Version 0.3.x

CONFIGURING AN RVI NODE

This document describes the process of configuring an RVI node so that it can serve locally connected services, and also find other RVI nodes in a network.

READER ASSUMPTIONS

The reader is assumed to be able to:

1. Have a basic understanding of Linux directory structures.
2. Start and stop programs on the RVI-hosting system
3. Edit configuration files.
4. Understand the basic concepts of IP addresses, ports and URLs.

PREREQUISITES

1. Erlang runtime R16B01 or later has to be installed on the hosting system.
2. The `setup_rvi_node.sh` tool is available to build a release.
3. `rvi_sample.config` is used as a starting point for a customized setup. Root access is not needed.

CONFIGURATION PROCESS OVERVIEW

To bring up an RVI node so that it can be used by locally connected services and communicate with other RVI nodes, the following steps must be taken.

1. Specify the node service prefix
This node will handle traffic to all services that start with the given prefix.

2. Specify RVI node external address
The external address is announced by the Data Link component to other RVI nodes, allowing them to connect to this node and exchange services.

2. Configure static nodes
Backend / Cloud-based RVI nodes have non-changing network addresses that should be known by other nodes in a network. This is acheived by setting up service prefixes and addresses of the static nodes in all other nodes

deployed in a network.

3. Specify Service Edge URL that local services connect to
The Service Edge URL is used by local services to send traffic that is to be forwarded to services on the local and remote nodes.

4. Specify URLs for RVI components
In addition to the Service Edge URL, the remaining components must have their URLs configured so that the components can locate each other and exchange commands.

5. Build the development release
The `setup_rvi_node.sh` is executed to read the configuration file and generate a development or production release.

6. Start the release
The `rvi_node.sh` is executed to launch the built development release. `$REL_HOME/rvi/bin/rvi start` is used to launch the production release.

CONFIGURATION FILE LOCATION AND FORMATS

There is a single configuration file, with the setup for all components and modules in the node, used for each release. A documented example file is provided as `rvi_sample.config`

The configuration file consists of an array of erlang tuples (records / structs / entries), where the `env` tuple contains configuration data for all components. The `rvi` tuple under `env` has all the configuration data for the RVI system. With the possible exception for the lager logging system, only the `rvi` tuple needs to be edited.

The term tuple and entry will be intermixed throughout this document.

SPECIFY NODE SERVICE PREFIX

All RVI nodes hosting locally connected services will announce these services toward other, external RVI nodes as a part of the service discovery mechanism. When announcing its local services to external RVI nodes, a node will prefix each service with a static string that is system-wide unique.

When a service sends traffic to another service, the local RVI node will prefix match the name of the destination service against the service prefix of all known nodes in the system. The node with the longest matching prefix will receive the traffic in order to have it forwarded to the targeted service that is connected to it.

The prefix always starts with an organisational ID that identifies the entity that manages the service. Best practises is to use the domain name of the hosting organisation.

Since every node's service prefix must be unique, they often contain a network address, a device id, a phone number, or similar device-unqiue information. Backend / Cloud nodes often have a symbolic, and unique prefix identifying what their role is.

Below are a few examples of prefixes:

`jaguarlandrover.com/vin/sajwa71b37sh1839/` - A JLR vehcile with the given vin.

`jaguarlandrover.com/mobile/+19492947872/` - A mobile device with a given number, managed by JLR, hosting an RVI node.

`jaguarlandrover.com/sota/` - JLR's global software over the air server.

`jaguarlandrover.com/3rd_party/` - JLR's 3rd party application portal.

`jaguarlandrover.com/diagnostic/` - JLR's diagnostic server.

The prefix for an RVI node is set in the `node_service_prefix` tuple.

An example entry is given below:

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { node_service_prefix, "jaguarlandrover.com/backend/" }
    ]}
  ]}
]
```

SPECIFY RVI NODE EXTERNAL ADDRESS

The external rvi node address is the address, as seen from the outside world, where this node's data link can be contacted. In IP based networks, this is usually a `hostname:port` value. In SMS-only networks, this will be the MSISDN of the node's mobile subscription. Any traffic directed to the given address should be forwarded to the Data Link component.

If the node lives behind a firewall, or should for some reason not accept incoming connections from other nodes, the node external address should be set to `"0.0.0.0:0"`.

The configuration element to set under the `rvi` tuple is `node_address`.

An example tuple is given below:

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { node_address, "92.52.72.132:8817 }
    ]}
  ]}
]
```

Please note that IP addresses, not DNS names, should be used in all network addresses.

In the default Data Link component, `data_link_bert_rpc`, you also need to specify the port it should listen to, and optionally also the interface to use.

This is done by editing the tuple `rvi -> data_link -> bert_rpc_server`, and set `port` to the port that traffic is recevied on. If `data_link_bert_rpc` is to listen for traffic on only one interface, the IP address can be specified as `ip`.

An example tuple is given below:

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { components, [
        ...
        { data_link, [
         ...
         { bert_rpc_server, [
            { port, 8817 },
            { ip, "192.168.11.234"}
          ]}
        ]}
      ]}
    ]}
  ]}
]
```

If `data_link_bert_rpc` is to listen to the port on all network interfaces, the `ip` tuple can be omitted.

CONFIGURE STATIC NODES

Some RVI nodes in a network, such as central backend servers, will always be available at static network addresses. Services on these static nodes should be made available to all other nodes in a network (given that network connectivity is available).

The service prefixes and network addresses of static nodes can be configured in all other nodes, making the static nodes globally available outside the regular, peer-to-peer service discovery mechanism.

When traffic targeting a remote service is received by the RVI node from a locally connected service, it will first try to locate the remote node hosting the destination service through the service discovery database. If this fails the statically configured nodes are searched, prefix matching the name of the destination service against the specified static nodes' service prefixes.

If there is a match, the request will be sent to the network address of the matching node. If there are multiple matches the static node with the longest matching prefix will receive the traffic.

Static nodes are configured as a list of tuples under the `static_nodes` tuple.

An example entry is gven below:

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { static_nodes, [
        { "jaguarlandrover.com/sota/", "92.52.72.132:8817" },
        { "jaguarlandrover.com/remote_diagnostic/", "92.52.72.132:8818" }
      ]}
    ]}
  ]}
]
```

Please note that IP addresses, not DNS names, should be used in all network addresses.

SPECIFY SERVICE EDGE URL

The Service Edge URL is that which will be used by locally connected services to interact, through JSON-RPC, with the RVI node.

Other components in the RVI node use the same URL to send internal traffic to Service Edge.

The URL of Service Edge is specified through the `service_edge` tuple's `url` entry, read by the other components in the node to locate it. When a URL is specified for Service Edge, the port that it is to listen to must be synchronzied as well, using the `exo\_http\_opts` tuple.

An example entry is gven below:

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { components, [
        ...
        { service_edge, [
          { url, "http://127.0.0.1:8811" },
          { exo_http_opts, [ { port, 8811 } ]}
        ]}
      ]}
    ]}
  ]}
]
```

Please note that IP addresses, not DNS names, should be used in all network addresses.

SPECIFY URLS OF RVI COMPONENTS

The remaining components in an RVI system needs to have their URLs and listening ports setup as well. It is recommended that consecutive ports after that used for `service_edge` are used.

Please note that if only erlang components are used (as is the case in the reference implementation), native erlang gen_server calls can be used instead of URLs, providing a 400% transactional speedup. Please see the genserver components chapter below for details.

Below is an example of a complete port/url configuration for all components, including the `bert_rpc_server` entry described in the external node address chapter:

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { components, [
        ...
        { service_edge, [
          { url, "http://127.0.0.1:8811" },
          { exo_http_opts, [ { port, 8811 } ]}
        ]},
        { service_discovery, [
          { url, "http://127.0.0.1:8812" },
          { exo_http_opts, [ { port, 8812 } ] }
        ]},
        { schedule, [
          { url, "http://127.0.0.1:8813" },
          { exo_http_opts, [ { port, 8813 } ] }
        ]},
        { authorize, [
          { url, "http://127.0.0.1:8814" },
          { exo_http_opts, [ { port, 8814 } ] }
        ]},
        { protocol, [
          { url, "http://127.0.0.1:8815" },
          { exo_http_opts, [ { port, 8815 } ] }
        ]},
        { data_link, [
          { url, "http://127.0.0.1:8816" },
          { exo_http_opts, [ { port, 8816 } ] },
          { bert_rpc_server, [ {port, 8817 } ] }
        ]}
      ]}
    ]}
  ]}
]
```

Please note that IP addresses, not DNS names, should be used in all network addresses.

SPECIFY GEN_SERVER ADDRESSES FOR RVI COMPONENTS

Communication between the RVi components can be either JSON-RPC or gen_server calls.

JSON-RPC calls provide compatability with replacement components written in languages other than Erlang. Gen_server calls provide native erlang inter-process calls that are about 4x faster than JSON-RPC when transmitting large data volumes.

If one or more of the RVI components are replaced with external components, use JSON-RPC by specifying url and exo*http*opts for all components.

If you are running an all-native erlang system, use gen*server calls by configuring gen*server.

If you specify both gen*server and url/exo*http*opts, the gen*server communicaiton path will be used for inter component communication.

Please note that communication between two RVI nodes are not affected by this since data*link*bert_rpc will use BERT-RPC to communicate (using the address/port specified by `bert_rpc_server`).

Below is an example of where gen_server is used where appropriate.

Please note that `service_edge always` need to have its `url` and `exo_http_opts` options specified since local services need an HTTP port to send JSON-RPC to. However, gen*server can still be specified in parallel, allowing for gen*server calls to be made between Servie Edge and other RVI components.

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { components, [
        ...
        { service_edge, [
          { gen_server, service_edge_rpc },
          { url, "http://127.0.0.1:8811" },
          { exo_http_opts, [ { port, 8811 } ]}
        ]},
        { service_discovery, [
          { gen_server, service_discovery_rpc }
        ]},
        { schedule, [
          { gen_server, schedule }
        ]},
        { authorize, [
          { gen_server, authorize_rpc }
        ]},
        { protocol, [
          { gen_server, protocol_rpc }
        ]},
        { data_link, [
          { gen_server, data_link_bert_rpc_rpc },
          { bert_rpc_server, [ {port, 8817 } ] }
        ]}
      ]}
    ]}
  ]}
]
```

SETTING UP WEBSOCKET SUPPORT ON A NODE

The service edge can, optionally, turn on its websocket support in order to support locally connected services written in javascript. This allows an RVI node to host services running in a browser, on node.js or similar web environments. Websocket support is enabled by adding a `websocket` entry to the configuration data of `servide_edge`. Below is the previous configuration example with such a setup.

```
[
  ...
  { env, [
    ...
    { rvi, [
      ...
      { components, [
        ...
        { service_edge, [
          { websocket, [ { port, 8818}]},
          { url, "http://127.0.0.1:8811" },
          { exo_http_opts, [ { port, 8811 } ]}
        ]},
...
```

Websocket clients can now connect to: `ws://1.2.3.4:8818/websession` and issue JSON-RPC commands to Service Edge. (Replace `1.2.3.4` with the IP address of the host).

COMPILING THE RVI SOURCE CODE

Before a development release can be built, the source code needs to be compiled. Please see BUILD.md for details on this process.

CREATING A DEVELOPMENT RELEASE

Please note that a new release must be created each time the configuration file has been updated

Once a configuration file has been completed, a development release is created.

The difference between a development and a production release is that the development release needs the compiled files located in the source tree to operate, while a production release is completely self contained (including the erlang runtime system) in its own subdirectory.

Each release will have a name, which will also be the name of the newly created subdirectory containing the files necessary to start the release.

If a configuration file, `test.config` is to be used when building release `test_rel`, the following command can be run from the build root:

```
./script/setup_rvi_node.sh -d -n test_rel -c test.config
```

Once executed (and no errors were found in test.config), a subdirectory called `test_rel` has been created. This directory contains the erlang configuration and boot files necessary to bring up the RVI node.

STARTING A DEVELOPMENT RELEASE

The newly built development release is started using the `rvi_node.sh` tool.

In order to start the test release, named `test_rel`, created in the previous chapter, the following command is run from the build root:

```
./scripts/rvi_node.sh -n tes_rel
```

When a development release is started the erlang console prompt will be displayed at the end of the startup process, allowing for manual inspection of the running system.

Once the RVI node has been brought up, services can connect to its Service Edge and start routing traffic.

CREATING A PRODUCTION RELEASE

Please note that a new release must be created each time the configuration file has been updated

To create a self contained production release using `prod.config` as the configuration file, and name the release `prod_rel`, the following command can be run from the build root:

```
./script/setup_rvi_node.sh -n prod_rel -c prod.config
```

Once executed (and no errors were found in test.config), a subdirectory called `rel/prod_rel` has been created.

The `prod_rel` directory contains a complete erlang runtime system, the RVI application, and the configuration data generated from `prod.config` the RVI node.

The `prod_rel` directory can be moved to anywhere in the file system, or to another host with the same architecture and OS setup.

STARTING A PRODUCTION RELEASE

The newly built product release is started using the `rel/prod_rel/rvi` tool:

```
./rel/prod_rel/rvi start
```

Stopping is done in a similar manner:

```
./rel/prod_rel/rvi stop
```

To check if a node is up, retrieve its process ID with:

```
./rel/prod_rel/rvi getpid
```

To attach to the console of a started node in order to inspect it run:

```
./rel/prod_rel/rvi attach
```

Note that you need to exit from the console with Ctrl-d. Pressing Ctrl-c will bring down the node itself.

Loggings

To get debug output on a console, start a development release, or attach to a production release, and set the log level manually:

```
1> lager:set_loglevel(lager_console_backend, debug)
```

Replace debug with info, notice, warning, or error for different log levels. A production release will also produce logs to `rel/[release]/log/erlang.log.?`.

Check the file modification date to find which of the log files are currently written to.

You can configure the log level through the lager configuration entry:

```
 {env,
  [
   {lager,
   [ { handlers,
    [ {lager_console_backend, debug} ]
    }
]}
...
```

Additional handlers can also be added for different log destinations.

See Basho's lager documentation at github for details on logging.