Container Network Interface (CNI) Specification

Version

This is CNI **spec** version **1.1.0**.

Note that this is **independent from the version of the CNI library and plugins** in this repository (e.g. the versions of releases).

Released versions

Released versions of the spec are available as Git tags.

tag	spec permalink	major changes
`spec-v1.0.0`	spec at v1.0.0	Removed non-list configurations; removed `version` field of `interfaces` array
`spec-v0.4.0`	spec at v0.4.0	Introduce the CHECK command and passing prevResult on DEL
`spec-v0.3.1`	spec at v0.3.1	none (typo fix only)
`spec-v0.3.0`	spec at v0.3.0	rich result type, plugin chaining
`spec-v0.2.0`	spec at v0.2.0	VERSION command
`spec-v0.1.0`	spec at v0.1.0	initial version

Do not rely on these tags being stable. In the future, we may change our mind about which particular commit is the right marker for a given historical spec version.

Overview

This document proposes a generic plugin-based networking solution for application containers on Linux, the *Container Networking Interface*, or *CNI*.

For the purposes of this proposal, we define four terms very specifically:

- *container* is a network isolation domain, though the actual isolation technology is not defined by the specification. This could be a network namespace or a virtual machine, for example.
- *network* refers to a group of endpoints that are uniquely addressable that can communicate amongst each other. This could be either an individual container (as specified above), a machine, or some other network device (e.g. a router). Containers can be conceptually *added to* or *removed from* one or more networks.
- *runtime* is the program responsible for executing CNI plugins.
- *plugin* is a program that applies a specified network configuration.

This document aims to specify the interface between "runtimes" and "plugins". The key words "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may" and "optional" are used as specified in RFC 2119.

Summary

The CNI specification defines:

1. A format for administrators to define network configuration.
2. A protocol for container runtimes to make requests to network plugins.
3. A procedure for executing plugins based on a supplied configuration.
4. A procedure for plugins to delegate functionality to other plugins.
5. Data types for plugins to return their results to the runtime.

Section 1: Network configuration format

CNI defines a network configuration format for administrators. It contains directives for both the container runtime as well as the plugins to consume. At plugin execution time, this configuration format is interpreted by the runtime and transformed in to a form to be passed to the plugins.

In general, the network configuration is intended to be static. It can conceptually be thought of as being "on disk", though the CNI specification does not actually require this.

Configuration format

A network configuration consists of a JSON object with the following keys:

- `cniVersion` (string): Semantic Version 2.0 of CNI specification to which this configuration list and all the individual configurations conform. Currently "1.1.0"
- `cniVersions` (string list): List of all CNI versions which this configuration supports. See version selection below.
- `name` (string): Network name. This should be unique across all network configurations on a host (or other administrative domain). Must start with an alphanumeric character, optionally followed by any combination of one or more alphanumeric characters, underscore, dot (.) or hyphen (-). Must not contain characters disallowed in file paths.
- `disableCheck` (boolean): Either `true` or `false`. If `disableCheck` is `true`, runtimes must not call `CHECK` for this network configuration list. This allows an administrator

to prevent `CHECK`ing where a combination of plugins is known to return spurious errors.

- `disableGC` (boolean): Either `true` or `false`. If `disableGC` is `true`, runtimes must not call `GC` for this network configuration list. This allows an administrator to prevent `GC`ing when it is known that garbage collection may have undesired effects (e.g. shared configuration between multiple runtimes).
- `loadOnlyInlinedPlugins` (boolean): Either `true` or `false`. If `false` (default), indicates plugin configuration objects can be aggregated from multiple sources. Any valid plugin configuration objects aggregated from other sources must be appended to the final list of `plugins` for that network name. If set to `true`, indicates that valid plugin configuration objects aggregated from sources other than the main network configuration will be ignored. If `plugins` is not present in the network configuration, `loadOnlyInlinedPlugins` cannot be set to `true`.
- `plugins` (list): A list of inlined plugin configuration objects. If this key is populated with inlined plugin objects, and `loadOnlyInlinedPlugins` is true, the final set of plugins for a network must consist of all the plugin objects in this list, merged with all the plugins loaded from the sibling folder with the same name as the network.

Plugin configuration objects:

Runtimes may aggregate plugin configuration objects from multiple sources, and must unambiguously associate each loaded plugin configuration object with a single, valid network configuration. All aggregated plugin configuration objects must be validated, and each plugin with a valid configuration object must be invoked.

Plugin configuration objects may contain additional fields beyond the ones defined here. The runtime MUST pass through these fields, unchanged, to the invoked plugin, as defined in section 3.

Required keys:

- `type` (string): Matches the name of the CNI plugin binary on disk. Must not contain characters disallowed in file paths for the system (e.g. / or \).

Optional keys, used by the protocol:

- `capabilities` (dictionary): Defined in section 3

Reserved keys, used by the protocol: These keys are generated by the runtime at execution time, and thus should not be used in configuration.

- `runtimeConfig`

- `args`
- Any keys starting with `cni.dev/`

Optional keys, well-known: These keys are not used by the protocol, but have a standard meaning to plugins. Plugins that consume any of these configuration keys should respect their intended semantics.

- `ipMasq` (boolean): If supported by the plugin, sets up an IP masquerade on the host for this network. This is necessary if the host will act as a gateway to subnets that are not able to route to the IP assigned to the container.
- `ipam` (dictionary): Dictionary with IPAM (IP Address Management) specific values:
  - `type` (string): Refers to the filename of the IPAM plugin executable. Must not contain characters disallowed in file paths for the system (e.g. / or \).
- `dns` (dictionary, optional): Dictionary with DNS specific values:
  - `nameservers` (list of strings, optional): list of a priority-ordered list of DNS nameservers that this network is aware of. Each entry in the list is a string containing either an IPv4 or an IPv6 address.
  - `domain` (string, optional): the local domain used for short hostname lookups.
  - `search` (list of strings, optional): list of priority ordered search domains for short hostname lookups. Will be preferred over `domain` by most resolvers.
  - `options` (list of strings, optional): list of options that can be passed to the resolver

Other keys: Plugins may define additional fields that they accept and may generate an error if called with unknown fields. Runtimes must preserve unknown fields in plugin configuration objects when transforming for execution.

Example configuration

The following is an example JSON representation of a network configuration `dbnet` with three plugin configurations (`bridge` , `tuning` , and `portmap`).

```
{
  "cniVersion": "1.1.0",
  "cniVersions": ["0.3.1", "0.4.0", "1.0.0", "1.1.0"],
  "name": "dbnet",
  "plugins": [
    {
      "type": "bridge",
      // plugin specific parameters
      "bridge": "cni0",
      "keyA": ["some more", "plugin specific", "configuration"],
```

```
      "ipam": {
        "type": "host-local",
        // ipam specific
        "subnet": "10.1.0.0/16",
        "gateway": "10.1.0.1",
        "routes": [
            {"dst": "0.0.0.0/0"}
        ]
      },
      "dns": {
        "nameservers": [ "10.1.0.1" ]
      }
    },
    {
      "type": "tuning",
      "capabilities": {
        "mac": true
      },
      "sysctl": {
        "net.core.somaxconn": "500"
      }
    },
    {
        "type": "portmap",
        "capabilities": {"portMappings": true}
    }
  ]
}
```

Version considerations

CNI runtimes, plugins, and network configurations may support multiple CNI specification versions independently. Plugins indicate their set of supported versions through the VERSION command, while network configurations indicate their set of supported versions through the `cniVersion` and `cniVersions` fields.

CNI runtimes MUST select the highest supported version from the set of network configuration versions given by the `cniVersion` and `cniVersions` fields. Runtimes MAY consider the set of supported plugin versions as reported by the VERSION command when determining available versions.

The CNI protocol follows Semantic Versioning principles, so the configuration format MUST remain backwards and forwards compatible within major versions.

Section 2: Execution Protocol

Overview

The CNI protocol is based on execution of binaries invoked by the container runtime. CNI defines the protocol between the plugin binary and the runtime.

A CNI plugin is responsible for configuring a container's network interface in some manner. Plugins fall in to two broad categories:

- "Interface" plugins, which create a network interface inside the container and ensure it has connectivity.
- "Chained" plugins, which adjust the configuration of an already-created interface (but may need to create more interfaces to do so).

The runtime passes parameters to the plugin via environment variables and configuration. It supplies configuration via stdin. The plugin returns a result on stdout on success, or an error on stderr if the operation fails. Configuration and results are encoded in JSON.

Parameters define invocation-specific settings, whereas configuration is, with some exceptions, the same for any given network.

The runtime must execute the plugin in the runtime's networking domain. (For most cases, this means the root network namespace / `dom0`).

Parameters

Protocol parameters are passed to the plugins via OS environment variables.

- `CNI_COMMAND`: indicates the desired operation; `ADD`, `DEL`, `CHECK`, `GC`, or `VERSION`.
- `CNI_CONTAINERID`: Container ID. A unique plaintext identifier for a container, allocated by the runtime. Must not be empty. Must start with an alphanumeric character, optionally followed by any combination of one or more alphanumeric characters, underscore (), dot (.) or hyphen (-).
- `CNI_NETNS`: A reference to the container's "isolation domain". If using network namespaces, then a path to the network namespace (e.g. `/run/netns/[nsname]`)
- `CNI_IFNAME`: Name of the interface to create inside the container; if the plugin is unable to use this interface name it must return an error.
- `CNI_ARGS`: Extra arguments passed in by the user at invocation time. Alphanumeric key-value pairs separated by semicolons; for example, "FOO=BAR;ABC=123"

- `CNI_PATH`: List of paths to search for CNI plugin executables. Paths are separated by an OS-specific list separator; for example ':' on Linux and ';' on Windows

Errors

A plugin must exit with a return code of 0 on success, and non-zero on failure. If the plugin encounters an error, it should output an "error" result structure (see below).

CNI operations

CNI defines 5 operations: `ADD`, `DEL`, `CHECK`, `GC`, and `VERSION`. These are passed to the plugin via the `CNI_COMMAND` environment variable.

`ADD`: Add container to network, or apply modifications

A CNI plugin, upon receiving an `ADD` command, should either

- create the interface defined by `CNI_IFNAME` inside the container at `CNI_NETNS`, or
- adjust the configuration of the interface defined by `CNI_IFNAME` inside the container at `CNI_NETNS`.

If the CNI plugin is successful, it must output a result structure (see below) on standard out. If the plugin was supplied a `prevResult` as part of its input configuration, it MUST handle `prevResult` by either passing it through, or modifying it appropriately.

If an interface of the requested name already exists in the container, the CNI plugin MUST return with an error.

A runtime should not call `ADD` twice (without an intervening DEL) for the same `(CNI_CONTAINERID, CNI_IFNAME)` tuple. This implies that a given container ID may be added to a specific network more than once only if each addition is done with a different interface name.

Input:

The runtime will provide a JSON-serialized plugin configuration object (defined below) on standard in.

Required environment parameters:

- `CNI_COMMAND`
- `CNI_CONTAINERID`
- `CNI_NETNS`

- `CNI_IFNAME`

Optional environment parameters:

- `CNI_ARGS`
- `CNI_PATH`

`DEL`: Remove container from network, or un-apply modifications

A CNI plugin, upon receiving a `DEL` command, should either

- delete the interface defined by `CNI_IFNAME` inside the container at `CNI_NETNS`, or
- undo any modifications applied in the plugin's `ADD` functionality

A `prevResult` must be supplied to CNI plugins as part of a `DEL` command. For the first plugin in the `DEL` command plugin chain, this `prevResult` will be the final result of the previous `ADD` command. Plugins should still return without error if `prevResult` is empty for a `DEL` command, however.

`DEL` command invocations are always considered best-effort - plugins should always complete a `DEL` action without error to the fullest extent possible, even if some resources or state are missing. For example, an IPAM plugin should generally release an IP allocation and return success even if the container network namespace no longer exists, unless that network namespace is critical for IPAM management. While DHCP may usually send a 'release' message on the container network interface, since DHCP leases have a lifetime this release action would not be considered critical and no error should be returned if this action fails. For another example, the `bridge` plugin should delegate the DEL action to the IPAM plugin and clean up its own resources even if the container network namespace and/or container network interface no longer exist.

Plugins MUST accept multiple `DEL` calls for the same (`CNI_CONTAINERID`, `CNI_IFNAME`) pair, and return success if the interface in question, or any modifications added, are missing.

Input:

The runtime will provide a JSON-serialized plugin configuration object (defined below) on standard in.

Required environment parameters:

- `CNI_COMMAND`
- `CNI_CONTAINERID`

- `CNI_IFNAME`

Optional environment parameters:

- `CNI_NETNS`
- `CNI_ARGS`
- `CNI_PATH`

`CHECK`: Check container's networking is as expected

`CHECK` is a way for a runtime to probe the status of an existing container.

Plugin considerations:

- The plugin must consult the `prevResult` to determine the expected interfaces and addresses.
- The plugin must allow for a later chained plugin to have modified networking resources, e.g. routes, on `ADD`.
- The plugin should return an error if a resource included in the CNI Result type (interface, address or route) was created by the plugin, and is listed in `prevResult`, but is missing or in an invalid state.
- The plugin should return an error if other resources not tracked in the Result type such as the following are missing or are in an invalid state:
  - Firewall rules
  - Traffic shaping controls
  - IP reservations
  - External dependencies such as a daemon required for connectivity
  - etc.
- The plugin should return an error if it is aware of a condition where the container is generally unreachable.
- The plugin must handle `CHECK` being called immediately after an `ADD`, and therefore should allow a reasonable convergence delay for any asynchronous resources.
- The plugin should call `CHECK` on any delegated (e.g. IPAM) plugins and pass any errors on to its caller.

Runtime considerations:

- A runtime must not call `CHECK` for a container that has not been `ADD`ed, or has been `DEL`eted after its last `ADD`.
- A runtime must not call `CHECK` if `disableCheck` is set to `true` in the configuration.

- A runtime must include a `prevResult` field in the network configuration containing the `Result` of the immediately preceding `ADD` for the container. The runtime may wish to use libcni's support for caching `Result`s.
- A runtime may choose to stop executing `CHECK` for a chain when a plugin returns an error.
- A runtime may execute `CHECK` from immediately after a successful `ADD`, up until the container is `DEL`eted from the network.
- A runtime may assume that a failed `CHECK` means the container is permanently in a misconfigured state.

Input:

The runtime will provide a json-serialized plugin configuration object (defined below) on standard in.

Required environment parameters:

- `CNI_COMMAND`
- `CNI_CONTAINERID`
- `CNI_NETNS`
- `CNI_IFNAME`

Optional environment parameters:

- `CNI_ARGS`
- `CNI_PATH`

All parameters, with the exception of `CNI_PATH`, must be the same as the corresponding `ADD` for this container.

`STATUS`: Check plugin status

`STATUS` is a way for a runtime to determine the readiness of a network plugin.

A plugin must exit with a zero (success) return code if the plugin is ready to service ADD requests. If the plugin knows that it is not able to service ADD requests, it must exit with a non-zero return code and output an error on standard out (see below).

For example, if a plugin relies on an external service or daemon, it should return an error to `STATUS` if that service is unavailable. Likewise, if a plugin has a limited number of resources (e.g. IP addresses, hardware queues), it should return an error if those resources are exhausted and no new `ADD` requests can be serviced.

The following error codes are defined in the context of `STATUS`:

- 50: The plugin is not available (i.e. cannot service `ADD` requests)
- 51: The plugin is not available, and existing containers in the network may have limited connectivity.

Plugin considerations:

- Status is purely informational. A plugin MUST NOT rely on `STATUS` being called.
- Plugins should always expect other CNI operations (like `ADD`, `DEL`, etc) even if `STATUS` returns an error. `STATUS` does not prevent other runtime requests.
- If a plugin relies on a delegated plugin (e.g. IPAM) to service `ADD` requests, it must also execute a `STATUS` request to that plugin when it receives a `STATUS` request for itself. If the delegated plugin return an error result, the executing plugin should return an error result.

Input:

The runtime will provide a json-serialized plugin configuration object (defined below) on standard in.

Optional environment parameters:

- `CNI_PATH`

`VERSION`: probe plugin version support

The plugin should output via standard-out a json-serialized version result object (see below).

Input:

A json-serialized object, with the following key:

- `cniVersion`: The version of the protocol in use.

Required environment parameters:

- `CNI_COMMAND`

`GC`: Clean up any stale resources

The GC command provides a way for runtimes to specify the expected set of attachments to a network. The network plugin may then remove any resources related to attachments that do not exist in this set.

Resources may, for example, include:

- IPAM reservations
- Firewall rules

A plugin SHOULD remove as many stale resources as possible. For example, a plugin should remove any IPAM reservations associated with attachments not in the provided list. The plugin MAY assume that the isolation domain (e.g. network namespace) has been deleted, and thus any resources (e.g. network interfaces) therein have been removed.

Plugins should generally complete a `GC` action without error. If an error is encountered, a plugin should continue; removing as many resources as possible, and report the errors back to the runtime.

Plugins MUST, additionally, forward any GC calls to delegated plugins they are configured to use (see section 4).

The runtime MUST NOT use GC as a substitute for DEL. Plugins may be unable to clean up some resources from GC that they would have been able to clean up from DEL.

Input:

The runtime must provide a JSON-serialized plugin configuration object (defined below) on standard in. It contains an additional key;

- `cni.dev/valid-attachments` (array of objects): The list of **still valid** attachments to this network:
  - `containerID` (string): the value of CNI_CONTAINERID as provided during the CNI ADD operation
  - `ifname` (string): the value of CNI_IFNAME as provided during the CNI ADD operation

Required environment parameters:

- `CNI_COMMAND`
- `CNI_PATH`

Output: No output on success, "error" result structure on error.

Section 3: Execution of Network Configurations

This section describes how a container runtime interprets a network configuration (as defined in section 1) and executes plugins accordingly. A runtime may wish to *add*, *delete*, or *check* a network configuration in a container. This results in a series of plugin `ADD`, `DELETE`, or `CHECK` executions, correspondingly. This section also defines how a network configuration is transformed and provided to the plugin.

The operation of a network configuration on a container is called an *attachment*. An attachment may be uniquely identified by the `(CNI_CONTAINERID, CNI_IFNAME)` tuple.

Lifecycle & Ordering

- The container runtime must create a new network namespace for the container before invoking any plugins.
- The container runtime must not invoke parallel operations for the same container, but is allowed to invoke parallel operations for different containers. This includes across multiple attachments.
    - **Exception**: The runtime must exclusively execute either *gc* or *add* and *delete*. The runtime must ensure that no *add* or *delete* operations are in progress before executing *gc*, and must wait for *gc* to complete before issuing new *add* or *delete* commands.
- Plugins must handle being executed concurrently across different containers. If necessary, they must implement locking on shared resources (e.g. IPAM databases).
- The container runtime must ensure that *add* is eventually followed by a corresponding *delete*. The only exception is in the event of catastrophic failure, such as node loss. A *delete* must still be executed even if the *add* fails.
- *delete* may be followed by additional *deletes*.
- The network configuration should not change between *add* and *delete*.
- The network configuration should not change between *attachments*.
- The container runtime is responsible for cleanup of the container's network namespace.

Attachment Parameters

While a network configuration should not change between *attachments*, there are certain parameters supplied by the container runtime that are per-attachment. They are:

- **Container ID:** A unique plaintext identifier for a container, allocated by the runtime. Must not be empty. Must start with an alphanumeric character, optionally followed by

any combination of one or more alphanumeric characters, underscore (), dot (.) or hyphen (-). During execution, always set as the `CNI_CONTAINERID` parameter.

- **Namespace**: A reference to the container's "isolation domain". If using network namespaces, then a path to the network namespace (e.g. `/run/netns/[nsname]`). During execution, always set as the `CNI_NETNS` parameter.
- **Container interface name**: Name of the interface to create inside the container. During execution, always set as the `CNI_IFNAME` parameter.
- **Generic Arguments**: Extra arguments, in the form of key-value string pairs, that are relevant to a specific attachment. During execution, always set as the `CNI_ARGS` parameter.
- **Capability Arguments**: These are also key-value pairs. The key is a string, whereas the value is any JSON-serializable type. The keys and values are defined by convention.

Furthermore, the runtime must be provided a list of paths to search for CNI plugins. This must also be provided to plugins during execution via the `CNI_PATH` environment variable.

Adding an attachment

For every configuration defined in the `plugins` key of the network configuration,

1. Look up the executable specified in the `type` field. If this does not exist, then this is an error.
2. Derive request configuration from the plugin configuration, with the following parameters:
    - If this is the first plugin in the list, no previous result is provided,
    - For all additional plugins, the previous result is the result of the previous plugins.
3. Execute the plugin binary, with `CNI_COMMAND=ADD`. Provide parameters defined above as environment variables. Supply the derived configuration via standard in.
4. If the plugin returns an error, halt execution and return the error to the caller.

The runtime must store the result returned by the final plugin persistently, as it is required for *check* and *delete* operations.

Deleting an attachment

Deleting a network attachment is much the same as adding, with a few key differences:

- The list of plugins is executed in **reverse order**

- The previous result provided is always the final result of the *add* operation.

For every plugin defined in the `plugins` key of the network configuration, *in reverse order*,

1. Look up the executable specified in the `type` field. If this does not exist, then this is an error.
2. Derive request configuration from the plugin configuration, with the previous result from the initial *add* operation.
3. Execute the plugin binary, with `CNI_COMMAND=DEL`. Provide parameters defined above as environment variables. Supply the derived configuration via standard in.
4. If the plugin returns an error, halt execution and return the error to the caller.

If all plugins return success, return success to the caller.

Checking an attachment

The runtime may also ask every plugin to confirm that a given attachment is still functional. The runtime must use the same attachment parameters as it did for the *add* operation.

Checking is similar to add with two exceptions:

- the previous result provided is always the final result of the *add* operation.
- If the network configuration defines `disableCheck`, then always return success to the caller.

For every plugin defined in the `plugins` key of the network configuration,

1. Look up the executable specified in the `type` field. If this does not exist, then this is an error.
2. Derive request configuration from the plugin configuration, with the previous result from the initial *add* operation.
3. Execute the plugin binary, with `CNI_COMMAND=CHECK`. Provide parameters defined above as environment variables. Supply the derived configuration via standard in.
4. If the plugin returns an error, halt execution and return the error to the caller.

If all plugins return success, return success to the caller.

Garbage-collecting a network

The runtime may also ask every plugin in a network configuration to clean up any stale resources via the *GC* command.

When garbage-collecting a configuration, there are no Attachment Parameters.

For every plugin defined in the `plugins` key of the network configuration,

1. Look up the executable specified in the `type` field. If this does not exist, then this is an error.
2. Derive request configuration from the plugin configuration.
3. Execute the plugin binary, with `CNI_COMMAND=GC`. Supply the derived configuration via standard in.
4. If the plugin returns an error, **continue** with execution, returning all errors to the caller.

If all plugins return success, return success to the caller.

Deriving request configuration from plugin configuration

The network configuration format (which is a list of plugin configurations to execute) must be transformed to a format understood by the plugin (which is a single plugin configuration). This section describes that transformation.

The request configuration for a single plugin invocation is also JSON. It consists of the plugin configuration, primarily unchanged except for the specified additions and removals.

The following fields are always to be inserted into the request configuration by the runtime:

- `cniVersion`: the protocol version selected by the runtime - the string "1.1.0"
- `name`: taken from the `name` field of the network configuration

For attachment-specific operations (ADD, DEL, CHECK), additional field requirements apply:

- `runtimeConfig`: the runtime must insert an object consisting of the union of capabilities provided by the plugin and requested by the runtime (more details below).
- `prevResult`: the runtime must insert consisting of the result type returned by the "previous" plugin. The meaning of "previous" is defined by the specific operation (*add*, *delete*, or *check*). This field must not be set for the first *add* in a chain.
- `capabilities`: must not be set

For GC operations:

- `cni.dev/valid-attachments`: as specified in section 2.

All other fields not prefixed with `cni.dev/` should be passed through unaltered.

Deriving `runtimeConfig`

Whereas CNI_ARGS are provided to all plugins, with no indication if they are going to be consumed, *Capability arguments* need to be declared explicitly in configuration. The runtime, thus, can determine if a given network configuration supports a specific *capability*. Capabilities are not defined by the specification - rather, they are documented conventions.

As defined in section 1, the plugin configuration includes an optional key, `capabilities`. This example shows a plugin that supports the `portMapping` capability:

```
{
  "type": "myPlugin",
  "capabilities": {
    "portMappings": true
  }
}
```

The `runtimeConfig` parameter is derived from the `capabilities` in the network configuration and the *capability arguments* generated by the runtime. Specifically, any capability supported by the plugin configuration and provided by the runtime should be inserted in the `runtimeConfig`.

Thus, the above example could result in the following being passed to the plugin as part of the execution configuration:

```
{
  "type": "myPlugin",
  "runtimeConfig": {
    "portMappings": [ { "hostPort": 8080, "containerPort": 80, "protocol": '
  }
  ...
}
```

Section 4: Plugin Delegation

There are some operations that, for whatever reason, cannot reasonably be implemented as a discrete chained plugin. Rather, a CNI plugin may wish to delegate some functionality to another plugin. One common example of this is IP address management.

As part of its operation, a CNI plugin is expected to assign (and maintain) an IP address to the interface and install any necessary routes relevant for that interface. This gives the CNI plugin great flexibility but also places a large burden on it. Many CNI plugins would need to have the same code to support several IP management schemes that users may desire (e.g. dhcp, host-local). A CNI plugin may choose to delegate IP management to another plugin.

To lessen the burden and make IP management strategy be orthogonal to the type of CNI plugin, we define a third type of plugin -- IP Address Management Plugin (IPAM plugin), as well as a protocol for plugins to delegate functionality to other plugins.

It is however the responsibility of the CNI plugin, rather than the runtime, to invoke the IPAM plugin at the proper moment in its execution. The IPAM plugin must determine the interface IP/subnet, Gateway and Routes and return this information to the "main" plugin to apply. The IPAM plugin may obtain the information via a protocol (e.g. dhcp), data stored on a local filesystem, the "ipam" section of the Network Configuration file, etc.

Delegated Plugin protocol

Like CNI plugins, delegated plugins are invoked by running an executable. The executable is searched for in a predefined list of paths, indicated to the CNI plugin via `CNI_PATH`. The delegated plugin must receive all the same environment variables that were passed in to the CNI plugin. Just like the CNI plugin, delegated plugins receive the network configuration via stdin and output results via stdout.

Delegated plugins are provided the *complete network configuration* passed to the "upper" plugin. In other words, in the IPAM case, not just the `ipam` section of the configuration.

Success is indicated by a zero return code and a *Success* result type output to stdout.

Delegated plugin execution procedure

When a plugin executes a delegated plugin, it should:

- Look up the plugin binary by searching the directories provided in `CNI_PATH` environment variable.

- Execute that plugin with the same environment and configuration that it received.
- Ensure that the delegated plugin's stderr is output to the calling plugin's stderr.

If a plugin is executed with `CNI_COMMAND=CHECK`, `DEL`, or `GC`, it must also execute any delegated plugins. If any of the delegated plugins return error, error should be returned by the upper plugin.

If, on `ADD`, a delegated plugin fails, the "upper" plugin should execute again with `DEL` before returning failure.

Section 5: Result Types

For certain operations, plugins must output result information. The output should be serialized as JSON on standard out.

ADD Success

Plugins must output a JSON object with the following keys upon a successful `ADD` operation:

- `cniVersion`: The same version supplied on input - the string "1.1.0"
- `interfaces`: An array of all interfaces created by the attachment, including any host-level interfaces:
  - `name` (string): The name of the interface.
  - `mac` (string): The hardware address of the interface (if applicable).
  - `mtu`: (uint) The MTU of the interface (if applicable).
  - `sandbox` (string): The isolation domain reference (e.g. path to network namespace) for the interface, or empty if on the host. For interfaces created inside the container, this should be the value passed via `CNI_NETNS`.
  - `socketPath` (string, optional): An absolute path to a socket file corresponding to this interface, if applicable.
  - `pciID` (string, optional): The platform-specific identifier of the PCI device corresponding to this interface, if applicable.
- `ips`: IPs assigned by this attachment. Plugins may include IPs assigned external to the container.
  - `address` (string): an IP address in CIDR notation (eg "192.168.1.3/24").
  - `gateway` (string): the default gateway for this subnet, if one exists.
  - `interface` (uint): the index into the `interfaces` list for a CNI Plugin Result indicating which interface this IP configuration should be applied to.

- `routes`: Routes created by this attachment:
  - `dst`: The destination of the route, in CIDR notation
  - `gw`: The next hop address. If unset, a value in `gateway` in the `ips` array may be used.
  - `mtu` (uint): The MTU (Maximum transmission unit) along the path to the destination.
  - `advmss` (uint): The MSS (Maximal Segment Size) to advertise to these destinations when establishing TCP connections.
  - `priority` (uint): The priority of route, lower is higher.
  - `table` (uint): The table to add the route to.
  - `scope` (uint): The scope of the destinations covered by the route prefix (global (0), link (253), host (254)).
- `dns`: a dictionary consisting of DNS configuration information
  - `nameservers` (list of strings): list of a priority-ordered list of DNS nameservers that this network is aware of. Each entry in the list is a string containing either an IPv4 or an IPv6 address.
  - `domain` (string): the local domain used for short hostname lookups.
  - `search` (list of strings): list of priority ordered search domains for short hostname lookups. Will be preferred over `domain` by most resolvers.
  - `options` (list of strings): list of options that can be passed to the resolver.

Plugins provided a `prevResult` key as part of their request configuration must output it as their result, with any possible modifications made by that plugin included. If a plugin makes no changes that would be reflected in the *Success result* type, then it must output a result equivalent to the provided `prevResult`.

Delegated plugins (IPAM)

Delegated plugins may omit irrelevant sections.

Delegated IPAM plugins must return an abbreviated *Success* object. Specifically, it is missing the `interfaces` array, as well as the `interface` entry in `ips`.

VERSION Success

Plugins must output a JSON object with the following keys upon a `VERSION` operation:

- `cniVersion`: The value of `cniVersion` specified on input
- `supportedVersions`: A list of supported specification versions

Example:

```
{
    "cniVersion": "1.0.0",
    "supportedVersions": [ "0.1.0", "0.2.0", "0.3.0", "0.3.1", "0.4.0", "1.0
}
```

Error

Plugins should output a JSON object with the following keys if they encounter an error:

- `cniVersion`: The protocol version in use - "1.1.0"
- `code`: A numeric error code, see below for reserved codes.
- `msg`: A short message characterizing the error.
- `details`: A longer message describing the error.

Example:

```
{
  "cniVersion": "1.1.0",
  "code": 7,
  "msg": "Invalid Configuration",
  "details": "Network 192.168.0.0/31 too small to allocate from."
}
```

Error codes 0-99 are reserved for well-known errors. Values of 100+ can be freely used for plugin specific errors.

Error Code	Error Description
`1`	Incompatible CNI version
`2`	Unsupported field in network configuration. The error message must contain the key and value of the unsupported field.
`3`	Container unknown or does not exist. This error implies the runtime does not need to perform any container network cleanup (for example, calling the `DEL` action on the container).
`4`	Invalid necessary environment variables, like CNI_COMMAND, CNI_CONTAINERID, etc. The error message must contain the names of

Error Code	Error Description
	invalid variables.
`5`	I/O failure. For example, failed to read network config bytes from stdin.
`6`	Failed to decode content. For example, failed to unmarshal network config from bytes or failed to decode version info from string.
`7`	Invalid network config. If some validations on network configs do not pass, this error will be raised.
`11`	Try again later. If the plugin detects some transient condition that should clear up, it can use this code to notify the runtime it should re-try the operation later.
`50`	The plugin is not available (i.e. cannot service `ADD` requests)
`51`	The plugin is not available, and existing containers in the network may have limited connectivity.

In addition, stderr can be used for unstructured output such as logs.

Version

Plugins must output a JSON object with the following keys upon a `VERSION` operation:

- `cniVersion`: The value of `cniVersion` specified on input
- `supportedVersions`: A list of supported specification versions

Example:

```
{
    "cniVersion": "1.1.0",
    "supportedVersions": [ "0.1.0", "0.2.0", "0.3.0", "0.3.1", "0.4.0", "1.0
}
```

Appendix: Examples

We assume the network configuration shown above in section 1. For this attachment, the runtime produces `portmap` and `mac` capability args, along with the generic argument "argA=foo". The examples uses `CNI_IFNAME=eth0`.

Add example

The container runtime would perform the following steps for the `add` operation.

1. Call the `bridge` plugin with the following JSON, `CNI_COMMAND=ADD`:

```
{
    "cniVersion": "1.1.0",
    "name": "dbnet",
    "type": "bridge",
    "bridge": "cni0",
    "keyA": ["some more", "plugin specific", "configuration"],
    "ipam": {
        "type": "host-local",
        "subnet": "10.1.0.0/16",
        "gateway": "10.1.0.1"
    },
    "dns": {
        "nameservers": [ "10.1.0.1" ]
    }
}
```

The bridge plugin, as it delegates IPAM to the `host-local` plugin, would execute the `host-local` binary with the exact same input, `CNI_COMMAND=ADD`.

The `host-local` plugin returns the following result:

```
{
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1"
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
}
```

The bridge plugin returns the following result, configuring the interface according to the delegated IPAM configuration:

```
{
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "99:88:77:66:55:44",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
}
```

2. Next, call the `tuning` plugin, with `CNI_COMMAND=ADD`. Note that `prevResult` is supplied, along with the `mac` capability argument. The request configuration passed is:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "tuning",
  "sysctl": {
    "net.core.somaxconn": "500"
```

```
  },
  "runtimeConfig": {
    "mac": "00:11:22:33:44:66"
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "99:88:77:66:55:44",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
  }
}
```

The plugin returns the following result. Note that the **mac** has changed.

```
{
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
```

```
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
}
```

3. Finally, call the `portmap` plugin, with `CNI_COMMAND=ADD`. Note that `prevResult` matches that returned by `tuning`:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "portmap",
  "runtimeConfig": {
    "portMappings" : [
      { "hostPort": 8080, "containerPort": 80, "protocol": "tcp" }
    ]
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
```

```
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
  }
}
```

The `portmap` plugin outputs the exact same result as that returned by `bridge`, as the plugin has not modified anything that would change the result (i.e. it only created iptables rules).

Check example

Given the previous *Add*, the container runtime would perform the following steps for the *Check* action:

1. First call the `bridge` plugin with the following request configuration, including the `prevResult` field containing the final JSON response from the *Add* operation, **including the changed mac**. `CNI_COMMAND=CHECK`

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "bridge",
  "bridge": "cni0",
  "keyA": ["some more", "plugin specific", "configuration"],
  "ipam": {
    "type": "host-local",
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
```

```
  },
  "dns": {
    "nameservers": [ "10.1.0.1" ]
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
  }
}
```

The `bridge` plugin, as it delegates IPAM, calls `host-local`, `CNI_COMMAND=CHECK`. It returns no error.

Assuming the `bridge` plugin is satisfied, it produces no output on standard out and exits with a 0 return code.

2. Next call the `tuning` plugin with the following request configuration:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "tuning",
  "sysctl": {
    "net.core.somaxconn": "500"
  },
  "runtimeConfig": {
    "mac": "00:11:22:33:44:66"
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
  }
}
```


Likewise, the `tuning` plugin exits indicating success.

3. Finally, call `portmap` with the following request configuration:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "portmap",
  "runtimeConfig": {
    "portMappings" : [
      { "hostPort": 8080, "containerPort": 80, "protocol": "tcp" }
    ]
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
  }
}
```

Delete example

Given the same network configuration JSON list, the container runtime would perform the following steps for the *Delete* action. Note that plugins are executed in reverse order from the *Add* and *Check* actions.

1. First, call `portmap` with the following request configuration, `CNI_COMMAND=DEL`:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "portmap",
  "runtimeConfig": {
    "portMappings" : [
      { "hostPort": 8080, "containerPort": 80, "protocol": "tcp" }
    ]
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
```

```
  }
}
```

2. Next, call the `tuning` plugin with the following request configuration, `CNI_COMMAND=DEL`:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "tuning",
  "sysctl": {
    "net.core.somaxconn": "500"
  },
  "runtimeConfig": {
    "mac": "00:11:22:33:44:66"
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
      "nameservers": [ "10.1.0.1" ]
    }
```

```
  }
}
```

3. Finally, call `bridge`:

```
{
  "cniVersion": "1.1.0",
  "name": "dbnet",
  "type": "bridge",
  "bridge": "cni0",
  "keyA": ["some more", "plugin specific", "configuration"],
  "ipam": {
    "type": "host-local",
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
  },
  "dns": {
    "nameservers": [ "10.1.0.1" ]
  },
  "prevResult": {
    "ips": [
        {
          "address": "10.1.0.5/16",
          "gateway": "10.1.0.1",
          "interface": 2
        }
    ],
    "routes": [
      {
        "dst": "0.0.0.0/0"
      }
    ],
    "interfaces": [
        {
            "name": "cni0",
            "mac": "00:11:22:33:44:55"
        },
        {
            "name": "veth3243",
            "mac": "55:44:33:22:11:11"
        },
        {
            "name": "eth0",
            "mac": "00:11:22:33:44:66",
            "sandbox": "/var/run/netns/blue"
        }
    ],
    "dns": {
```

```
        "nameservers": [ "10.1.0.1" ]
      }
    }
}
```

The bridge plugin executes the `host-local` delegated plugin with `CNI_COMMAND=DEL` before returning.