Version 0.5.0

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 18.2 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. Provide paths to keys, certificates and service credentials
RVI Core uses X.509 keys certificates for authentication, and credentials specifying which services can be registered and invoked.

3. 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.

4. 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.

5. 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.

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

7. Start the release
The `rvi_node.sh` script 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 `priv/config/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.

Erlang terms

For a full description of Erlang types, read the Erlang Reference Manual. The following is a brief summary:

- Tuples, written as `{ Elem1, ..., ElemN }` are like arrays whose elements are accessed by position.
- Lists, written as `[ Elem1, ..., ElemN ]` are linked lists whose elements are accessed by iterating from the beginning of the list. Another notation is `[ Head | Tail ]`. Strings are actually lists of integers, and `"RVI"` is equivalent to `[82,86,73]`, or `[$R,$V,$I]`.
- Numbers, e.g. `17`, `1.44`, `2#101` (binary notation), `16#5A` (hex notation).
- Atoms, names starting with a lowercase letter or enclosed in single quotes, are essentially labels.
- Variable names start with an uppercase letter.

Setup config files

RVI Core uses the setup tool to build from configuration files.

The files used by `setup` are evaluated using the file:script/1 function, meaning that the file can contain a sequence of executable Erlang expressions, each terminated with a full stop (`.`). Anything starting with `%` and to the end of the line constitutes a comment.

Example from `rvi_core/priv/config/rvi_sample.config`:

```
Env = fun(V, Def) ->
              case os:getenv(V) of
                  false -> Def;
                  Str when is_integer(Def) -> list_to_integer(Str);
                  Str when is_atom(Def) -> list_to_atom(Str);
                  Str -> Str
              end
      end.
IPPort = fun(IP, Port) ->
                 IP ++ ":" ++ integer_to_list(Port)
         end.
MyPort = Env("RVI_PORT", 9000).
MyIP = Env("RVI_MYIP", "127.0.0.1").
MyNodeAddr = Env("RVI_MY_NODE_ADDR", IPPort(MyIP, MyPort)).
BackendIP = Env("RVI_BACKEND", "38.129.64.31").
BackendPort = Env("RVI_BACKEND_PORT", 8807).
LogLevel = Env("RVI_LOGLEVEL", notice).
```

The above defines two function objects used to check for the existence of a given OS environment variable, and using the corresponding value, or else using a default value. For example, the variable `MyPort` is set to either the value of `RVI_PORT` or else to `9000`. These variables are used further down in the configuration file.

The `setup` utility is called when the `rvi.sh` script is run. To e.g. enable debug logging, you can do the following:

```
$ RVI_LOGLEVEL=debug rvi.sh ...
```

`Setup` expects certain configuration entries, e.g. `{apps, [App1, ...]}`, `{env, [{App, [{Key, Val}, ...]}, ...]}`. Most of the configuration work for RVI is done in `{env, ...}`.

It is possible to include existing configuration files and then modifying the result. For example, the `rvi_sample.config` file includes `rvi_core/priv/config/rvi_common.config` via the following line:

```
 {include_lib, "rvi_core/priv/config/rvi_common.config"},
```

Includes can be used at several levels. For example, `priv/test_config/basic_backend.config` includes `priv/test_config/backend.config`, which in its turn includes `priv/config/backend.config`, which, finally, includes `priv/config/rvi_common.config`.

Other examples can be found in `rvi_core/priv/test_config/`, where configurations used by the test suite are located:

```
{ok, CurDir} = file:get_cwd().
[
 {include_lib, "rvi_core/priv/test_config/backend.config"},
 {remove_apps, [bt, dlink_bt]},
 {set_env,
  [{rvi_core,
    [
     {[components, data_link, dlink_tcp_rpc, server_opts, ping_interval], 500}
    ]}
  ]}
].
```

The `set_env` instruction specifically takes a list of `{App, [{Key, Value}]}` tuples, where the `Key` is either an atom or list of atoms, the latter indicating an entry in a 'tree' of entries. In the example above, the tree would look like:

```
{rvi_core,
 [
  ...
  {components,
   [
    ...
    {data_link,
     [
      ...
      {dlink_tcp_rpc,
       [
        ...
        {server_opts,
         [
          ...
          {ping_interval, 500}
         ]}
       ]}
     ]}
   ]}
 ]}
```

If the entry in question exists in the tree, it will be modified; if not, it will be added.

For more details about what can be done with `setup`, see (the setup_gen manual) [https://github.com/uwiger/setup/blob/master/doc/setup_gen.md].

CONFIGURATION FILE VALUE SUBSITUTION

Some forms of substitution are supported by `setup`, see (the docs on variable expansion) [https://github.com/uwiger/setup/blob/master/doc/setup.md#Variable_expansion].

RVI Core supports some additional substitution of its own. All substitution is done automatically when RVI starts, so the running applications see the final results of the substitutions.

All string values under the rvi tuple tree are scanned for specific dokens during startup. If a token is found, it will be replaced with a value referenced by it. Tokens can one of the following:

- `$rvi_file(FileName,Default)` - File content
  When an `$rvi_file()` token is encountered, the first line of the referenced file is read. The line (without the newline) replaces the token.
  Example:
  `{ node_service_prefix, "jlr.com/vin/$rvi_file(/etc/vin,default_vin)"}`

  will be substituted with the first line from the file `/etc/vin`:

  `{ node_service_prefix, "jlr.com/vin/2GKEG25HXP4093669"}`

  If `/etc/vin` cannot be opened, the value `default_vin` will be used instead.

- `$rvi_env(EnvironemtnName,Default)` - Environment variable
  When an `$rvi_env()` token is encountered, the value of the Linux process environment variable (such as $HOME) is read to replace the token.

  Example:
  `{ node_service_prefix, "jlr.com/vin/$rvi_env(VIN,default_vin)"}`

  will be substituted with the value of the `$VIN` environment variable:

```
`{ node_service_prefix, "jlr.com/vin/2GKEG25HXP4093669"}`

If VIN is not a defined environment variable, the value
`default_vin` will be used instead.
```

- `$rvi_uuid(Default)` - Unique machine identifier
  When an `$rvi_uuid()` token is encountered, the UUID of the root disk used by the system is read to replace the token. The UUID of the root disk is retrieved by opening `/proc/cmdline` and extracting the `root=UUID=[DiskUUID]` value. This value is generated at system install time and is reasonably world wide unique.

  Example:
  `{ node_service_prefix, "jlr.com/vin/$uuid(default_vin)"}`

  will be substituted with the value of the root disk UUID: `{ node_service_prefix, "jlr.com/vin/afc0a6d8-0264-4f8a-bb3e-51ff8655b51c"}`

  If the root UUID cannot be retrieved, the value `default_vin` will be used instead.

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

PROVIDE PATHS TO KEYS, CERTIFICATES AND SERVICE CREDENTIALS

The following settings are required for the RVI Core authentication framework:

- `{device_key, DevKeyFile}`
- `{root_cert, RootCertFile}`
- `{device_cert, DevCertFile}`
- `{cred_dir, CredDirectory}`

The doc/rvi_protocol.md document explains the authentication protocol and how to create the necessary keys and certificates.

Note that the `cred_dir` option needs to be a directory. RVI Core will pick up any valid credentials present in that directory.

Default values - the sample keys and certs used in `rvi_protocol.md` - are specified in rvi_common.config. The defaults should *only* be used for testing and demos - never for live use.

CONFIGURE DATA LINK LAYERS

The `data_link` components are specified as `{Module, Type, Options}`, e.g.

```
[
  ...
  { env, [
    ...
    { rvi_core, [
      ...
      { components, [
        ...
        { dlink_tls_rpc, gen_server, [ ... ] }
       ] }
     ] }
   ] }
]
```

In the data link component, `dlink_tls_rpc`, you can specify the following options:

```
{ server_opts, Opts }
```

These are options for the TLS listener and subsequent connections:

- `{port, Port}` - which port to listen to
- `{ip, Address}` - optionally specifies which interface to listen to
- `{max_msg_size, Sz}` - maximum fragment size (see doc/rvi_protocol.md for details
- `{reliable, true | false}` - if `max_msg_size` isn't specified, this will enable reliable transmission using the fragmentation protocol.

  { persistent_connections, Addresses }

`Addresses` is a list of IP:Port strings signifying other RVI nodes to connect to. The connection- and fragmentation-related options above will also be applied to these connections.

An example tuple is given below:

```
[
  ...
  { env, [
    ...
    { rvi_core, [
      ...
      { components, [
        ...
        { data_link, [
         ...
         { dlink_tls_rpc, gen_server,
          [
            ...
            { server_opts, [ { ip, "192.168.11.234"},
                             { port, 8807 },
                             { max_msg_size, 1024 } ]},
            { persistent_connections, [ "38.129.64.13:8807" ]}
          ]
        }]
      }]
    }]
  }]
}
]
```

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

The `persistent_connections` section lists the IP:Port pair of all remote RVI nodes that this node should maintain a connection with. If the address is not available, a reconnection attempt will be made every five seconds.

This allows a solid connection between RVI nodes where only one node can initiate a connection (such as a vehicle-to-server link in a mobile network).

ROUTING RULES

Routing rules determining how to get a message targeting a specific service to its destination.

A routing rule specifies a number of different way to reach an RVI node hosting a specific service prefix, such as `jlr.com/vin/ABCD/sota/`.

Please note that if a remotely initiated (== client) data link is available and has announced that the targeted service is available, that data link will be used regardless of what it is.

Service name prefix that rules are specified for The service prefix with the longest match against the service targeted by the message will be used.

Example: Targeted service = `jlr.com/backend/sota/get_updates`

Prefix 1: `{ "jlr.com/backend", [...]}`
Prefix 2: `{ "jlr.com/backend/sota", [...]}`

In this case, Prefix 2 will be used.

This allows you to setup different servers for different types of services (SOTA, remote door unlock, HVAC etc).

Make sure to have a default routin rule if you don't want your message to error out immediately. With a default the message will be queued until it times out, waiting for a remote node to connect and announce that it can handle the targeted service. Below is an example of a default rule.

```
{ routing_rules, [
  { "", [
    { proto_json_rpc, dlink_tcp_rpc}
  ]}
]}
```

This rule specifies that, unless another rule has a longer prefix match, a request shall be encoded using `proto_json_rpc`, and transmitted using `dlink_tcp_rpc`.

To direct an in-vehicle RVI-node to send all its backend requests to a specific address, add the following rule.

```
{ routing_rules, [
  { "", [
    { proto_json_rpc, dlink_tcp_rpc}
  ]},
  { "jlr.com/backend/",  [
       { proto_json_rpc, { dlink_tcp_rpc, [ { target, "38.129.64.31:8807" } ]}}
  ]}
]}
```

This rule specifies that any message to a service starting with `jlr.com/backend` shall first be encoded using `proto_json_rpc`, and transmitted using `dlink_tcp_rpc`. The `dlink_tcp_rcp` data link module will be instructed to send all messages targeting `jlr.com/backend/...` to the IP-address:port `38.129.64.31:8807`.

To setup Vehicle-to-Vehicle communication, where a vehicle can reach services on other vehicle's starting with `jlr.com/vin/`, add the following rule.

```
{ routing_rules, [
  { "", [
    { proto_json_rpc, dlink_tcp_rpc}
  ]},
  { "jlr.com/backend/",  [
       { proto_json_rpc, { dlink_tcp_rpc, [ { target, "38.129.64.31:8807" } ]}}
  ]},
  { "jlr.com/vin/", [
       { proto_json_rpc, { dlink_tcp_rpc, [ broadcast, { interface, "wlan0" } ] } },
       { proto_json_rpc, { dlink_3g_rpc, [ initiate_outbound ]} },
       { proto_sms_rpc, { dlink_sms_rpc, [ { max_msg_size, 140 } ] } }
      ]
    }
]}
```

This rule specifies that any message to a service starting with `jlr.com/vin` shall first be encoded using the protocol - data link pair `proto_json_rpc - dlink_tcp_rpc`, where WiFi broadcasts shall be used (thrugh `wlan0` and `broadcast`) to find other vehiclces.

If that does not work, a 3G connection to the vehicle shall be attempted, through the `proto_json_rpc - dlink_3g_rpc` pair, where we are allowed to initiate outbound connections to the 3G network in case a connection is not already available.

Finally, an SMS can be sent through the `proto_sms_rpc - dlink_sms_rpc` pair, maximizing message size to 140 bytes.

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.

In cases where JSON-RPC is used instead of Erlang-internal gen_server calls, other components in the RVI node use the same URL to send traffic to Service Edge

The URL of Service Edge is specified through the `service_edge_rpc` tuple's `json_rpc_address` entry, read by the other components in the node to locate it.

An example entry is gven below:

```
[
  ...
  { env, [
    ...
    { rvi_core, [
      ...
      { components, [
        ...
         { service_edge, [
          { service_edge_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8801 } },
            { websocket, [ { port, 8808}]}
          ]}
        ]}
      ]}
    ]}
   ]}
]
```

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_rpc` 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 significant 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_core , [
      ...
      { components, [
        ...

         { service_edge, [
          { service_edge_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8801 } },
            ...
          ]}
        ]},
         { service_discovery, [
          { service_discovery_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8802 }}
          ]}
        ]},
        { schedule, [
          { schedule_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8803 }}
          ]}
        ]},
        { authorize, [
          { authorize_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8804 } }
          ]}
        ]},
        { protocol, [
          { proto_json_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8805 } }
          ]}
        ]},
        { data_link, [
          { dlink_tcp_rpc, json_rpc,  [
            { json_rpc_address, { "127.0.0.1", 8806 } },
            ...
          ]}
        ]}
      ]}
    ]}
  ]}
]
```

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 Erlang-internal gen_server calls. JSON-RPC calls provide compatibility with replacement components written in languages other than Erlang. gen_server calls provide native erlang inter-process calls that are signficantly 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 `json_rpc_address` for all components. If an all-native erlang system is configured, use gen_server calls by configuring `gen_server` . If both `gen_server` and `json_rpc_address` are specified, 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 the protocol and data links specified by the matching routing rule to communicate. See Routing Rules chapter for details. Below is an example of where gen_server is used where

approrpiate. Please note that `service_edge_rpc` always need to have its `json_rpc_address` 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_core , [
      ...
      { components, [
        ...

         { service_edge, [
          { service_edge_rpc, gen_server, [
            { json_rpc_address, { "127.0.0.1", 8801 } },
            ...
          ]}
        ]},
         { service_discovery, [
          { service_discovery_rpc, gen_server, [] }
        ]},
        { schedule, [
          { schedule_rpc, gen_server, [] }
        ]},
        { authorize, [
          { authorize_rpc, gen_server, [] }
        ]},
        { protocol, [
          { proto_json_rpc, gen_server, [] }
        ]},
        { data_link, [
          { dlink_tcp_rpc, gen_server,  [
            ...
          ]}
        ]}

      ]}
    ]}
  ]}
]
```

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_rpc`.

Below is the previous configuration example with such a setup.

```
[
  ...
  { env, [
    ...
    { rvi_core, [
      ...
      { components, [
        ...
         { service_edge, [
          { service_edge_rpc, json_rpc, [
            { json_rpc_address, { "127.0.0.1", 8801 } },
            { websocket, [ { port, 8808}]}
          ]}
        ]}
      ]}
    ]}
  ]}
]
```

Websocket clients can now connect to: `ws://127.0.0.1:8801/websession` and issue JSON-RPC commands to Service Edge. Outbound service invocations, sent from the RVI node to the javascript code, will be transmitted over the same socket.

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, `rvi_sample.config` is to be used when building release `test_rel`, the following command can be run from the build root:

```
./scripts/setup_rvi_node.sh -d -n test_rel -c rvi_sample.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.