7.0

Internet Systems Consortium

Oct 12, 2022

CONTENTS

Stork is an open source monitoring application and dashboard for ISC's Kea DHCP, and eventually for ISC's BIND 9. It is the spiritual successor of the earlier projects Kittiwake and Anterius.

This is the reference guide for Stork version 1.7.0. 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

ONE

OVERVIEW

1.1 Goals

The goals of the ISC Stork project are:

- To provide monitoring and insight into Kea DHCP operations.
- To provide alerting mechanisms that indicate failures, fault conditions, and other unwanted events in Kea DHCP services.
- To permit easier troubleshooting of these services.

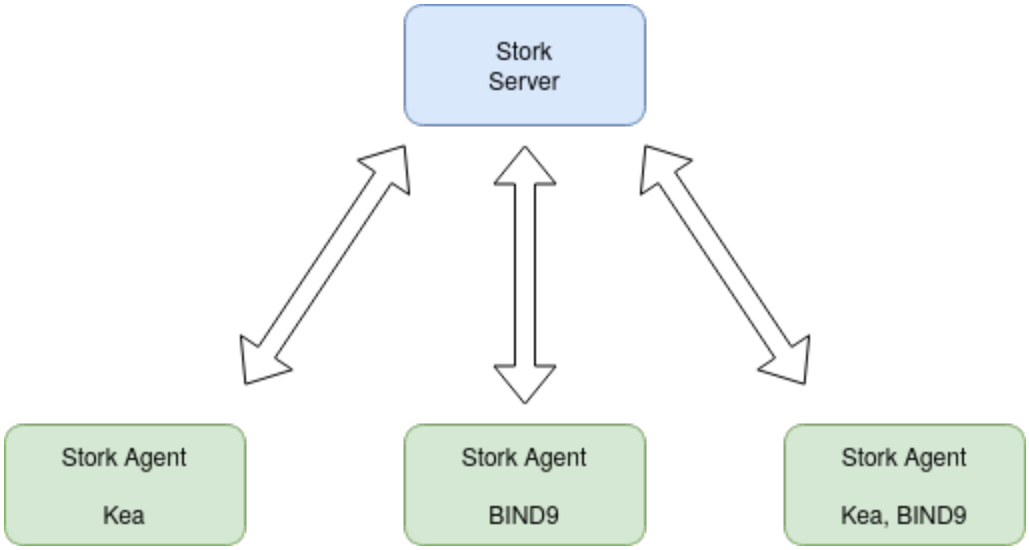
Although Stork currently only offers monitoring, insight, and alerts for Kea DHCP, we plan to add similar capabilities for BIND 9 in future versions.

1.2 Architecture

Stork is comprised of two components: the Stork server (`stork-server`) and the Stork agent (`stork-agent`).

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 services. It provides an integrated, centralized front end for these services. Only one Stork server is deployed in a network.

The Stork agent is installed along with Kea DHCP and interacts directly with those services. There may be many agents deployed in a network, one per machine.


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

CHAPTER

TWO

INSTALLATION

Stork can be installed from pre-built packages or from sources; the following sections describe both methods. Unless there is 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 8
- MacOS 11.3*

* MacOS is not and will not be officially supported. However, many developers on ISC's team use Macs, so the intention is to keep Stork buildable on this platform.

`stork-server` and `stork-agent` 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 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.

Stork requires the premium Host Commands (`host_cmds`) hook library to be loaded by the Kea instance to retrieve host reservations stored in an external database. Stork works without the Host Commands hook library, but is not able to display host reservations. Stork can retrieve host reservations stored locally in the Kea configuration without any additional hook libraries.

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

Stork uses Go implementation to handle 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 10 or later is required. Stork will attempt to run with older versions, but may not work correctly. 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.

2.3 Stork Tool

The `Stork Tool` is a program installed with the `Stork Server`, providing commands to set up server's database and manage TLS certificates. Using this tool is facultative because the server runs the database migrations and creates suitable certificates at startup on its own. However, the tool provides useful commands for inspecting the current database schema version and downgrading to one of the previous versions. In addition, in the *Preparing Stork Server Database* section it is described how the tool can be conveniently used to create a new database and its credentials without a need to run SQL commands directly using the `psql` program.

The *Inspecting Keys and Certificates* section describes how to use the tool for TLS certificates management.

Further sections describe different methods for installing the Stork Server from packages. See: *Installing on Debian/Ubuntu* and *Installing on CentOS/RHEL/Fedora*. The `stork-tool` program is installed from the packages together with the server. Alternatively, the tool can be built from sources:

```
$ rake build:tool
```

Please refer to the manual page for usage details: *stork-tool - A Tool for Managing Stork Server*.

2.4 Preparing Stork Server Database

Before running `Stork Server`, a PostgreSQL database and the user with suitable privileges must be created. Using the `stork-tool` is the most convenient way to set up the database.

The following command creates a new database `stork` and a user `stork` with all privileges in this database. It also installs the `pgcrypto` extension required by the Stork Server.

```
$ stork-tool db-create --db-name stork --db-user stork
created database and user for the server with the following credentials  database_
↪name=stork password=L82B+kJEOyhDoMnZf9qPAGyKjH5Qo/Xb user=stork
```

By default, `stork-tool` connects to the database as user `postgres`, a default admin role in many PostgreSQL installations. If an installation uses a different administrator name, it can be specified with the `--db-maintenance-user` option. For example:

```
$ stork-tool db-create --db-maintenance-user thomson --db-name stork --db-user stork
created database and user for the server with the following credentials  database_
↪name=stork password=L82B+kJEOyhDoMnZf9qPAGyKjH5Qo/Xb user=stork
```

Similarly, a `postgres` database should often exist in a PostgreSQL installation. However, a different maintenance database can be selected with the `--db-maintenance-name` option.

The `stork-tool` generates a random password to the created database. This password needs to be copied into the server environment file or used in the `stork-server` command line to configure the server to use this password while connecting to the database. Use the `--db-password` option with the `db-create` command to create a user with a specified password.

It is also possible to create the database manually (i.e., using the `psql` tool).

First, connect to PostgreSQL using `psql` and the `postgres` administration user. Depending on the system's configuration, it 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 "postgres".
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. Please also consider using the `stork-tool` to generate a random password.

To generate a random password run:

```
$ stork-tool db-password-gen
generated new database password               password=1qWVzmLKy/j40/FVsvjM2ylcFdaFfNxh
```

The newly created database is not ready for use until necessary database migrations are executed. The migrations create tables, indexes, triggers, and functions required by the `Stork Server`. As mentioned above, the server can automatically run the migrations at startup, bringing up the database schema to the latest version. However, if a user wants to run the migrations before starting the server, they can use the `stork-tool`:

```
$ stork-tool db-init
$ stork-tool db-up
```

The `up` and `down` commands have an optional `-t` parameter that specifies the desired schema version. It is useful when debugging database migrations or downgrading to one of the earlier Stork versions.

```
$ # migrate up version 25
$ stork-tool db-up -t 25
$ # migrate down back to version 17
$ stork-tool db-down -t 17
```

The server requires the latest database version to run, always runs the migration on its own, and refuses to start if the migration fails for any reason. The migration tool is mostly useful for debugging problems with migration, or for migrating the database without actually running the service. For the complete manual page, please see *stork-tool - A Tool for Managing Stork Server*.

To debug migrations, another useful feature is SQL tracing using the `--db-trace-queries` parameter. The options are either "all" (trace all SQL operations, including migrations and runtime) or "run" (only trace runtime operations

and skip migrations). If specified without any parameters, "all" is assumed. With it enabled, `stork-tool` 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`.

```
$ stork-tool db-down -t 0
$ stork-tool db-up --db-trace-queries 2> stork-schema.txt
```

2.5 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 potential impact of the `stork-server` on other services running on the same machine.

2.5.1 Installing the Stork Server

2.5.1.1 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
```

2.5.1.2 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
```

2.5.1.3 Installing on Alpine

The first step for Alpine is:

```
$ curl -1sLf 'https://dl.cloudsmith.io/public/isc/stork/cfg/setup/setup.alpine.sh' | sh
```

Next, install the Stork server package:

```
$ apk add --allow-untrusted isc-stork-server
```

Warning: For the time being, using the `--allow-untrusted` flag is the only option. The FPM packaging tool we use to prepare the package doesn't support the signatures for the APK package type.

2.5.1.4 Setup

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

Configure the Stork server settings in `/etc/stork/server.env`. The following settings are required for the database connection (they have a common `STORK_DATABASE_` prefix):

- `STORK_DATABASE_HOST` - the address of a PostgreSQL database; the default is `localhost`
- `STORK_DATABASE_PORT` - the port of a PostgreSQL database; the default is `5432`
- `STORK_DATABASE_NAME` - the name of a database; the default is `stork`
- `STORK_DATABASE_USER_NAME` - the username for connecting to the database; the 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 RESTful API configuration (the `STORK_REST_` prefix):

- `STORK_REST_HOST` - the IP address on which the server listens
- `STORK_REST_PORT` - the port number on which the server listens; the 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 user interface

---

Note: The `STORK_REST_STATIC_FILES_DIR` should be set to `/usr/share/stork/www` for the Stork Server installed from the binary packages. It's the default location for the static content.

Note: The Stork agent must trust the REST TLS certificate presented by Stork server. Otherwise, the registration process fails due to invalid Stork Server certificate verification. We strongly recommend using proper, trusted certificates for security reasons. If you need to use a self-signed certificate (e.g., for deployment in the Docker environment), then

you can add its CA certificate to the system certificates on the Stork agent machine. See Stack Overflow thread and discussion in #859.

The remaining settings pertain to the server's Prometheus `/metrics` endpoint configuration (the `STORK_SERVER_` prefix is for general purposes):

- `STORK_SERVER_ENABLE_METRICS` - enable the Prometheus metrics collector and `/metrics` HTTP endpoint

Warning: The Prometheus `/metrics` endpoint does not require authentication. Therefore, securing this endpoint from external access is highly recommended to prevent unauthorized parties from gathering the server's metrics. One way to restrict endpoint access is by using an appropriate HTTP proxy configuration to allow only local access or access from the Prometheus host. Please consult the NGINX example configuration file shipped with Stork.

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 a web browser at http://localhost:8080. To use a different IP address or port, 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`.

The logging colorization is configured analogously to the *Stork Agent logging colorization*.

2.5.1.5 Securing the Database Connection

The PostgreSQL server can be configured to encrypt communications between the clients and the server. Detailed information on how to enable encryption on the database server, and how to create the suitable certificate and key files, is available in the PostgreSQL documentation.

The Stork server supports secure communications with the database. The following configuration settings in the `server.env` file enable and configure communication encryption with the database server. They correspond with the SSL settings provided by `libpq` - the native PostgreSQL client library written in C:

- `STORK_DATABASE_SSLMODE` - the SSL mode for connecting to the database (i.e., `disable`, `require`, `verify-ca`, or `verify-full`); the default is `disable`
- `STORK_DATABASE_SSLCERT` - the location of the SSL certificate used by the server to connect to the database
- `STORK_DATABASE_SSLKEY` - the location of the SSL key used by the server to connect to the database
- `STORK_DATABASE_SSLROOTCERT` - the location of the root certificate file used to verify the database server's certificate

The default SSL mode setting, `disable`, configures the server to use unencrypted communication with the database. Other settings have the following meanings:

- `require` - use secure communication but do not verify the server's identity unless the root certificate location is specified and that certificate exists. If the root certificate exists, the behavior is the same as in the case of `verify-ca`.
- `verify-ca` - use secure communication and verify the server's identity by checking it against the root certificate stored on the Stork server machine.
- `verify-full` - use secure communication and verify the server's identity against the root certificate. In addition, check that the server hostname matches the name stored in the certificate.

Specifying the SSL certificate and key location is optional. If they are not specified, the Stork server uses the ones from the current user's home directory: `~/.postgresql/postgresql.crt` and `~/.postgresql/postgresql.key`. If they are not present, Stork tries to find suitable keys in common system locations.

Please consult the libpq documentation for similar `libpq` configuration details.

2.5.2 Installing the Stork Agent

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

2.5.2.1 Agent Configuration Settings

The following are the Stork agent configuration settings available in the `/etc/stork/agent.env` file after installing the package. All these settings use the `STORK_AGENT_` prefix to indicate that they configure the Stork agent.

The general settings:

- `STORK_AGENT_HOST` - the IP address of the network interface or DNS name which `stork-agent` should use to receive connections from the server; the default is `0.0.0.0` (i.e. listen on all interfaces)
- `STORK_AGENT_PORT` - the port number the agent should use to receive connections from the server; the default is `8080`
- `STORK_AGENT_LISTEN_STORK_ONLY` - this enables Stork functionality only, i.e. disables Prometheus exporters; the default is `false`
- `STORK_AGENT_LISTEN_PROMETHEUS_ONLY` - this enables the Prometheus exporters only, i.e. disables Stork functionality; the default is `false`
- `STORK_AGENT_SKIP_TLS_CERT_VERIFICATION` - this skips TLS certificate verification when `stork-agent` connects to Kea over TLS and Kea uses self-signed certificates; the 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 connections from Prometheus when fetching Kea statistics; the 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_KEA_EXPORTER_PER_SUBNET_STATS` - enable or disable collecting per subnet stats from Kea; default is `true` (collecting enabled). You can use this option to limit the data passed to Prometheus/Grafana in large networks.

- 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 an agent token:

- STORK_AGENT_SERVER_URL - the `stork-server` URL used by the agent to send REST commands to the server during agent registration

Warning: `stork-server` does not currently support communication with `stork-agent` via an IPv6 link-local address with zone ID (e.g., `fe80::%eth0`). This means that the STORK_AGENT_HOST variable must be set to a DNS name, an IPv4 address, or a non-link-local IPv6 address.

2.5.2.2 Colorization Settings

To control the logging colorization, Stork supports the CLICOLOR and CLICOLOR_FORCE standard UNIX environment variables. When set, the following rules will be applied:

- **CLICOLOR_FORCE != 0** ANSI colors should be enabled no matter what.
- **CLICOLOR_FORCE == 0** Don't output ANSI color escape codes.
- **CLICOLOR_FORCE is unset and CLICOLOR == 0** Don't output ANSI color escape codes.
- **Otherwise** ANSI colors are enabled if TTY is used.

For example, to disable the output colorization:

```
rake run:agent CLICOLOR=0
```

Note: The `true` and `false` values are also accepted instead of the `1` and `0`.

2.5.2.3 Securing Connections Between the Stork Server and a Stork Agent

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 at 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 an agent token
2. using a 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 administrator 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 common token 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. When this registration method is used, the Stork agent is typically installed using a script that downloads the agent packages embedded in the server.

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

The installation and registration processes using each method are described in the subsequent sections.

2.5.2.4 Securing Connections Between `stork-agent` and the Kea Control Agent

The Kea Control Agent (CA) may be configured to accept connections only over TLS. It requires specifying `trust-anchor`, `cert-file` and `key-file` values in the `kea-ctrl-agent.conf` file. For details, see the Kea Administrator Reference Manual.

The Stork agent can communicate with Kea over TLS, via the same certificates that it uses in communication with the Stork server.

The Stork agent by default requires that the Kea Control Agent provide a trusted TLS certificate. If Kea uses a self-signed certificate, the Stork agent can be launched with the `--skip-tls-cert-verification` flag or `STORK_AGENT_SKIP_TLS_CERT_VERIFICATION` environment variable set to 1, to disable Kea certificate verification.

The Kea CA accepts only requests signed with a trusted certificate, when the `cert-required` parameter is set to `true` in the Kea CA configuration file. In this case, the Stork agent must use valid certificates; it cannot use self-signed certificates created during Stork agent registration.

Kea 1.9.0 added support for basic HTTP authentication to control access for incoming REST commands over HTTP. If the Kea CA is configured to use Basic Auth, valid credentials must be provided in the Stork agent credentials file: `/etc/stork/agent-credentials.json`.

By default, this file does not exist, but the `/etc/stork/agent-credentials.json.template` file provides example data. The template file can be renamed by removing the `.template` suffix; then the file can be edited and valid credentials can be provided. The `chown` and `chmod` commands should be used to set the proper permissions; this file contains the secrets, and should be readable/writable only by the user running the Stork agent and any administrators.

Warning: Basic HTTP authentication is weak on its own as there are known dictionary attacks, but those attacks require a "man in the middle" to get access to the HTTP traffic. That can be eliminated by using basic HTTP authentication exclusively over TLS. In fact, if possible, using client certificates for TLS is better than using basic HTTP authentication.

For example:

```
{
   "basic_auth": [
      {
         "ip": "127.0.0.1",
         "port": 8000,
         "user": "foo",
```

(continues on next page)

(continued from previous page)

```
        "password": "bar"
      }
   ]
}
```

It contains a single object with a single "basic" key. The "basic" value is a list of the Basic Auth credentials. All credentials must contain the values for four keys:

- `ip` - the IPv4 or IPv6 address of the Kea CA. It supports IPv6 abbreviations (e.g. "FF:0000::" is the same as "ff::").
- `port` - the Kea Control Agent port number.
- `user` - the Basic Auth user ID to use in connection with a specific Kea CA.
- `password` - the Basic Auth password to use in connection with a specific Kea CA.

To apply changes in the credentials file, the `stork-agent` daemon must be restarted.

If the credentials file is invalid, the Stork agent will run but without Basic Auth support. The notice will be indicated with a specific message in the log.

2.5.2.5 Installation From Cloudsmith and Registration With an Agent Token

This section describes how to install an agent from the Cloudsmith repository and perform 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 the local Linux distribution:

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

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

instead of the server installation commands.

Next, specify the required settings in the `/etc/stork/agent.env` file. The `STORK_AGENT_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_HOST` 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 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
```

The following log messages should be returned 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 a machine can be monitored. Visit the `Services -> Machines` page in the Stork UI, and click the `Unauthorized` button located above the list of machines on the right side. This list contains all machines pending registration approval. Before authorizing a 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.

2.5.2.6 Installation With a Script and Registration With a Server Token

This section describes how to install an agent using a script and packages downloaded from the Stork server and register the agent 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 and 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
```

`stork.example.org` is an example URL for the Stork server; it must be replaced with the real server URL used in the 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` page. Copy the server token from the dialog box and paste it in at 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 fully qualified domain name (FQDN) that 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 the local settings.

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

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

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

2.5.2.7 Installation With a Script and Registration With an Agent Token

This section describes how to install an agent using a script and packages downloaded from the Stork server and perform the agent's registration using an agent token. It is an interactive 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* section.

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 that 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 the local settings.

The following log messages should be returned when the agent successfully sends the registration request to the server:

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

As with *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.

2.5.2.8 Installation From Cloudsmith and Registration With a Server Token

This section describes how to install an agent from the Cloudsmith repository and perform 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 the Linux distribution:

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

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

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:8080'
```

The last parameter should be the appropriate Stork server URL.

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

2.5.2.9 Registration Methods Summary

Stork supports two different agent-registration methods, described above. Both methods can be used interchangeably, and it is often a matter of preference which one the administrator selects. However, 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 sends the registration request to the server and awaits 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 might 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 any new registrations.

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

2.5.2.10 Agent Setup Summary

After successful agent setup, the agent periodically tries to detect installed Kea DHCP or BIND9 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.

2.5.2.11 Inspecting Keys and Certificates

The Stork server maintains TLS keys and certificates internally to secure the communication between `stork-server` and any agents. They can be inspected and exported using `stork-tool`, with a command such as:

```
$ stork-tool cert-export --db-url postgresql://user:pass@localhost/dbname -f srvcert -o␣
↪srv-cert.pem
```

The above command may fail if the database password contains any characters requiring URL encoding. In this case, a command line with multiple switches can be used instead:

```
$ stork-tool cert-export --db-user user --db-password pass --db-host localhost --db-name␣
↪dbname -f srvcert -o srv-cert.pem
```

The certificates and secret keys can be inspected using OpenSSL, using commands such as `openssl x509 -noout -text -in srv-cert.pem` (for the certificates) and `openssl ec -noout -text -in cakey` (for the keys).

There are five secrets that can be exported or imported: the Certificate Authority secret key (`cakey`), the Certificate Authority certificate (`cacert`), the Stork server private key (`srvkey`), the Stork server certificate (`srvcert`), and a server token (`srvtkn`).

For more details, please see *stork-tool - A Tool for Managing Stork Server*.

2.5.2.12 Using External Keys and Certificates

It is possible to use external TLS keys and certificates. They can be imported to the Stork server using `stork-tool`:

```
$ stork-tool cert-import --db-url postgresql://user:pass@localhost/dbname -f srvcert -i␣
↪srv-cert.pem
```

The above command may fail if the database password contains any characters requiring URL encoding. In this case, a command line with multiple switches can be used instead:

```
$ stork-tool cert-import --db-user user --db-password pass --db-host localhost --db-name␣
↪dbname -f srvcert -i srv-cert.pem
```

Both the Certificate Authority key and the Certificate Authority certificate must be changed at the same time, as the certificate depends on the key. If they are changed, then the server key and certificate must also be changed.

The ability to use external certificates and keys is considered experimental.

For more details, please see *stork-tool - A Tool for Managing Stork Server*.

2.5.3 Upgrading

Due to the new security model introduced with TLS in Stork 0.15.0, upgrades from versions 0.14.0 and earlier require the agents to be re-registered.

The server upgrade procedure is the same as the initial installation procedure.

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

2.6 Installing From Sources

2.6.1 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 the 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, which does not require root privileges. If the demo environment will be run, Docker is needed but not Java; Docker installs Java within a container.

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

2.6.2 Download Sources

The Stork sources are available in ISC's GitLab instance: https://gitlab.isc.org/isc-projects/stork.

To get the latest sources invoke:

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

2.6.3 Building

There are two Stork components:

- `stork-agent` - this is a binary, written in Go
- `stork-server` - this is comprised of two parts: - backend service - a binary, written in Go - frontend - an Angular application written in Typescript

All components can be built using the following command:

```
$ rake build
```

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 in the specific system directories; this is useful for installation in a production environment. For the testing purposes tt can be customized via the `DEST` variable, e.g.:

```
$ rake install:server DEST=/home/user/stork
```

2.6.4 Installing on FreeBSD

Stork is not regularly tested on FreeBSD but can be installed on this operating system with the manual steps provided below.

The first step is the installation of packages from the repository:

```
pkg install ruby
pkg install rubygem-rake
pkg install wget
pkg install openjdk11-jre
pkg install node14
pkg install npm-node14
npm install -g npm
pkg install python3
pkg install protobuf
pkg install gcc
pkg install gtar
```

The utility to build the packages requires the GNU tar in PATH. The BSD tar isn't compatible. We need to rename the existing executable.

```
mv /usr/bin/tar /usr/bin/bsdtar
ln -s /usr/local/bin/gtar /usr/bin/tar
```

Stork build system can install all remaining dependencies automatically.

The binary packages can be built using:

```
rake build:server_pkg
rake build:agent_pkg
```

The output binaries will be located in the `dist/pkgs/` directory and can be installed using the `pkg install` command.

2.6.5 Installing on OpenBSD

Stork is not regularly tested on OpenBSD but can be installed on this operating system with the manual steps provided below. The installation guide is similar to FreeBSD one.

The first step is the installation of packages from the repository:

```
pkg_add ruby
ln -s /usr/local/bin/gem31 /usr/local/bin/gem
gem install --user-install rake
pkg_add wget
pkg_add jdk
pkg_add node
pkg_add unzip
pkg_add protobuf
pkg_add gcc
pkg_add go
```

Stork requires Golang version 1.18 or later.

Stork build system can install all remaining dependencies automatically.

Unfortunately, there is no possibility to build the binary packages for OpenBSD. But it is possible to build the contents of the packages (executables, UI, man, and docs).

```
rake build:server_dist
rake build:agent_dist
```

The output files will be located in the `dist/` directory.

2.7 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.

2.7.1 Prometheus Integration

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

```
# 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 to Monitor BIND 9 and Kea services*.

The Stork server can also be optionally integrated, but Prometheus support for it is disabled by default. To enable it, run the server with the `-m` or `--metrics` flag or set the `STORK_SERVER_ENABLE_METRICS` environment variable. Next, update the `prometheus.yml` file:

```
# statistics from Stork Server
- job_name: 'storkserver'
   static_configs:
      - targets: ['server.example.org:8080']
```

The Stork server exports metrics on the assigned HTTP/HTTPS port (defined via the `--rest-port` flag).

Note: The Prometheus client periodically collects metrics from the clients (`stork-server` or `stork-agent`, for example), via an HTTP call. By convention, the endpoint that shares the metrics has the `/metrics` path. This endpoint returns data in Prometheus-specific format.

Warning: The Prometheus `/metrics` endpoint does not require authentication. Therefore, securing this endpoint from external access is highly recommended to prevent unauthorized parties from gathering the server's metrics. One way to restrict endpoint access is by using an appropriate HTTP proxy configuration to allow only local access or access from the Prometheus host. Please consult the NGINX example configuration file shipped with Stork.

After restarting, the Prometheus web interface can be used to inspect whether the statistics have been 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`); and Stork server statistics use the `storkserver_` prefix.

2.7.2 Alerting in Prometheus

Prometheus provides the ability to configure alerting. A good starting point is the Prometheus documentation on alerting. Briefly, the three main steps are: configure the Alertmanager; configure Prometheus to talk to the Alertmanager; and define the alerting rules in Prometheus. There are no specific requirements or recommendations, as these are very deployment-dependent. The following is an incomplete list of ideas that could be considered:

- The `storkserver_auth_unreachable_machine_total` metric is reported by `stork-server` and shows the number of unreachable machines. Its value under normal circumstances should be zero. Configuring an alert for non-zero values may be the best indicator of a large-scale problem, such as a whole VM or server becoming unavailable.
- The `storkserver_auth_authorized_machine_total` and `storkserver_auth_unauthorized_machine_total` metrics may be used to monitor situations when new machines (e.g. by automated VM cloning) may appear in the network or existing machines may disappear.
- The `kea_dhcp4_addresses_assigned_total` metric, along with `kea_dhcp4_addresses_total`, can be used to calculate pool utilization. If the server allocates all available addresses, it will not be able to handle new devices, which is one of the most common failure cases of the DHCPv4 server. Depending on the deployment specifics, a threshold alert when the pool utilization approaches 100% should be seriously considered.
- Contrary to popular belief, DHCPv6 can also run out of resources, in particular with prefix delegation (PD). The `kea_dhcp6_pd_assigned_total` metric divided by `kea_dhcp6_pd_total` can be considered an indicator of PD pool utilization. It is an important metric if PD is being used.

The alerting mechanism configured in Prometheus has the relative advantage of not requiring an additional component (Grafana). The alerting rules are defined in a text file using simple YAML syntax. For details, see the Prometheus documentation on alerting rules. One potentially important feature is Prometheus' ability to automatically discover available Alertmanager instances, which may be helpful in various redundancy considerations. The Alertmanager provides a rich list of receivers, which are the actual notification mechanisms used: email, PagerDuty, Pushover, Slack, Opsgenie, webhook, WeChat, and more.

ISC makes no specific recommendations between Prometheus or Grafana. This is a deployment consideration.

2.7.3 Grafana Integration

Stork provides several Grafana templates that can easily be imported, available in the `grafana/` directory of the Stork source code. The currently available templates are `bind9-resolver.json`, `kea-dhcp4.json`, and `kea-dhcp6.json`. Grafana integration requires three steps:

1. Prometheus must be added as a data source. This can be done in several ways, including using 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; then specify the necessary parameters to connect to the Prometheus instance. In

test environments, the only 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 for 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
```

The Grafana dashboard files can also be copied to `/var/lib/grafana/dashboards/` (again, the exact location may vary depending on the installation).

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

2.7.4 Subnet Identification

The Kea Control Agent shares subnet statistics labeled with the internal Kea IDs. The Prometheus/Grafana subnet labels depend on the installed Kea hooks. By default, the internal, numeric Kea IDs are used. However, if the `subnet_cmds` hook is installed, then the numeric IDs are resolved to subnet prefixes. This makes the Grafana dashboard more human-friendly and descriptive.

2.7.5 Alerting in Grafana

Grafana offers multiple alerting mechanism options that can be used with Stork; users are encouraged to see the Grafana page on alerting.

The list of notification channels (i.e. the delivery mechanisms) is extensive, as it supports email, webhook, Prometheus' Alertmanager, PagerDuty, Slack, Telegram, Discord, Google Hangouts, Kafka REST Proxy, Microsoft Teams, Opsgenie, Pushover, and more. Existing dashboards provided by Stork can be modified and new dashboards can be created. Grafana first requires a notification channel to be configured (Alerting -> Notifications Channel menu). Once configured, existing panels can be edited with alert rules. One caveat is that most panels in the Stork dashboards use template variables, which are not supported in alerting. This stackoverflow thread discusses several ways to overcome this limitation.

Compared to Prometheus alerting, Grafana alerting is a bit more user-friendly. The alerts are set using a web interface, with a flexible approach that allows custom notification messages, such as instructions on what to do when receiving an alert, information on how to treat situations where received data is null or there is a timeout, etc.

The defined alerts are considered an integral part of a dashboard. This may be a factor in a deployment configuration, e.g. the dashboard can be tweaked to specific needs and then deployed to multiple sites.

CHAPTER

THREE

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 on the Stork server machine. If `stork-server` 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 when Stork is initially installed. 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:

- The `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, each user is associated with one of the two predefined groups (roles), which are `super-admin` or `admin`; one of these must be selected when a 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 their 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 interface. 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 is 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 the URLs of the Prometheus and Grafana instances used with Stork to be specified.

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 Disconnecting From 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 terminates 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. For example, when the Stork Agent has been installed from packages, run:

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

Alternatively:

```
$ sudo killall -9 stork-agent
```

3.4.3 Dumping Diagnostic Information Into a File

It is sometimes difficult or impossible to diagnose issues without seeing the actual logs, database contents, and configuration files. Gathering such information can be challenging for a user because it requires looking into many places like databases, remote machine logs, etc.

Stork makes it convenient for users to gather diagnostic information from the selected machines with a single click. Navigate to the `Machines` page (i.e., the page where all monitored machines are listed), click on the `Action` button for the selected machine, and choose the `Dump Troubleshooting Data` option. Alternatively, navigate to the selected machine's page and click on the `Dump Troubleshooting Data` button at the bottom of the page. In both cases, the Stork server will automatically gather useful diagnostics information and offer it for download as a `tar.gz` file. The downloaded package contains configurations, log tails, `stork-server` settings, warning and error-level events, high availability services' states etc.

Note: Stork sanitizes passwords and other sensitive information when it creates the package.

The tarball can be easily sent via email or attached to a bug report.

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 columns; typically, they include: DHCPv4, DHCPv6, DDNS, and Kea Control Agent (CA).

Stork uses `rndc` to retrieve the application's 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 if `named` is built with `json-c`, because it gathers statistics via the JSON statistics API. The `named.conf` file must have `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 is `kea@abc.example.org`. Similarly, if a BIND 9 app runs on the machine with the address `192.0.2.3`, the resulting app name is `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.

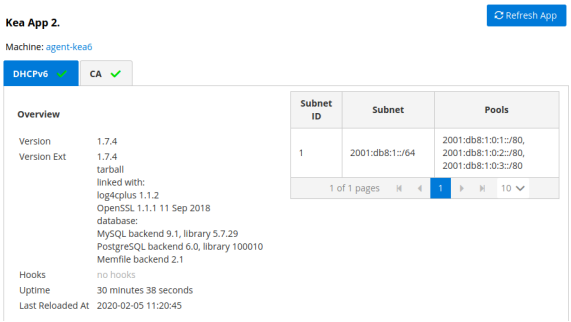
Suppose there are two DHCP servers in the network, one on the first floor of a building and one on the second floor. A user may assign the names `Floor 1 DHCP` and `Floor 2 DHCP` to the respective DHCP servers in this case. The new names need not have the same pattern as the default names and may contain 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 to explain the issues with the new name.

3.5.3 IPv4 and IPv6 Subnets per Kea Application

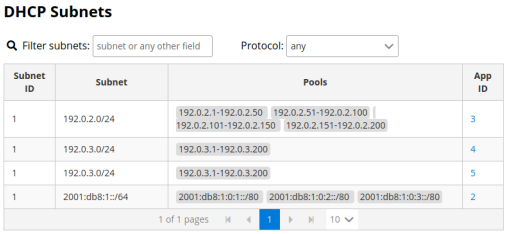
One of the primary configuration aspects of any network is the layout of IP addressing. This is represented in Kea with IPv4 and IPv6 subnets. Each subnet represents addresses used on a physical link. Typically, certain parts of each subnet ("pools") are delegated to the DHCP server to manage. Stork 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. 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 range, 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 the inspection of networks and the subnets that belong in them. Pool utilization is shown for each subnet.

3.5.6 Host Reservations

3.5.6.1 Listing 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 assignment of 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 hook library.

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

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.

The next two columns contain the static assignments of the IP addresses and/or delegated prefixes to the clients. There may be one or more such 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 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 hook 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`.

3.5.6.2 Host Reservation Usage Status

Clicking on a selected host in the host reservations list opens a new tab that shows host details. The tab also includes information about reserved addresses and delegated prefixes usage. Stork needs to query Kea servers to gather the lease information for each address and prefix in the selected reservation. It may take several seconds or longer before this information is available. The lease information can be refreshed using the `Leases` button at the bottom of the tab.

The usage status is shown next to each IP address and delegated prefix. Possible statuses and their meanings are listed in the table below.

Table 1: Possible IP reservation statuses

Status	Meaning
`in use`	There are valid leases assigned to the client. The client owns the reservation, or the reservation includes the `flex-id` or `circuit-id` identifier, making it impossible to detect conflicts (see note below).
`expired`	At least one of the leases assigned to the client owning the reservation is expired.
`declined`	The address is declined on at least one of the Kea servers.
`in conflict`	At least one of the leases for the given reservation is assigned to a client that does not own this reservation.
`unused`	There are no leases for the given reservation.

View status details by expanding a selected address or delegated prefix row. Clicking on the selected address or delegated prefix navigates to the leases search page, where all leases associated with the address or prefix can be listed.

Note: Detecting `in conflict` status is currently not supported for host reservations with `flex-id` or `circuit-id` identifiers. If there are valid leases for such reservations, they are marked `in use` regardless of whether the conflict exists.

3.5.6.3 Sources of Host Reservations

There are two ways to configure 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 or PostgreSQL. The second solution requires the given Kea server to be configured to use the `host_cmds` premium hook library. This library implements control commands used to store and fetch the host reservations from the host database to which the Kea server is connected. If the `host_cmds` hook 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 is not visible in Stork until up to 60 seconds after it was applied. This interval is configurable in the Stork interface.

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.

3.5.6.4 Creating Host Reservations

Above the list of the host reservations, there is the `New Host` button that opens a tab where a user can specify a new host reservation in one or more Kea servers. These Kea servers must be configured to use the `host_cmds` hooks library, and only these servers are available for selection in the `DHCP Servers` dropdown.

A user has a choice between a subnet-level or global host reservation. Selecting a subnet using the `Subnet` dropdown is required for a subnet-level reservation. If the desired subnet is not displayed in the dropdown, it is possible that the selected DHCP servers do not include this subnet in their configuration. Setting the `Global reservation` option disables subnet selection.

To associate the new host reservation with a DHCP client, the user can select one of the identifier types supported by Kea. Available identifiers differ depending on whether the user selected DHCPv4 or DHCPv6 servers. The identifier can be specified using `hex` or `text` format. For example, the `hw-address` is typically specified as a string of hexadecimal digits: `ab:76:54:c6:45:31`. In that case, select `hex` option. Some identifiers, e.g. `circuit-id`, are often specified using "printable characters", e.g. `circuit-no-1`. In that case, select `text` option. Please refer to Host Reservations in DHCPv4 and Host Reservations in DHCPv6 for more details regarding allowed DHCP identifiers and their formats.

Further in the form, the user can specify the actual reservations. It is possible to specify at most one IPv4 address. In the case of the DHCPv6 servers, it is possible to specify multiple IPv6 addresses and delegated prefixes.

`Hostname` is currently the only supported non-IP reservation type besides DHCP options.

DHCP options can be added to the host reservation by clicking the `Add Option` button. The list of the standard DHCP options is available via the dropdown. However, if the list is missing a desired option, the user can simply type the option code in the dropdown. The `Always Send` checkbox specifies whether the option should always be returned to a DHCP client assigned this host reservation, regardless of whether the client requests this option from the DHCP server.

In the current Stork version, the user must explicitly select an option payload suitable for the option. Thus, they must be familiar with the DHCP option formats and select appropriate option fields in the right order using the `Add <field-type>` button below the option code. For example, the `(5) Name Server` option can comprise one or more IPv4 addresses. After selecting this option, the user should select an `ipv4-address` option field once or more and fill the option fields with the IP addresses.

Note: Currently, Stork does not verify whether or not the specified options comply with the formats specified in the RFCs, nor does it check them against the runtime option definitions configured in Kea. If the user specifies wrong

option format, Stork will try to send the option to Kea for verification, and Kea will reject the new reservation. The reservation can be submitted again after correcting the option payload.

Please use the `Add <field-type>` button to add suboptions to a DHCP option. Stork supports top-level options with maximum two levels of suboptions.

If a host reservation is configured in several DHCP servers, typically, all servers comprise the same set of parameters (i.e., IP addresses, hostname, and DHCP options). By default, creating a new host reservation for several servers sends an identical copy of the host reservation to each. A user may choose to specify a different set of options for different servers by selecting `Toggle editing DHCP options individually for each server` at the top of the form. In this case, the user must specify the complete option sets for each DHCP server. Leaving options blank for some servers means that these servers receive no DHCP options with the reservation.

Submitted host reservations may appear in Stork's host reservations list with some delay. Please allow some time for the reservations to propagate to the Kea DHCP servers and refresh the list.

3.5.6.5 Updating Host Reservations

In a selected host reservation's view, click `Edit` button to open a form for editing host reservation information. The form automatically toggles editing DHCP options individually for each server (see above) when it detects different option sets on different servers using the reservation. Besides editing the host reservation information, it is also possible to deselect some of the servers (using the DHCP Servers dropdown), which will delete the reservation from these servers.

Use the `Revert Changes` button to remove all applied changes and restore the original host reservation information. Use `Cancel` to close the form without applying the changes.

3.5.6.6 Deleting Host Reservations

To delete a host reservation from all DHCP servers for which it is configured, click on the reservation in the host reservations list. Find the `Delete` button and confirm the reservation deletion. Use it with caution because this operation cannot be undone. The reservation is removed from the DHCP servers' databases. It must be re-created to be restored.

Note: The `Delete` button is unavailable for host reservations configured in the Kea configuration files or when the reservations are configured in the host database, but the `host_cmds` hook library is not loaded.

3.5.7 Leases Search

Stork can search DHCP leases on monitored Kea servers, which 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 any Kea servers on which he intends to search the leases have the lease commands hook library loaded. Stork does not search leases on Kea instances without this library.

The leases search is available via the `DHCP -> Leases Search` menu. Enter 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`. Be aware that this query may return a large result if there are many declined leases, and thus 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. If some Kea servers are unavailable or return an error, 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's occurrences. For example, if there is a pair of servers cooperating via the High Availability hook 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.

3.5.8 Kea High Availability Status

When viewing the details of the Kea application for which High Availability (HA) is enabled (via the `libdhcp_ha.so` hook 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.

High Availability

Local server	Remote server (4 seconds ago)
State: *load-balancing*	State: *load-balancing*
Role: *primary*	Role: *secondary*
Scopes served: *server1*	Scopes served: *(none)*

Note
The local server responds to the entire DHCP traffic.

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; 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 returned by Stork includes the age of the data displayed.

The Stork 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. 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 also contains information about the heartbeat status between the two servers and information about failover progress.

The failover progress information is only presented when one of the active servers has been unable to communicate with the partner via refthe 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 a lease), connecting clients (all clients currently trying to get a 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 the High Availability status information provided by Kea can be found in the Kea ARM.

3.5.9 Viewing the Kea Log

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

Note: This mechanism currently 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 a 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. Each 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.

3.5.10 Viewing the Kea Configuration as a JSON Tree

Kea uses JavaScript Object Notation (JSON) to represent its configuration in the configuration files and the command channel. Parts of the Kea configuration held in the Configuration Backend are also converted to JSON and returned over the control channel in that format. Diagnosis of issues with a particular server often begins by inspecting its configuration.

In the `Kea App` view, select the appropriate tab for the daemon configuration to be inspected, and then click on the `Raw Configuration` button. The displayed tree view comprises the selected daemon's configuration fetched using the Kea `config-get` command.

Note: The `config-get` command returns the configuration currently in use by the selected Kea server. It is a combination of the configuration read from the configuration file and from the config backend, if Kea uses the backend. Therefore, the configuration tree presented in Stork may differ (sometimes significantly) from the configuration file contents.

The nodes with complex data types can be individually expanded and collapsed. All nodes can also be expanded or collapsed by toggling the `Expand` button. When expanding nodes with many sub-nodes, they may be paginated to avoid degrading browser performance.

Click the `Refresh` button to fetch and display the latest configuration. Click `Download` to download the entire configuration into a text file.

Note: Some of the configuration fields may contain sensitive data (e.g. passwords or tokens). The content of these fields is hidden, and a placeholder is shown. Configurations downloaded as JSON files by users other than super-admins contain null values in place of the sensitive data.

3.5.11 Configuration Review

Kea DHCP servers are controlled by numerous configuration parameters. It poses a risk of misconfiguration or inefficient server operation when the parameters are misused. Stork can help determine typical problems in a Kea server configuration using built-in configuration checkers.

It generates configuration reports for a monitored Kea daemon when it detects its configuration has changed. To view the reports for the daemon, navigate to the application page and select one of the daemons. The `Configuration Review Reports` panel lists issues and proposed configuration updates generated by the configuration checkers. Each checker focuses on one particular problem.

If you consider some of the reports false alarms in your deployment, you can disable some configuration checkers for a selected daemon or globally for all daemons. Click the `Checkers` button to open the list of available checkers and their current state. Click on the values in the `State` column for the respective checkers until they are in the desired states. Besides enabling and disabling the checker, it is possible to configure it to use the globally specified setting (i.e., globally enabled or globally disabled). The global settings control the checker states for all daemons for which explicit states are not selected.

Select `Configuration -> Review checkers` from the main menu to modify the global states. Use the checkboxes in the `State` column to modify the global states for respective checkers.

The `Selectors` listed for each checker inform about the types of daemons whose configurations they validate:

- `each-daemon` - run for all types of daemons,
- `kea-daemon` - run for all Kea daemons,
- `kea-ca-daemon` - run for Kea Control Agents,

- `kea-dhcp-daemon` - run for DHCPv4 and DHCPv6 daemons,
- `kea-dhcp-v4-daemon` - checkers run for Kea DHCPv4 daemons,
- `kea-dhcp-v6-daemon` - run for Kea DHCPv6 daemons
- `kea-d2-daemon` - run for Kea D2 daemons,
- `bind9-daemon` - run for Bind 9 daemons

The triggers inform in which cases the checkers are executed. Currently, there are three types of triggers:

- `manual` - run on user's request,
- `config change` - run when daemon configuration change has been detected,
- `host reservations change` - run when a change in the Kea host reservations database has been detected.

The selectors and triggers are not configurable by a user.

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 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

FOUR

TROUBLESHOOTING

4.1 stork-agent

This section describes the solutions for some common issues with the Stork agent.

Issue A machine is authorized in the Stork server successfully, but there are no applications.

Description The user installed and started `stork-server` and `stork-agent` and authorized the machine. The "Last Refreshed" column has a value on the Machines page, the "Error" column value shows no errors, but the "Daemons" column is still blank. The "Application" section on the specific Machine page is also blank.

Solution Make sure that the daemons are running:

- Kea Control Agent, Kea DHCPv4 server, and/or Kea DHCPv6 server
- BIND 9

Explanation If the "Last Refreshed" column has a value, and the "Error" column value has no errors, the communication between `stork-server` and `stork-agent` works correctly, which implies that the cause of the problem is between the Stork agent and the daemons. The most likely issue is that none of the Kea/BIND 9 daemons are running. `stork-agent` communicates with the BIND 9 daemon directly; however, it communicates with the Kea DHCPv4 and Kea DHCPv6 servers via the Kea Control Agent. If only the "CA" daemon is displayed in the Stork interface, the Kea Control Agent is running, but the DHCP daemons are not.

Issue After starting the Stork agent, it gets stuck in an infinite "sleeping" loop.

Description `stork-agent` is running with server support (the `--listen-prometheus-only` flag is unused). The `try to register agent in Stork server` message is displayed initially, but the agent only prints the recurring `sleeping for 10 seconds before next registration attempt` message.

Solution 1. `stork-server` is not running. Start the Stork server first and restart the `stork-agent` daemon.

Solution 2. The configured server URL in `stork-agent` is invalid. Correct the URL and restart the agent.

Issue After starting `stork-agent`, it keeps printing the following messages: `loaded server cert: /var/lib/stork-agent/certs/cert.pem and key: /var/lib/stork-agent/certs/key.pem`

Description `stork-agent` runs correctly, and its registration is successful. After the `started serving Stork Agent` message, the agent prints the recurring message about loading server certs. The network traffic analysis to the server reveals that it rejects all packets from the agent (TLS HELLO handshake failed).

Solution Re-register the agent to regenerate the certificates, using the `stork-agent register` command.

Explanation The `/var/lib/stork-agent/certs/ca.pem` file is missing or corrupted. The re-registration removes old files and creates new ones.

Issue The cert PEM file is not loaded.

Description The agent fails to start and prints an `open /var/lib/stork-agent/certs/cert.pem: no such file or directory could not load cert PEM file: /var/lib/stork-agent/certs/cert.pem` error message.

Solution Re-register the agent to regenerate the certificates, using the `stork-agent register` command.

Issue A connection problem to the DHCP daemon(s).

Description The agent prints the message `problem with connecting to dhcp daemon: unable to forward command to the dhcp6 service: No such file or directory. The server is likely to be offline`.

Solution Try to start the Kea service: `systemctl start kea-dhcp4 kea-dhcp6`

Explanation The `kea-dhcp4.service` or `kea-dhcp6.service` (depending on the service type in the message) is not running. If the above commands do not resolve the problem, check the Kea Administrator Reference Manual (ARM) for troubleshooting assistance.

Issue `stork-agent` receives a `remote error: tls: certificate required` message from the Kea Control Agent.

Description The Stork agent and the Kea Control Agent are running, but they cannot establish a connection. The `stork-agent` log contains the error message mentioned above.

Solution Install the valid TLS certificates in `stork-agent` or set the `cert-required` value in `/etc/kea/kea-ctrl-agent.conf` to `false`.

Explanation By default, `stork-agent` does not use TLS when it connects to Kea. If the Kea Control Agent configuration includes the `cert-required` value set to `true`, it requires the Stork agent to use secure connections with valid, trusted TLS certificates. It can be turned off by setting the `cert-required` value to `false` when using self-signed certificates, or the Stork agent TLS credentials can be replaced with trusted ones.

Issue Kea Control Agent returns a `Kea error response - status: 401, message: Unauthorized` message.

Description The Stork agent and the Kea Control Agent are running, but they cannot connect. The `stork-agent` logs contain similar messages: `failed to parse responses from Kea: { "result": 401, "text": "Unauthorized" }` or `Kea error response - status: 401, message: Unauthorized`.

Solution Update the `/etc/stork/agent-credentials.json` file with the valid user/password credentials.

Explanation The Kea Control Agent can be configured to use Basic Authentication. If it is enabled, valid credentials must be provided in the `stork-agent` configuration. Verify that this file exists and contains a valid username, password, and IP address.

Issue During the registration process, `stork-agent` prints a `problem with registering machine: cannot parse address` message.

Description Stork is configured to use an IPv6 link-local address. The agent prints the `try to register agent in Stork server` message and then the above error. The agent exists with a fatal status.

Solution Use a global IPv6 or an IPv4 address.

Explanation IPv6 link-local addresses are not supported by `stork-server`.

Issue A protocol problem occurs during the agent registration.

Description During the registration process, `stork-agent` prints a `problem with registering machine: Post "/api/machines": unsupported protocol scheme ""` message.

Solution The `--server-url` argument is provided in the wrong format; it must be a canonical URL. It should begin with the protocol (`http://` or `https://`), contain the host (DNS name or IP address; for IPv6 escape them with square brackets), and end with the port (delimited from the host by a colon). For example: `http://storkserver:8080`.

Issue The values in `/etc/stork/agent.env` or `/etc/stork/agent-credentials.json` were changed, but `stork-agent` does not register the changes.

Solution Restart the daemon.

Explanation `stork-agent` reads configurations only at startup.

Issue The values in `/etc/stork/agent.env` were changed and the daemon was restarted, but the agent still uses the default values.

Description The agent is running using the `stork-agent` command. It uses the parameters passed from the command line but ignores the `/etc/stork/agent.env` file entries. If the agent is running as the systemd daemon, it uses the expected values.

Solution Load the environment variables from the `/etc/stork/agent.env` file before running the CLI tool. For example, run `. /etc/stork/agent.env`.

Explanation The `/etc/stork/agent.env` file contains the environment variables, but `stork-agent` does not automatically load them; the file must be loaded manually. The default systemd service unit is configured to load this file before starting the agent.

CHAPTER

FIVE

BACKEND API

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

CHAPTER

SIX

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.

6.1 Rakefile

Rakefile is a script for performing many development tasks, like building source code, running linters and 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.

6.2 Generating Documentation

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

6.3 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. 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 build:ui_live
```

```
$ 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.

6.3.1 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.

6.4 Agent API

The connection between `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",
  "memory": "16",
  "usedMemory": "59",
  "uptime": "2",
  "os": "darwin",
  "platform": "darwin",
  "platformFamily": "Standalone Workstation",
  "platformVersion": "10.14.6",
  "kernelVersion": "18.7.0",
```

(continues on next page)

(continued from previous page)

```
  "kernelArch": "x86_64",
  "hostID": "c41337a1-0ec3-3896-a954-a1f85e849d53"
}
```

6.5 RESTful API

The primary user of the RESTful API is the Stork UI in a web browser. The definition of the RESTful 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`, which then generates code for:

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

All these steps are accomplished by Rakefile.

6.6 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.

6.6.1 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 `DB_HOST` environment variable, e.g.:

```
$ rake unittest:backend DB_HOST=host:port
```

By default it points to `localhost:5432`.

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

```
$ rake unittest:backend DB_PASSWORD=secret123
```

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

6.6.2 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.

6.6.3 Benchmarks

Benchmarks are part of 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 new logic to a function often causes performance degradation, and careful examination of the benchmark result drop for that function may drive improved efficiency of the new code.

6.6.4 Short Testing Mode

It is possible to filter out long-running unit tests, by setting the `SHORT` variable to `true` on the command line:

```
$ rake unittest:backend SHORT=true
```

6.7 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 unittest:ui
```

The tests require the Chromium (on Linux) or Chrome (on Mac) browser. The `rake unittest:ui` 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 unittest:ui
```

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 unittest:ui 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 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.

```
TEST=src/app/ha-status-panel/ha-status-panel.component.spec.ts rake unittest:ui
```

By default, all tests are executed. To run only a specific test file, set the "TEST" environment variable to a relative path to any `.spec.ts` file (relative to the project directory).

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.

6.8 System Tests

Stork system tests interact with its REST API to ensure proper server behavior, error handling, and stable operation for malformed requests. Depending on the test case, the system testing framework can automatically set up and run Kea or Bind9 daemons and the Stork Agents the server will interact with during the test. It runs these daemons inside the Docker containers.

6.8.1 Dependencies

System tests require:

- Linux or macOS operating system (Windows and BSD were not tested)
- Python >= 3.18
- Rake (as a launcher)
- Docker
- docker-compose >= 1.28

6.8.2 Initial steps

A user must be a member of the `docker` group to run the system tests. The following commands create create this group and add the current user to it on Linux.

1. Create the docker group.

```
$ sudo groupadd docker
```

2. Add your user to the `docker` group.

```
$ sudo usermod -aG docker $USER
```

3. Log out and log back in so that your group membership is re-evaluated.

6.8.3 Running System Tests

After preparing all the dependencies, the tests can be started using the following command:

```
$ rake systemtest
```

This command first prepares all necessary toolkits (except these listed above) and configuration files. Next, it calls `pytest`, a Python testing framework used in Stork for executing the system tests.

Some test cases use the premium Kea hooks. They are disabled by default. To enable them, specify the valid CloudSmith access token in the CS_REPO_ACCESS_TOKEN variable.

```
$ rake systemtest CS_REPO_ACCESS_TOKEN=<access token>
```

Test results for individual test cases are shown at the end of the tests execution.

Warning: Users should not attempt to run the system tests by directly calling pytest because it would bypass the step to generate the necessary configuration files. This step is conducted by the rake tasks.

To run a particular test case, specify its name in the TEST variable:

```
$ rake systemtest TEST=test_users_management
```

To list available tests without actually running them, use the following command:

```
$ rake systemtest:list
```

To run the test cases with a specific Kea version, provide it in the KEA_VERSION variable:

```
$ rake systemtest KEA_VERSION=2.0
$ rake systemtest KEA_VERSION=2.0.2
$ rake systemtest KEA_VERSION=2.0.2-isc20220227221539
```

Accepted version format is: `MAJOR.MINOR[.PATCH][-REVISION]`. The version must contain at least major and minor components.

Similarly, to run test cases with a specific BIND9 version, provide it in the BIND9_VERSION variable:

```
$ rake systemtest BIND9_VERSION=9.16
```

Expected version format is: `MAJOR.MINOR`.

6.8.4 System Tests Framework Structure

The system tests framework is located in the tests/system directory that has the following structure:

- `config` - the configuration files for specific docker-compose services
- `core` - implements the system tests logic, docker-compose controller, wrappers for interacting with the services, and integration with `pytest`
- `openapi_client` - an autogenerated client interacting with the Stork Server API
- `test-results` - logs from the last tests execution
- `tests` - the test cases (in the files prefixed with the `test_`)
- `conftest.py` - defines hooks for `pytest`
- `docker-compose.yaml` - the docker-compose services and networking

6.8.5 System Test Structure

Let's consider the following test:

```
from core.wrappers import Server, Kea

def test_search_leases(kea_service: Kea, server_service: Server):
    server_service.log_in_as_admin()
    server_service.authorize_all_machines()

    data = server_service.list_leases('192.0.2.1')
    assert data['items'][0]['ipAddress'] == '192.0.2.1'
```

The system tests framework runs in the background and maintains the docker-compose services that contain different applications. It provides the wrappers to interact with them using a domain language. They are the high-level API and encapsulate the internals of the docker-compose and other applications. The following line:

```
from core.wrappers import Server, Kea
```

imports the typings for these wrappers. Importing them is not necessary to run the test case, but it enables the hints in IDE, which is very convenient during the test development.

The next line:

```
def test_search_leases(kea_service: Kea, server_service: Server):
```

defines the test function. It uses the arguments that are handled by the `pytest` fixtures. There are four fixtures:

- `kea_service` - it starts the container with Kea daemon(s) and Stork Agent. If no fixture argument is specified (see later), it also runs the Stork Server containers and performs the Stork Agents registration. The default configuration is described by the `agent-kea` service in the `docker-compose` file.
- `server_service` - it starts the container with Stork Server. The default configuration is described by the `server` service in the `docker-compose` file.
- `bind9_service` - it starts the container with the Bind9 daemon and Stork Agent. If not fixture argument was used (see later), it runs also the Stork Server containers and Agent registers. The default configuration is described by the `agent-kea` service in the `docker-compose` file.
- `perfdhcp_service` - it provides the container with the `perfdhcp` utility. The default configuration is described by the `perfdhcp` service in the `docker-compose` file.

If the fixture is required, the specified container is automatically built and run. The test case is executed only when the service is operational - it means it is started and healthy (i.e., the health check defined in the Docker image passes). The containers are stopped and removed, and the logs are fetched after the test.

Only one container of a given kind can run in the current version of the system tests framework.

```
server_service.log_in_as_admin()
server_service.authorize_all_machines()
```

Test developers should use the methods provided by the wrappers to interact with the services. Typical operations are already available as functions.

Use `server_service.log_in_as_admin()` to login as an administrator and start the session. Subsequent requests will contain the credentials in the cookie file.

The `server_service.authorize_all_machines()` fetches all unauthorized machines and authorizes them. They are returned by the function. The agent registration is performed during the fixture preparation.

Use the `server_service.wait_for_next_machine_states()` to block and wait until new machine states are fetched and returned.

The server wrapper provides functions to list, search, create, read, update, or delete the items via the REST API without a need to manually prepare the requests and parse the responses. For example:

```
data = server_service.list_leases('192.0.2.1')
```

To verify the data returned by the call above:

```
assert data['items'][0]['ipAddress'] == '192.0.2.1'
```

6.8.6 System Tests with a Custom Service

Test developers should not reconfigure the docker-compose service in a test case for the following reasons.

- It is slow - stopping and re-running the service The test case should assume that the environment is configured.
- It can be unstable - if a service fails to start or is not operational after restart; stopping one service may affect another service. Handling unexpected situations increases the test case duration and increases its complexity.
- It is hard to write and maintain - it is often needed to use regular expressions to modify the content of the existing files, create new files dynamically, and execute the custom commands inside the container. It requires a lot of work, is complex to audit, and is hard to debug.

The definition of the test case environment should be placed in the `docker-compose.yaml` file. Use the environment variables, arguments, and volumes to configure the services. It allows for using static values and files that are easy to read and maintain.

Three general services should be sufficient for most test cases and can be extended in more complex scenarios.

1. **`server-base` - the standard Stork Server. It doesn't use the TLS to** connect to the database.
2. **`agent-kea` - it runs a container with the Stork Agent, Kea DHCPv4, and** Kea DHCPv6 daemons. The agent connects to Kea over IPv4, does not use the TLS or the Basic Auth credentials. Kea is configured to provision 3 IPv4 and 2 IPv6 networks.
3. `agent-bind9` - it runs a container with the Stork Agent and Bind9 daemon.

The services can be customized using the `extends` keyword. The test case should inherit the service configuration and apply suitable modifications.

Note: Test cases should use absolute paths to define the volumes. The host paths should begin with `$PWD` environment variable returning the root project directory.

To run your test case with specific services, use the special helpers:

1. `server_parametrize`
2. `kea_parametrize`
3. `bind9_parametrize`

They accept the name of the docker-compose service to use in the first argument:

```
from core.fixtures import kea_parametrize

@kea_parametrize("agent-kea-many-subnets")
def test_add_kea_with_many_subnets(server_service: Server, kea_service: Kea):
    pass
```

The Kea and Bind9 helpers additionally accept the `suppress_registration` parameter. If it is set to `True` the server service is not automatically started, and the Stork Agent does not try to register.

```
from core.fixtures import kea_parametrize

@kea_parametrize(suppress_registration=True)
def test_kea_only_fixture(kea_service: Kea):
    pass
```

Note: It is not supported to test Stork with different Kea or Bind9 versions. This feature is under construction.

6.8.7 Update Packages in System Tests

A specialized `package_service` docker-compose service is dedicated to performing system tests related to updating the packages. The service contains the Stork Server and Stork Agent (without any Kea or Bind daemons) installed from the CloudSmith packages (instead of the source code).

The installed version can be customized using an `package_parametrize` decorator. If not provided, then the latest version will be installed. Using many different Stork versions in the system tests may impact their execution time.

Additionally, the OpenAPI client is generated from the current Stork version and maybe be incompatible with the older ones. It is possible to use the `no_validate` context to suppress some compatibility errors.

```
with package_service.no_validate() as legacy_service:
    pass
```

6.8.8 Using perfdhcp to Generate Traffic

The `agent-kea` service includes an initialized lease database. It should be enough for most test cases. To generate some DHCP traffic, use the `perfdhcp_service`.

```
from core.wrappers import Kea, Perfdhcp

def test_get_kea_stats(kea_service: Kea, perfdhcp_service: Perfdhcp):
    perfdhcp_service.generate_ipv4_traffic(
        ip_address=kea_service.get_internal_ip_address("subnet_00", family=4),
        mac_prefix="00:00"
    )

    perfdhcp_service.generate_ipv6_traffic(
        interface="eth1"
    )
```

Please note above that an IPv4 address is used to send DHCPv4 traffic and an interface name for the DHCPv6 traffic. There is no easy way to recognize which Docker network is connected to which container interface. The system tests use the `priority` property to ensure that the networks are assigned to the consecutive interfaces.

```
networks:
  storknet:
    ipv4_address: 172.20.42.200
    priority: 1000
  subnet_00:
    ipv4_address: 172.100.42.200
    priority: 500
```

In the configuration above, the `storknet` network should be assigned to the `eth0` (the first) interface, and the `subnet_00` network to the `eth1` interface. Our experiments show that this assumption works reliably.

6.8.9 Debugging System Tests

The system test debugging may be performed at different levels. You can debug the test execution itself or connect the debugger to an executable running in the Docker container.

The easiest approach is to attach the debugger to the running `pytest` process. It can be done using the standard `pdb` Python debugger without any custom configuration, as the debugger is running on the same machine as debugged binary. It allows you to break the test execution at any point and inject custom commands or preview the runtime variables.

Another possibility to use the Python debugger is by running the `pytest` executable directly by `pdb`. You need manually call the `rake systemtest:build` to generate all needed artifacts before running tests. It's recommended to pass the `-s` and `-k` flags to `pytest`.

Even if the test execution is stopped on a breakpoint, the Docker containers are still running in the background. You can check their logs using `rake systemtest:logs SERVICE=<service name>` or run the console inside the container by `rake systemtest:shell SERVICE=<service name>` where the `<service name>` is a service name from the `docker-compose.yaml` file (e.g., `agent-kea`). To check the service status in the container console, type `supervisorctl status`. These tools should suffice to troubleshoot most problems with misconfigured Kea or Bind9 daemons.

It is possible to attach the local debugger to the executable running in the Docker container for more complex cases. This possibility is currently implemented only for the Stork Server. To use it, you must be sure that the codebase on a host is the same as on the container. In system tests, the server is started by the `dlv` Go debugger and listens on the 45678 host port. You can use the `rake utils:connect_dbg` command to attach the `gdlv` debugger. It is recommended to attach the Python debugger and stop the test execution first. Then, attach the Golang debugger to the server.

6.8.10 System Test Commands

The following commands run the system tests and help with troubleshooting:

Table 1: Rake tasks for system testing

Rake Tasks	Description
`rake systemtest`	Runs the system tests. Use TEST variable to run a selected test.
`rake systemtest:build`	Build the system test containers.
`rake systemtest:down`	Stops all system test containers and removes them. It also removes all networks, and volumes.
`rake systemtest:list`	Lists the test cases.
`rake systemtest:logs`	Displays the container logs. Use the SERVICE variable to get the logs only for a specific service.
`rake systemtest:perfdhcp`	Low-level access to the perfdhcp command in a container. The Rake-style arguments can be specified to control `perfdhcp`, e.g.: `rake systemtest:perfdhcp[-6,-l,eth1]`.
`rake systemtest:sh`	Low-level access to the docker-compose with all necessary parameters. Use Rake-style arguments, e.g. `rake systemtest:sh[ps]`
`rake systemtest:shell`	Attaches to a shell in a container with provided name by SERVICE variable.
`rake gen:systemtest:swagger`	Generates the system test OpenAPI client.
`rake gen:systemtest:configs`	Generates the configs used by system tests.

6.8.11 Running Tests Alpine Linux

Running system tests on Alpine Linux requires additional setup steps. Alpine uses `libc-musl` instead of `libc`, which causes issues with the *npm* dependency in Stork build scripts. Installing `nodejs` manually using the package manager solves this problem:

```
$ apk add --no-cache nodejs
```

and set the `USE_SYSTEM_NODEJS` environment variable to `true`:

```
$ rake demo:up USE_SYSTEM_NODEJS=true
```

6.9 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 2: Rake tasks for managing development containers

Rake Task	Description
`rake demo:up:kea`	Build and run an `agent-kea` container with a Stork agent and Kea with DHCPv4. Published port is 8888.
`rake demo:up:kea6`	Build and run an `agent-kea6` container with a Stork agent and Kea with DHCPv6. Published port is 8886.
`rake demo:up:kea_ha`	Build and run two containers, `agent-kea-ha1` and `agent-kea-ha2` that are configured to work together in High Availability mode, with Stork agents, and Kea DHCPv4.
`rake demo:up:kea_premium`	Build and run the `agent-kea-premium-one` and `agent-kea-premium-two` containers with Stork agents and Kea DHCPv4 and DHCPv6 servers, with host reservations stored in a database. It requires **premium** features.
`rake demo:up:bind9`	Build and run an `agent-bind9` container with a Stork agent and BIND 9. Published port is 9999.
`rake demo:up:postgres`	Build and run a Postgres container.
`rake demo:up`	Build and run all above containers
`rake demo:down`	Stop and remove all containers and all referenced volumes and networks

Note: It is recommended that these commands be run using a user account without superuser privileges, which may require some previous steps to set up. On most systems, adding the account to the `docker` group should be enough. On most Linux systems, this is done with:

```
$ sudo usermod -aG docker ${user}
```

A restart may be required for the change to take effect.

The Kea and BIND 9 containers connect to the Stork Server container by default. It can be useful for developers to connect them to the locally running server. You can specify the target server using the SERVER_MODE environment variable with one of the values:

- host - Do not run the server in Docker. Use the local one instead (it must be run separately on the host).

- no-server - Do not run the server.
- with-ui - Run the server in Docker with UI.
- without-ui - Run the server in Docker without UI.
- default - Use the default service configuration from the Docker compose file (default).

For example, to connect the agent from the Docker container to the locally running Stork Server:

1. Run the Stork Server locally:

```
$ rake run:server
```

2. Run a specific agent service with the SERVER_MODE parameter set to `host`:

```
$ rake demo:up:kea SERVER_MODE=host
```

3. Check the unauthorized machines page for a new machine

The Stork Agent containers use the Docker hostnames during communication with Stork Server. If you use the server running locally, located on the Docker host, it cannot resolve the Docker hostnames. You need to explicitly provide the hostname mapping in your `/etc/hosts` file to fix it. You can use the `rake demo:check_etchosts` command to check your actual `/etc/hosts` and generate the content that needs to be appended. This task will automatically run if you use `SERVER_MODE=host` then you don't need to call it manually. It's mainly for diagnostic purposes.

6.10 Packaging

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

The package type is selected based on the OS that executes the command. Use the `utils:print_pkg_type` to get the package type supported by your OS.

Use `rake build:agent_pkg` to build the agent package and `rake build:server_pkg` for server package. The package binaries are located in the `dist/pkgs` directory.

Stork build system attempts to detect native package format. If multiple tools are present, e.g., deb and rpm tools on a Debian-based system, a specific packaging format can be enforced using the PKG_TYPE variable. The available package types will be prompted on a console.

Internally, these packages are built by FPM. It is installed automatically, but it requires the `ruby-dev`, `gnutar`, and `make` to build.

6.11 Storybook

Stork build system has integrated Storybook for development purposes.

> "Storybook is a tool for UI development. It makes development faster and easier by isolating components. This allows you to work on one component at a time. You can develop entire UIs without needing to start up a complex dev stack, force certain data into your database, or navigate around your application."
>
> —Storybook documentation

To run Storybook, type:

```
$ rake storybook
```

and wait for opening a web browser.

6.11.1 Writing a Story

To create a new story for a component, create a new file with the `.stories.ts` extension in the component's directory. It must begin with the story metadata:

```
export default {
    title: 'App/JSON-Tree',
    component: JsonTreeComponent,
    decorators: [
        moduleMetadata({
            imports: [PaginatorModule],
            declarations: [JsonTreeComponent],
        }),
    ],
    argTypes: {
        value: { control: 'object' },
        customValueTemplates: { defaultValue: {} },
        secretKeys: { control: 'object', defaultValue: ['password', 'secret'] },
    },
} as Meta
```

It specifies a title and the main component of the story. The declaration of the `moduleMetadata` decorator is the key part of the file. It contains all related modules, components, and services. It should have similar content to the dictionary passed to the `TestBed.configureTestingModule` in a `.spec.ts` file. The `imports` list should contain all used PrimeNG modules (including these from the sub-components) and Angular modules. Unlike in unit tests, you can use the standard Angular modules instead of the testing modules. Especially:

- `HttpClientModule` instead of `HttpClientTestingModule` to work with the HTTP mocks.
- `BrowserAnimationsModule` instead of `NoopAnimationsModule` to enable animations.

The `declarations` list should contain all Stork-owned components used in the story. The `providers` list should contain all needed services.

Note: There are different ways to mock the services communicating over the REST API, but the easiest and most convenient is simply to mock the actual HTTP calls.

If your component accepts the arguments, specify them using the `argTypes`. It allows you to change their values from the Storybook UI.

Warning: Storybook can discover the component's properties automatically but this feature is currently disabled due to the bug in Storybook for Angular.

Next, create the template object instance by passing the component type as generic type:

```
const Template: Story<JsonTreeComponent> = (args: JsonTreeComponent) => ({
    props: args,
})
```

Finally, bind the template object and provide its arguments:

```
export const Basic = Template.bind({})
```

(continues on next page)

(continued from previous page)

```
Basic.args = {
    key: 'key',
    value: {
        foo: 42
    }
}
```

6.11.2 HTTP Mocks

The easiest way to mock the REST API is using the storybook-addon-mock First, you need to import it:

```
import mockAddon from 'storybook-addon-mock'
```

and append to the `decorators` list in the story metadata:

```
export default {
    title: ...,
    component: ...,
    decorators: [
        moduleMetadata({
            ...
        }),
        mockAddon
    ],
    argTypes: ...,
    parameters: {
        mockData: [
            ...
        ],
    },
} as Meta
```

The mocked API responses are specified by the `parameters.mockData` list that is a property of the metadata object.

Note: Remember to use `HttpClientModule` instead of `HttpTestingClientModule` in the `imports` list of the `moduleMetadata` decorator.

6.11.3 Toast messages

The Stork components often use `MessageService` to present temporary messages to the user. The messages are passed into a dedicated, top-level component responsible for displaying them as temporary rectangles (so-called toasts) in the upper right corner. Due to this, the top-level component is associated with no particular component and does not exist in the isolated Storybook environment. As a result, the toasts are not presented.

To workaround this problem, the `toastDecorator` can be used. It injects additional HTML while rendering the Story. The extra code contains the top-level component to handle toasts and ensures they are correctly displayed.

First, you need to import the decorator:

```
import { toastDecorator } from '../utils.stories'
```

and append it to the `decorators` property of the metadata object:

```
export default {
    title: ...,
    component: ...,
    decorators: [
        moduleMetadata({
            ...
        }),
        toastDecorator
    ],
    argTypes: ...
} as Meta
```

Remember to add the `MessageService` to the `providers` list of the `moduleMetadata` decorator.

6.12 Implementation details

6.12.1 Agent Registration Process

The diagram below shows a flowchart of the agent registration process in Stork. It merely demonstrates the successful registration path. The first Certificate Signing Request (CSR) is generated using an existing or new private key and agent token. The CSR, server token (optional), and agent token are sent to the Stork server. A successful server response contains a signed agent certificate, a server CA certificate, and an assigned Machine ID. If the agent was already registered with the provided agent token, only the assigned machine ID is returned without new certificates. The agent uses the returned machine ID to verify that the registration was successful.

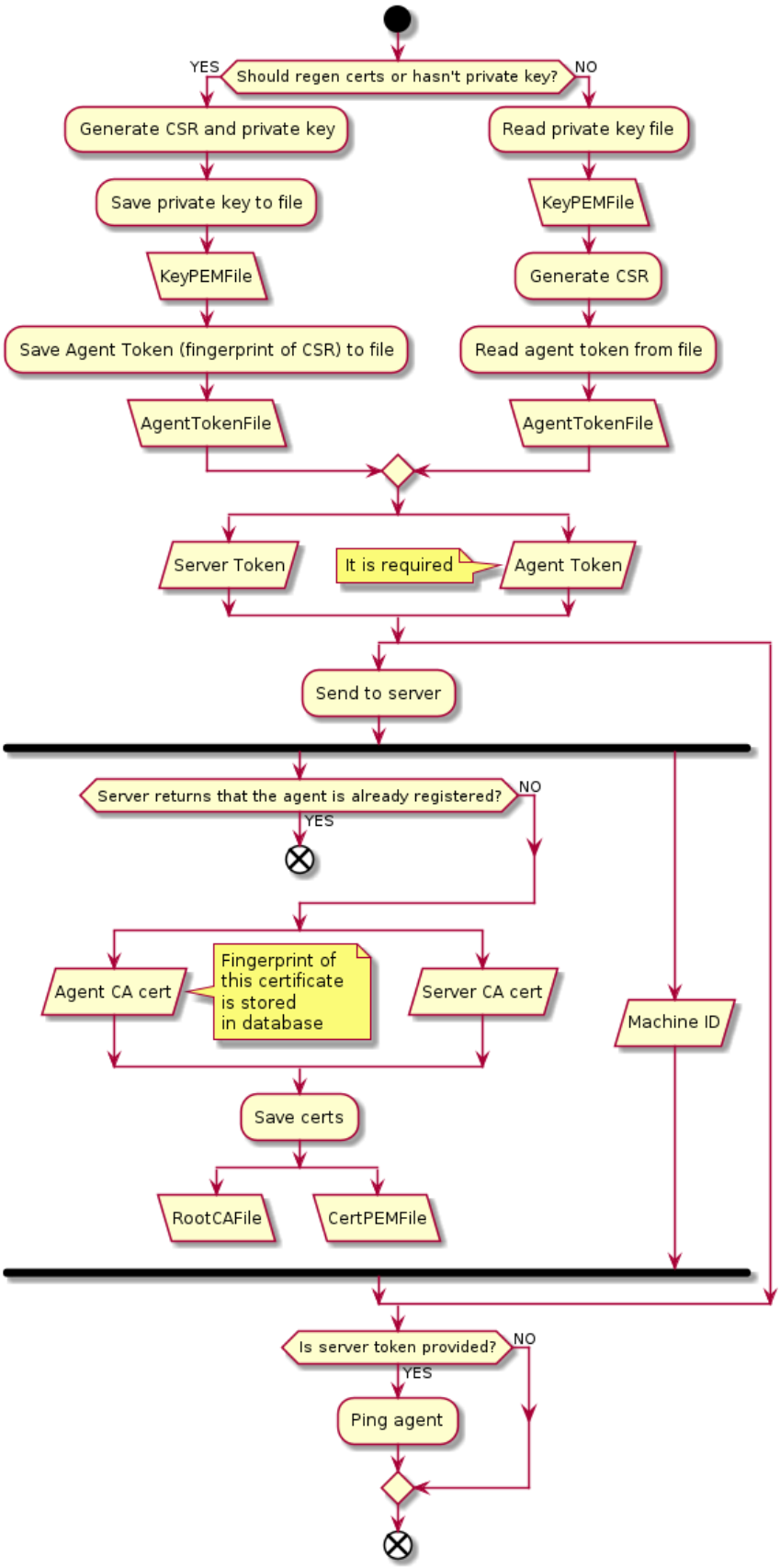


Fig. 1: Agent registration

CHAPTER

SEVEN

DEMO

A sample installation of Stork can be used to demonstrate its 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 Kea Using Many Subnets
- Stork Agent with BIND 9
- Stork Agent with BIND 9-2
- Stork Environment Simulator
- PostgreSQL database
- Prometheus & Grafana

These services allow observation of many Stork features.

7.1 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

7.2 Setup Steps

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

```
$ rake demo:up
```

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

The `stork-demo.sh` script can be used to start the demo without the Ruby toolkit:

```
$ ./stork-demo.sh
```

7.2.1 Premium Features

It is possible to run the demo with premium features enabled in the Kea apps; it requires the demo to be started with an access token for the Kea premium repositories. Access tokens are provided to ISC's paid support customers and format-specific setup instructions can be found on https://cloudsmith.io/~isc/repos/kea-2-0-prv/setup/#tab-formats-deb. ISC paid support customers should feel free to open a ticket for assistance if needed.

```
$ rake demo:up CS_REPO_ACCESS_TOKEN=<access token>
```

7.2.2 Detached Mode

It is possible to start the demo in detached mode. In that case, it does not depend on the terminal and runs in the background until the `rake demo:down` call. To enable the detached mode, specify the DETACH variable set to `true`.

```
$ rake demo:up DETACH=true
```

7.3 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 `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 instances 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 (nearly 7000) subnets defined in its configuration.

agent-kea-premium-one and agent-kea-premium-two These containers run agents with Kea DHCPv4 and DHCPv6 servers connected to a MySQL database containing host reservations. They are only available when premium features have been enabled during the demo build.

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

postgres This container is essential. It runs the PostgreSQL database that is used by `stork-server` and the Kea containers. Without it, `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 the 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.

mariadb This container is essential. It runs the MariaDB database that is used by the Kea containers.

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).

dns-proxy-server Used only when the Stork Agent from container connects to a locally running server. The Kea/Bind containers use internal Docker hostnames that the host cannot resolve. We run the DNS proxy in the background that translates the Docker hostnames to valid IP addresses.

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.

7.4 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

7.5 Stork Environment Simulator

The Stork Environment Simulator demonstrates how Stork:

- sends DHCP traffic to Kea applications
- sends DNS requests to BIND 9 applications
- stops and starts Stork agents and the Kea and BIND 9 daemons

The 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.

The Stork Environment Simulator also allows demonstration DNS traffic to be sent selected DNS servers.

The 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 port 5000 when the demo is running.

7.6 Prometheus

The Prometheus instance is preconfigured in the Stork demo and pulls statistics from:

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

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

7.7 Grafana

The Grafana instance is also preconfigured in the Stork demo. 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

EIGHT

MANUAL PAGES

8.1 stork-server - Main Stork Server

8.1.1 Synopsis

stork-server [**-h**] [**-v**] [**-m**] [**-u**] [**–dbhost**] [**-p**] [**-d**] [**–db-sslmode**] [**–db-sslcert**] [**–db-sslkey**] [**–db-sslrootcert**] [**–db-trace-queries=**] [**–rest-cleanup-timeout**] [**–rest-graceful-timeout**] [**–rest-max-header-size**] [**–rest-host**] [**–rest-port**] [**–rest-listen-limit**] [**–rest-keep-alive**] [**–rest-read-timeout**] [**–rest-write-timeout**] [**–rest-tls-certificate**] [**–rest-tls-key**] [**–rest-tls-ca**] [**–rest-static-files-dir**]

8.1.2 Description

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

8.1.3 Arguments

`stork-server` takes the following arguments (equivalent environment variables are listed in square brackets, where applicable):

-h|--help Returns the list of available parameters.

-v|--version Returns the `stork-server` version.

-m|--metrics Enables the periodic metrics collector and /metrics HTTP endpoint for Prometheus. This endpoint requires no authentication; it is recommended to restrict external access to it (e.g. using the HTTP proxy). It is disabled by default. `[$STORK_SERVER_ENABLE_METRICS]`

--initial-puller-interval Default interval used by pullers fetching data from Kea. If not provided the recommended values for each puller are used. `[$STORK_SERVER_INITIAL_PULLER_INTERVAL]`

-u|--db-user Specifies the user name to be used for database connections. The default is `stork`. `[$STORK_DATABASE_USER_NAME]`

--db-host Specifies the name of the host where the database is available. The default is `localhost`. `[$STORK_DATABASE_HOST]`

-p|--db-port Specifies the port on which the database is available. The default is 5432. `[$STORK_DATABASE_PORT]`

-d|--db-name= Specifies the name of the database to connect to. The default is `stork`. `[$STORK_DATABASE_NAME]`

--db-sslmode Specifies the SSL mode for connecting to the database; possible values are `disable`, `require`, `verify-ca` or `verify-full`. The default is `disable`. [$STORK_DATABASE_SSLMODE] Acceptable values are:

`disable` Disables encryption between the Stork server and the PostgreSQL database.

`require` Uses secure communication but does not verify the server's identity, unless the root certificate location is specified and that certificate exists. If the root certificate exists, the behavior is the same as in the case of `verify-ca`.

`verify-ca` Uses secure communication and verifies the server's identity by checking it against the root certificate stored on the Stork server machine.

`verify-full` Uses secure communication and verifies the server's identity against the root certificate. In addition, checks that the server hostname matches the name stored in the certificate.

--db-sslcert Specifies the location of the SSL certificate used by the server to connect to the database. [$STORK_DATABASE_SSLCERT]

--db-sslkey Specifies the location of the SSL key used by the server to connect to the database. [$STORK_DATABASE_SSLKEY]

--db-sslrootcert Specifies the location of the root certificate file used to verify the database server's certificate. [$STORK_DATABASE_SSLROOTCERT]

--db-trace-queries= Enables tracing of SQL queries. Possible values are `run` - only runtime, without migrations, or `all` - both migrations and runtime. [$STORK_DATABASE_TRACE]

--rest-cleanup-timeout Specifies the period to wait, in seconds, before killing idle connections. The default is 10.

--rest-graceful-timeout Specifies the period to wait, in seconds, before shutting down the server. The default is 15.

--rest-max-header-size Specifies the maximum number of bytes the server reads when parsing the request header's keys and values, including the request line. It does not limit the size of the request body. The default is 1024 (1MB).

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

--rest-port Specifies the port to listen on for connections over the RESTful API. The default is 8080. [$STORK_REST_PORT]

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

--rest-keep-alive Specifies the TCP keep-alive timeout, in minutes, on accepted connections. After this period, the server prunes dead TCP connections (e.g. when a laptop is closed mid-download). The default is 3.

--rest-read-timeout Specifies the maximum duration, in seconds, before timing out a read of the request. The default is 30.

--rest-write-timeout Specifies the maximum duration, in seconds, before timing out a write of the response. The default is 60.

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

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

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

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

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.

To control the logging colorization, Stork supports the `CLICOLOR` and `CLICOLOR_FORCE` standard UNIX environment variables. Use `CLICOLOR_FORCE` to enforce enabling or disabling the ANSI colors usage. Set `CLICOLOR` to `0` or `false` to disable colorization even if the TTY is attached.

8.1.4 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://www.isc.org/mailinglists/. The community provides best-effort support on both of those lists.

8.1.5 History

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

8.1.6 See Also

stork-agent(8)

8.2 `stork-agent` - Stork Agent to Monitor BIND 9 and Kea services

8.2.1 Synopsis

stork-agent [**–listen-stork-only**] [**–listen-prometheus-only**] [**-v**] [**–host=**] [**–port=**] [**–skip-tls-cert-verification=**] [**–prometheus-kea-exporter-address=**] [**–prometheus-kea-exporter-port=**] [**–prometheus-kea-exporter-interval=**] [**-h**]

8.2.2 Description

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

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

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

`-v|--version` Returns the software version.

Stork server flags:

--host= Specifies the IP address or hostname to listen on for incoming Stork server connections. [$STORK_AGENT_HOST]

--port= Specifies the TCP port to listen on for incoming Stork server connections. The default is 8080. [$STORK_AGENT_PORT]

--skip-tls-cert-verification= Indicates that TLS certificate verification should be skipped when the Stork agent connects to Kea over TLS and Kea uses self-signed certificates. The default is false. [$STORK_AGENT_SKIP_TLS_CERT_VERIFICATION]

Prometheus Kea Exporter flags:

--prometheus-kea-exporter-address= Specifies the IP address or hostname on which the agent exports Kea statistics to Prometheus. The default is 0.0.0.0. [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_ADDRESS]

--prometheus-kea-exporter-port= Specifies the port on which the agent exports Kea statistics to Prometheus. The default is 9547. [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_PORT]

--prometheus-kea-exporter-interval= Specifies how often the agent collects statistics from Kea, in seconds. The default is 10. [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_INTERVAL]

--prometheus-kea-exporter-per-subnet-stats= enable or disable collecting per subnet stats from Kea; (default: true) [$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_PER_SUBNET_STATS]

Prometheus BIND 9 Exporter flags:

--prometheus-bind9-exporter-address= Specifies the IP address or hostname on which the agent exports BIND 9 statistics to Prometheus. The default is 0.0.0.0. [$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_ADDRESS]

--prometheus-bind9-exporter-port= Specifies the port on which the agent exports BIND 9 statistics to Prometheus. The default is 9119. [$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_PORT]

--prometheus-bind9-exporter-interval= Specifies how often the agent collects statistics from BIND 9, in seconds. The default is 10. [$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_INTERVAL]

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

To control the logging colorization, Stork supports the CLICOLOR and CLICOLOR_FORCE standard UNIX environment variables. Use CLICOLOR_FORCE to enforce enabling or disabling the ANSI colors usage. Set CLICOLOR to 0 or false to disable colorization even if the TTY is attached.

8.2.4 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://www.isc.org/mailinglists. The community provides best-effort support on both of those lists.

8.2.5 History

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

8.2.6 See Also

stork-server(8)

8.3 `stork-tool` - A Tool for Managing Stork Server

8.3.1 Synopsis

`stork-tool` [**global options**] command [**command options**]

8.3.2 Description

`stork-tool` provides three features:

- Certificate management - it allows the Stork server to export keys, certificates and tokens that are used to secure communication between Stork server and Stork agents.
- Database Creation - it facilitates creating a new database for the Stork Server, and a user that can access this database with a generated password
- Database migration - it allows database schema migrations to be performed, overwriting the database schema version and getting its current value. There is normally no need to use this, as the Stork server always runs the migration scripts on startup.

8.3.3 Certificate Management

`stork-tool` takes the following arguments (equivalent environment variables are listed in square brackets, where applicable):

- `cert-export` Exports a certificate or other secret data. The options are:

  `-f|--object=` Specifies the object to dump, which can be one of `cakey`, `cacert`, `srvkey`, `srvcert`, or `srvtkn`. `[$STORK_TOOL_CERT_OBJECT]`

  `-o|--file=` Specifies the location of the file where the object should be saved. `[$STORK_TOOL_CERT_FILE]`

  To print the Certificate Authority key in the console:

```
$ stork-tool cert-export --db-url postgresql://user:pass@localhost/dbname -f cakey
INFO[2021-05-25 12:36:07]       connection.go:59    checking connection to database
INFO[2021-05-25 12:36:07]            certs.go:225   CA key:
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQghrTv9SVZ/hv0xSM+
jvUk+VehIcf1tD/yMfAF4IiVXaahRANCAATgene6dVwo1xCmYjMKYxSrxgOWRm2G
R5X1x72axq2cAhCFm7EpD88oYZ3EBdoXmG9fihV5ZGtfFkSpIdzCNPQI
-----END PRIVATE KEY-----
```

  To export the server certificate to a file:

```
$ stork-tool cert-export --db-url postgresql://user:pass@localhost/dbname -f␣
↪srvcert -o srv-cert.pem
INFO[2021-05-25 12:36:46]       connection.go:59    checking connection to database
INFO[2021-05-25 12:36:46]            certs.go:221   server cert saved to file: srv-
↪cert.pem
```

- `cert-import` Imports a certificate or other secret data. The options are:

  -f|--object= Specifies the object to dump, which can be one of `cakey`, `cacert`, `srvkey`, `srvcert`, or `srvtkn`. [$STORK_TOOL_CERT_OBJECT]

  -i, --file= Specifies the location of the file from which the object is loaded. [$STORK_TOOL_CERT_FILE]

  To read the server token from stdin:

```
$ echo abc | stork-tool cert-import --db-url postgresql://user:pass@localhost/
↪dbname -f srvtkn
INFO[2021-08-11 13:31:55]       connection.go:59    checking connection to database
INFO[2021-08-11 13:31:55]            certs.go:259   reading server token from stdin
INFO[2021-08-11 13:31:55]            certs.go:261   server token read from stdin,␣
↪length 4
```

  To import the server certificate from a file:

```
$ stork-tool cert-import --db-url postgresql://user:pass@localhost/dbname -f␣
↪srvcert -i srv.cert
INFO[2021-08-11 15:22:28]       connection.go:59    checking connection to database
INFO[2021-08-11 15:22:28]            certs.go:257   server cert loaded from srv.
↪cert file, length 14
```

8.3.4 Database Creation

`stork-tool` offers the following commands for creating the database for the Stork Server:

- `db-create` Create new database
- `db-password-gen` Generate random database password

Options specific to `db-create` command:

-m, --db-maintenance-name existing maintenance database name. (default: "postgres") [$STORK_DATABASE_MAINTENANCE_NAME]

-a, --db-maintenance-user database administrator user name. (default: "postgres") [$STORK_DATABASE_MAINTENANCE_USER_NAME]

--db-maintenance-password database administrator password; if not specified, the user will be prompted for the password.

-f, --force recreate the database and the user if they exist. (default false)

8.3.4.1 Examples

Create a new database `stork` with user `stork` and a generated password:

```
$ stork-tool db-create --db-maintenance-user postgres --db-name stork --db-user stork
INFO[2022-01-25 17:04:56]             main.go:145   created database and user for the␣
↪server with the following credentials  database_name=stork␣
↪password=L82B+kJEOyhDoMnZf9qPAGyKjH5Qo/Xb user=stork
```

When a database is created using `psql` tool, it is sometimes useful to generate a hard-to-guess password for this database:

```
$ stork-tool db-password-gen
INFO[2022-01-25 17:56:31]             main.go:157   generated new database password
↪         password=znYDfWzvMhWRZyJJuu3EvUxH5KMi1SmJ
```

8.3.5 Database Migration

`stork-tool` offers the following commands:

- `db-init` Creates a schema versioning table in the database.
- `db-up` Runs all available migrations; use `-t` to migrate to a specific version.
- `db-down` Reverts the last migration; use `-t` to migrate to a specific version.
- `db-reset` Reverts all migrations.
- `db-version` Prints the current migration version.
- `db-set-version` Sets the database version without running migrations.

  The following option is specific to the `db-up`, `db-down`, and `db-set-version` commands:

  -t|--version= Specifies the target database schema version. The default is `stork`. `[$STORK_TOOL_DB_VERSION]`

To initialize a database schema:

```
$ STORK_DATABASE_PASSWORD=pass stork-tool db-init -u user -d dbname
INFO[2021-05-25 12:30:53]       connection.go:59    checking connection to database
INFO[2021-05-25 12:30:53]             main.go:100   Database version is 0 (new version
↪33 available)
```

To overwrite the current schema version to an arbitrary value:

```
$ STORK_DATABASE_PASSWORD=pass stork-tool db-set-version -u user -d dbname -t 42
INFO[2021-05-25 12:31:30]             main.go:77    Requested setting version to 42
INFO[2021-05-25 12:31:30]       connection.go:59    checking connection to database
INFO[2021-05-25 12:31:30]             main.go:94    Migrated database from version 0 to
↪42
```

8.3.6 Common Options

The following options pertain to both `db-` and `cert-` commands:

--db-url= Specifies the URL for the Stork PostgreSQL database. `[$STORK_DATABASE_URL]`

-u|--db-user= Specifies the user name for database connections. The default is `stork`. `[$STORK_DATABASE_USER_NAME]`

--db-password= Specifies the database password for database connections. `[$STORK_DATABASE_PASSWORD]`

--db-host= Specifies the name of the host where the database is available. The default is `localhost`. `[$STORK_DATABASE_HOST]`

-p|--db-port= Specifies the port on which the database is available. The default is 5432. `[$STORK_DATABASE_PORT]`

-d|--db-name= Specifies the name of the database to connect to. The default is `stork`. `[$STORK_DATABASE_NAME]`

--db-sslmode Specifies the SSL mode for connecting to the database; possible values are `disable`, `require`, `verify-ca`, or `verify-full`. The default is `disable`. [$STORK_DATABASE_SSLMODE] Acceptable values are:

`disable` Disables encryption between the Stork server and the PostgreSQL database.

`require` Uses secure communication but does not verify the server's identity, unless the root certificate location is specified and that certificate exists. If the root certificate exists, the behavior is the same as in the case of `verify-ca`.

`verify-ca` Uses secure communication and verifies the server's identity by checking it against the root certificate stored on the Stork server machine.

`verify-full` Uses secure communication and verifies the server's identity against the root certificate. In addition, checks that the server hostname matches the name stored in the certificate.

--db-sslcert Specifies the location of the SSL certificate used by the server to connect to the database. [$STORK_DATABASE_SSLCERT]

--db-sslkey Specifies the location of the SSL key used by the server to connect to the database. [$STORK_DATABASE_SSLKEY]

--db-sslrootcert Specifies the location of the root certificate file used to verify the database server's certificate. [$STORK_DATABASE_SSLROOTCERT]

--db-trace-queries= Enables tracing of SQL queries. Possible values are `run` - only runtime, without migrations, or `all` - both migrations and runtime. [$STORK_DATABASE_TRACE_QUERIES]

-h|--help Shows a 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.

To control the logging colorization, Stork supports the `CLICOLOR` and `CLICOLOR_FORCE` standard UNIX environment variables. Use `CLICOLOR_FORCE` to enforce enabling or disabling the ANSI colors usage. Set `CLICOLOR` to `0` or `false` to disable colorization even if the TTY is attached.

8.3.7 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://www.isc.org/mailinglists. The community provides best-effort support on both of those lists.

8.3.8 History

`stork-tool` was first coded in October 2019 by Marcin Siodelski; at that time it was called `stork-db-migrate`. In 2021, it was refactored as `stork-tool` and commands for Certificate Management were added by Michal Nowikowski.

8.3.9 See Also

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