XORP Router Manager Process (rtrmgr)

Version 1.4

XORP Project
International Computer Science Institute
Berkeley, CA 94704, USA
http://www.xorp.org/
feedback@xorp.org

March 20, 2007

1 Introduction

This document provides a high-level technical overview of the Router Manager (rtrmgr) code structure, intended to aid anyone needing to understand or modify the software. It is not a user manual.

The XORP software base consists of a number of routing protocols (BGP, OSPF, PIM-SM, etc), a Routing Information Base (RIB) process, a Forwarding Engine Abstraction (FEA) process, and a forwarding path. Other management, monitoring or application processes may also supplement this set. Figure 1 illustrates these processes and their principle communication channels.

For research purposes, these processes may be started manually or from scripts, so long as the dependencies between then are satisfied. But when using XORP in a more operational environment, the network manager typically does not wish to see the software structure, but rather would like to interact with the router *as a whole*. Minimally, this consists of a configuration file for router startup, and a command line interface to interact with the router during operation. The rtrmgr process provides this unified view of the router.

The rtrmgr is normally the only process explicitly started at router startup. The rtrmgr process includes a built-in XRL finder, so no external finder process is required. The following sequence of actions then occurs:

1. The rtrmgr reads all the template files in the router's template directory. Typically there is one template file per XORP process that might be needed. A template file describes the functionality that is provided by the corresponding process in terms of all of the configuration parameters that may be set. It also describes the dependencies that need to be satisfied before the process can be started. After reading the template files, the rtrmgr knows all the configuration parameters currently supportable on this router, and it stores this information in its *template tree*. After all template files are read, the template tree is checked for errors (*e.g.,* invalid variable names, etc). The rtrmgr will exit if there is an error.

2. The rtrmgr next reads the contents of the XRL directory to discover all the XRLs that are supported by the processes on this router. These XRLs are then checked against the XRLs in the template tree. As it is normal for the XRLs in the XRL directory to be used to generate stub code in the XORP

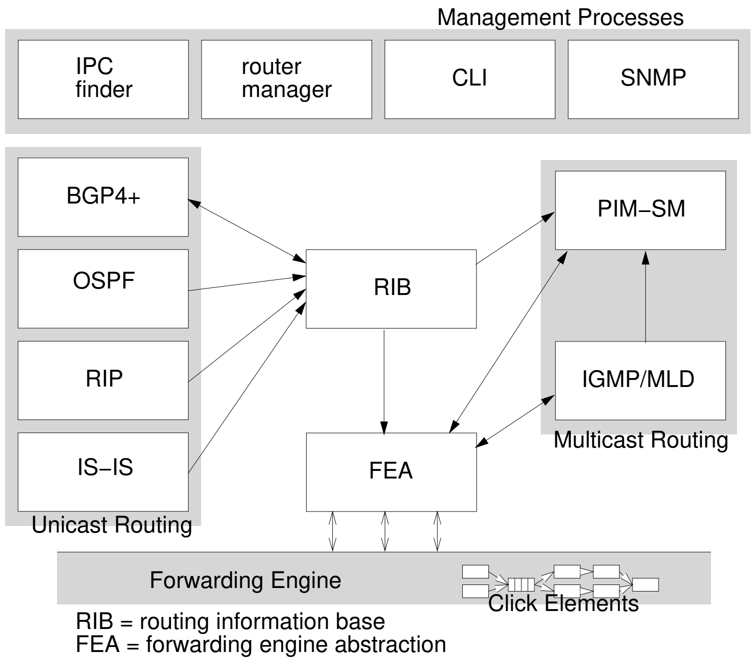


Figure 1: Overview of XORP processes

processes, this forms the definitive version of a particular XRL. Checking against this version detects if a template file has somehow become out of sync with the router's codebase. Doing this check at startup prevents subtle run time errors later. The rtrmgr will exit if a mismatch is discovered.

3. The rtrmgr then reads the router configuration file. All the configuration options in the config file must correspond to configurable functionality as described by the template files. As it reads the config file, the rtrmgr stores the intended configuration in its *configuration tree*. At this point, the nodes in the configuration tree are annotated as *not existing* - that is this part of the configuration has not yet been communicated to the process that will implement the functionality.

4. The rtrmgr next traverses the configuration tree to discover the list of processes that need to be started to provide the required functionality. Typically not all the available software on the router will be needed for a specific configuration.

5. The rtrmgr traverses the template tree again to discover an order for starting the required processes that satisfies all their dependencies.

6. The rtrmgr starts the first process in the list of processes to be started.

7. If no error occurs, the rtrmgr traverses the configuration tree to build the list of commands that need to be executed to configure the process just started. A command can be either an XRL or an external program. These commands are then called, one after another, with the successful completion of one command triggering the calling of the next. The commands are ordered according to the command semantics (*e.g.,* see below the description of commands %create, %activate, etc). If the semantics of the commands do not specify the ordering, then the commands follow the order they are defined in the rtrmgr template files. Some processes may require calling a transaction start command before

configuration, and a transaction complete command after configuration - the rtrmgr can do this if required.

8. If no error occurred during configuration, the next process is started, and configured, and so forth, until all the required processes are started and configured.

9. At this point, the router is up and running. The rtrmgr will now allow connections from the xorpsh process to allow interactive operation.

2 Template Files

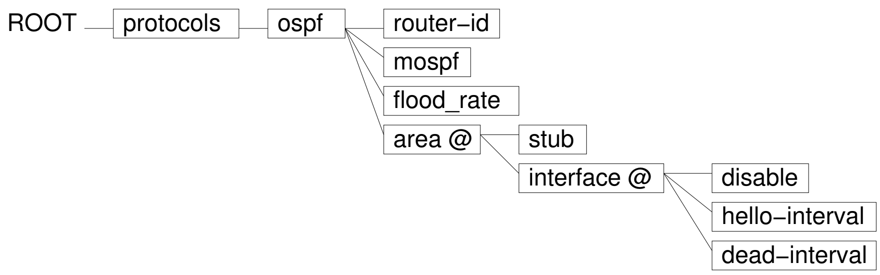
The router manager reads a directory of template files to discover the configuration options that the router supports. A fragment of such a configuration file might look like:

```
protocols {
  ospf {
    router-id: ipv4;
    mospf: toggle = false;
    flood_rate: i32;
    area @: ipv4 {
      stub: toggle = false;
      interface @: txt {
        disable: toggle = false;
        hello-interval: u32 = 30;
        dead-interval: u32 = 95;
      }
    }
  }
}
```

This defines a subset of the configuration options for OSPF. The configuration options form a tree, with three types of nodes:

- Structural nodes such as `protocol` and `ospf` that exist merely to provide scope.
- Named interior nodes such as "`area @`" and "`interface @`", where there can be multiple instances of the node. Symbol `@` indicates that a name is required; in the case of "area @" the fragment above specifies that the name must be an IPv4 address.
- Leaf nodes such as `flood_rate` and `hello-interval`. These nodes are also typed, and may optionally specify a default value. In the example above, `hello-interval` is of type `u32` (unsigned 32 bit integer), and takes the default value of 30.

Thus the template tree created from this template file would look like:



The same node may occur multiple times in the template file. This might happen because the node can take more than one type (for example, it might have an IPv4 or an IPv6 address), or it might happen because the second definition adds information to the existing definition.

In addition to specifying the configurable options, the template file should also specify what the rtrmgr should do when an option is modified. These commands annotating the template file begin with a "%". Thus the template file above might also contain the following annotated version of the template tree:

```
protocols ospf {
  %modinfo: provides ospf;
  %modinfo: depends rib;
  %modinfo: path "ospfd/xorp/ospfd";
  %modinfo: default_targetname "ospf";
  %mandatory: $(@.targetname), $(@.router-id);
  targetname {
    %set:;
  }
  router-id {
    %set: xrl "$(ospf.targetname)/ospf/0.1/set_router_id?id:u32=$(@)";
    %get: xrl "$(ospf.targetname)/ospf/0.1/get_router_id->id:u32";
  }
  area @ {
    %create: xrl "$(ospf.targetname)/ospf/0.1/add_or_configure_area?area_id:u32=$(a
    %delete: xrl "$(ospf.targetname)/ospf/0.1/delete_area?area_id:u32=$(area.@)";
  }
  mospf {
    %set: xrl "$(ospf.targetname)/ospf/0.1/set_mospf?enabled:bool=$(@)";
    %delete: xrl "$(ospf.targetname)/ospf/0.1/set_mospf?enabled:bool=$(DEFAULT)";
    %get: xrl "$(ospf.targetname)/ospf/0.1/get_mospf->enabled:bool=$(@)";
  }
}
```

The first four annotations apply to the "protocols ospf" node, and specify the "%modinfo" command, which provides information about the module providing this functionality. In this case they specify the following:

- This functionality is provided by the module called `ospf`.
- This module depends on the module called `rib`.
- The program in `ospfd/xorp/ospfd` should be run run to provide this module.
- XRL target name `ospf` should be used by default when validating an XRL specification that uses a variable inside the `ospf` module (*e.g.,* `$(ospf.targetname)`) to specify the XRL target.

The "%mandatory" annotation contains the list of nodes or variables that must be configured in the user configuration file or that must have a default value. In the above example, this applies to variables "`targetname`" and "`router-id`".

The "`protocols ospf targetname`" node carries an annotation to specify the existence of variable name "`targetname`" that can be used to specify the XRL target name of an OSPF instance. The specific value of "`targetname`" can be configured elsewhere.

The "`protocols ospf router-id`" node carries annotations to set the value of the router ID in the ospf process, and to get the value back. The set command is:

```
%set: xrl "$(ospf.targetname)/ospf/0.1/set_router_id?id:u32=$(@)";
```

This specifies that to set this value, the rtrmgr must call the specified XRL. In this case it specifies a variable expansion of variables `$(ospf.targetname)` and `$(@)`. All variables take the form `$(...)`.

The variable `$(ospf.targetname)` means the value of node "`protocols ospf targetname`". The variable `$(@)` means the value of the current node. Hence, if the targetname is set in the configuration tree to (or had a default value in the template tree of) `"ospf"`, and the router ID node in the configuration tree had the value 1.2.3.4, then the XRL to call would be:

```
ospf/ospf/0.1/set_router_id?id:u32=1.2.3.4
```

The `%set` command only applies to leaf nodes that have values and only if the value is allowed to be changed. For example, node "`protocols ospf router-id`" has `%set` command because its value can be changed. On contrary, node "`protocols ospf area @`" does not have `%set` command, because it defines a node that can have multiple instances. Each instance has a value when the instance is created, but that value cannot be changed later.

Internal nodes would typically use the `%create` command to create a new instance of the node, as shown with the "`protocols ospf area @`" node. In the example above, the `%create` command involves two variable expansions: `$(area.@)` and `$(@.stub)`. The form `$(area.@)` means "this area", and so in this case it is directly equivalent to `$(@)` meaning "this node". The variable `$(@.stub)` means the value of the leaf node called `stub` that is a child node of "this node".

Default template value of a variable can be specified by the keyword `DEFAULT`. For example, `$(DEFAULT)` or `$(@.DEFAULT)` would refer to the default template value of "this" node, while `$(foo.bar.DEFAULT)` would refer to the default template value of node `"foo.bar"`.

Thus, the template tree specifies the following information:

- The nodes of the tree specify all the configuration options possible on the router.
- Some of the nodes are annotated with information to indicate which software to run to provide the functionality rooted at that node, to indicate which other modules this software depends on being running, and to provide additional information about this module.
- Most of the nodes are annotated with commands to be run when the value of the node changes in the configuration tree, when a new instance of the node is created or an instance of the node is deleted in the configuration tree, or to get the current value of a node from the running processes providing the functionality.

Note that for verification purpuse all variable names must refer to valid nodes in the template tree. Hence, the template tree may contain dummy nodes that shoudn't be used for configuration purpose. For example, the internal variable `TID` that can be used to store the transient transaction ID should be specified as:

```
interfaces {
  %modinfo: ...
  ...

  TID {
```

```
    %create:;
  }
  ...
}
```

2.1 Template Tree Node Types

The following types are currently supported for template tree nodes:

`u32`
: Unsigned 32 bit integer

`u32range`
: A range of unsigned 32 bit integers defined by an upper and lower inclusive boundary. Boundaries are separated by two dots, e.g. `1234..5678`. If upper and lower boundaries are equal it is sufficient to specify only a single value, e.g. `1234`.

`i32`
: Signed 32 bit integer

`bool`
: Boolean - valid values are `true` and `false`.

`toggle`
: Similar to boolean, but requires a default value. Display of the config tree node is suppressed if the value is the default.

`ipv4`
: An IPv4 address in dotted decimal format.

`ipv4net`
: An IPv4 address and prefix length in the conventional format. E.g.: `1.2.3.4/24`.

`ipv4range`
: A range of IPv4 addresses defined by an upper and lower inclusive boundary. IPv4 addresses are specified in dotted decimal format delimited by two dots, e.g. `1.2.3.4..5.6.7.8`. If upper and lower boundaries are equal it is sufficient to specify only a single value, e.g. `1.2.3.4`.

`ipv6`
: An IPv6 address in the canonical colon-separated human-readable format.

`ipv6net`
: An IPv6 address and prefix in the conventional format. E.g.: `fe80::1/64`

`ipv6range`
: A range of IPv6 addresses defined by an upper and lower inclusive boundary. IPv6 addresses are specified in colon-separated format and are delimited by two dots, e.g. `fe80::1234..fe80::5678`. If upper and lower boundaries are equal it is sufficient to specify only a single value, e.g. `fe80::1234`.

macaddr
: An MAC address in the conventional colon-separated hex format. E.g.: `00:c0:4f:68:8c:58`

com32
: Unsigned 32 bit integer representing a BGP community tag. It can be specified either in a colon-separated format using two 16 bit integers, e.g. `65001:1`, or as a single 32 bit unsigned integer.

It is likely that additional types will be added in the future, as they are found to be needed.

2.2 Template Tree Commands

This section provides a complete listing of all the template tree commands that the rtrmgr supports.

2.2.1 The `%modinfo` Command.

The sub-commands to the `%modinfo` command are:

`%modinfo: provides` *ModuleName*
: The `provides` subcommand takes one additional parameter, which gives the name of the module providing the functionality rooted at this node.

`%modinfo: depends` *list of modules*
: The `depends` subcommand takes at least one additional parameter, giving a list of the other modules that must be running and configured before this module may be started.

`%modinfo: path` *ProgramPath*
: The `path` subcommand takes one additional parameter giving the pathname of the software to be run to provide this functionality. The pathname may be absolute or relative to the root of the XORP tree. The ordering in computing the root of the tree is: (a) the shell environment XORP_ROOT (if exists); (b) the parent directory the rtrmgr is run from (only if it contains the etc/templates and the xrl/targets directories); (c) the XORP_ROOT value as defined in config.h (currently this is the installation directory, and defaults to "/usr/local/xorp").

`%modinfo: default_targetname` *TargetName*
: The `default_targetname` subcommand takes one additional parameter giving the value of the XRL target name that should be used by default when validating an XRL specification (*e.g.,* if the specification uses a variable inside that module to specify the XRL target name).

`%modinfo: start_commit` *method argument*
: The `start_commit` subcommand takes two or more additional parameters, that are used to specify the mechanism to be call before performing any change to the configuration of the module. The only methods currently supported are `xrl` which takes an XRL specification as an argument, and `program` which takes an executable program as an argument.

`%modinfo: end_commit` *method argument*
: The `end_commit` subcommand takes two or more additional parameters, that are used to specify the mechanism to be called to complete any change to the configuration of the module. The only methods currently supported are `xrl` which takes an XRL specification as an argument, and `program` which takes an executable program as an argument. Both `start_commit` and `end_commit` are optional. They provide a way to make batch changes to a module configuration as an atomic operation.

`%modinfo: status_method` *method argument*
: The `status_method` subcommand takes two or more additional parameters, that are used to specify the mechanism to be used to discover the status of the module. The only methods current supported are `xrl` which takes an XRL specification as an argument, and `program` which takes an executable program as an argument.

`%modinfo: startup_method` *method argument*
: The `startup_method` subcommand takes two or more additional parameters, that are used to specify the mechanism to be used to gracefully startup the module. The only methods current supported are `xrl` which takes an XRL specification as an argument, and `program` which takes an executable program as an argument.

Before the `startup_method` subcommand is called, it is expected that the process is in `PROC_STARTUP` state; after the subcommand is called the process should transition to the `PROC_READY` state. Note that this subcommand is optional and if it is not specified, then it is expected that the process will transition on its own to the `PROC_READY` state.

`%modinfo: shutdown_method` *method argument*
: The `shutdown_method` subcommand takes two or more additional parameters, that are used to specify the mechanism to be used to gracefully shutdown the module. The only methods current supported are `xrl` which takes an XRL specification as an argument, and `program` which takes an executable program as an argument. If the process does not then transition to `PROC_SHUTDOWN` state, the rtrmgr will then kill the process.

2.2.2 The `%mandatory` Command.

`%mandatory` is used to specify the list of nodes or variables that must be configured in the user configuration file or that must have a default value. This command can appear multiple times anywhere in the template tree. If it appears multiple times within the same template node, then all listed nodes are mandatory. However, note that it cannot be used to specify a multi-value node such as
`$(interfaces.interface.@.vif.@.address.@)`.

2.2.3 The `%create` Command.

`%create` is used to create a new instance of an interior node in the configuration tree.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to create the new configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to create the new configuration tree instance of this template tree node.

Note that if a node has no `%create` command, then the `%set` command (if exists) for that node is used instead (see below).

2.2.4 The `%activate` Command.

`%activate` is used to activate a new instance of an interior node in the configuration tree. It is typically paired with `%create` - the `%create` command is executed before the relevant configuration of the node's children has been performed, whereas `%activate` is executed after the node's children have been configured. A particular interior node might have either `%create`, `%activate` or both.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to activate the new configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to activate the new configuration tree instance of this template tree node.

For example, if the template tree held the following:

```
address @:  ipv4 {
    %create:  xrl XRL1;
    %activate:  xrl XRL2;
    netmask:  ipv4 {
        %set:  xrl XRL3;
    }
}
```

Then when an instance of address and netmask are created and configured, the execution order of the XRLs will be: *XRL1, XRL3, XRL2*.

2.2.5 The `%update` Command.

`%update` is used to update an existing instance of a node in the configuration tree. It is typically paired with `%activate` - the `%activate` command is executed after the node's children have been configured for very first time (*e.g.,* on startup), whereas `%update` is executed if some of the node's children have been modified (*e.g.,* via xorpsh).

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to update the configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to update the configuration tree instance of this template tree node.

Note that if the value of a node is modified, only the closest `%update` command up in the hierarchy is executed. For example, if the template tree held the following:

```
address @: ipv4 {
    %create: xrl XRL1;
    %activate: xrl XRL2;
    %update: xrl XRL3;
    netmask: ipv4 {
        %update: xrl XRL4;
        disable: bool {
            %set:;
        }
    }
    broadcast: ipv4 {
        %set:;
    }
}
```

Then when the value of `disable` is modified, only *XRL4* will be called. If the value of `broadcast` is modified, then *XRL3* will be called.

2.2.6 The `%list` Command.

`%list` is called to obtain a list of all the configuration tree instances of a particular template tree node. For example, a particular template tree node might represent the interfaces on a router. The configuration tree would then contain an instance of this node for each interface currently configured. The `%list` command on this node would then return the list of interfaces.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to return the list.
- If the action is `program`, then the second parameter specifies the program to execute to return the list.

2.2.7 The `%delete` Command.

`%delete` is called to delete a configuration tree node and all its children. A node that has a `%create` or `%activate` command should also have a `%delete` command.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to delete the configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to delete the configuration tree instance of this template tree node.

If a node that is deleted does not have a `%delete` command, then the `%delete` commands of its children are called instead. This rule is applied recursively for each child that does not have a `%delete` command. For example, lets say A is a parent of B1 and B2, and B1 is a parent of C1. Also, lets say that only B2 and C1 have `%delete` methods. If we delete A, then both B2's and C1's `%delete` methods are invoked. If, however, B1 also has a `%delete` method, then deleting A will invoke only B1 and B2's `%delete` methods.

2.2.8 The `%set` Command.

`%set` is called to set the value of a leaf node in the configuration tree.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to set the value of configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to set the value of configuration tree instance of this template tree node.

Note that when a new instance of a node in the configuration tree is created, if that node has no `%create` command, then the `%set` command (if exists) for that node is used instead.

2.2.9 The `%unset` Command.

`%unset` is called to unset the value of a leaf node in the configuration tree. The value will return to its default value if a default value is specified.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.
- If the action is `xrl`, then the second parameter specifies the XRL to call to unset the value of configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to unset the value of configuration tree instance of this template tree node.

2.2.10 The `%get` Command.

`%get` is called to get the value of a leaf node in the configuration tree. Normally the rtrmgr will know the value if there is no external means to change the value, but the `%get` command provides a way for the rtrmgr to re-sync if the value has changed.

- The first parameter indicates the form of action to take to perform this action - typically it is `xrl` which indicates an XRL should be called. To execute an external program instead, the action should be `program`.

- If the action is `xrl`, then the second parameter specifies the XRL to call to get the value of configuration tree instance of this template tree node.
- If the action is `program`, then the second parameter specifies the program to execute to get the value of configuration tree instance of this template tree node.

2.2.11 The `%allow` Command.

The `%allow` command provides a way to restrict the value of certain nodes to specific values.

- The first parameter gives the name of the variable to be restricted.
- The second parameter is a possible allowed value for this variable.
- The third parameter must be the "%help:" keyword.
- The fourth parameter is the help string for this value.

If there is more than one possible values, each of them should be specified by a separate `%allow` command.

For example, a node might specify an address family, which is intended to be one of "inet" or "inet6". The type of the node is `txt`, which would allow any value, so the allow command might allow the rtrmgr to restrict the legal values without having to communicate with the process providing this functionality.

A more subtle use might be to allow certain nodes to exist only if a parent node was of a certain value.

For example:

```
family @: txt {
    %allow: $(@) "inet" %help: "IPv4 address family";
    %allow: $(@) "inet6" %help: "IPv6 address family";
    address @: ipv4 {
        %allow: $(family.@) "inet" %help: "IPv4 address family";
        broadcast: ipv4;
    }
    address @: ipv6 {
        %allow: $(family.@) "inet6" %help: "IPv6 address family";
    }
}
```

In this case, there are two different typed versions of the "`address @`" node, once for IPv4 and one for IPv6. Only one of them has a leaf node called `broadcast`. The allow command permits the rtrmgr to do type-checking to ensure that only the permitted combinations are allowed.

2.2.12 The `%allow-range` Command.

The `%allow-range` command restricts the range of values an integer configuration item may take. The syntax is:

`%allow-range:` *varName lowValue highValue %help: help-string*`;`

where the first parameter, *varName*, gives the name of the variable to be restricted. This is typically ¨`$(@)`¨. The *lowValue* and *highVal* parameters specify the lower and upper bound of the allowed range of values. The *%help:* is a mandatory keyword and is followed by the help string. The help string is used for command-line completion purpose.

An example of use appears in the interface address prefix specification:

```
address @: ipv4 {
  prefix-length: u32;
}
...
address @: ipv4 {
  prefix-length {
    %allow-range: $(@) "1" "32" %help: "The prefix length";
    %set: xrl "...";
    %get: xrl "...";
  }
}
```

If there is more than one `%allow-range` command restricting the value of a variable, then the assigned value must belong to any of the specified ranges.

2.2.13 The `%help` Command.

The `%help` command specifies the CLI configuration-mode help string. The syntax is:
`%help:` {*short* | *long*} `"Help string";`
where the first parameter, *short* or *long*, specifies whether this is the short-version or the long-version of the help, and the second parameter is the help string itself.

2.2.14 The `%deprecated` Command.

The `%deprecated` command can be used to deprecate a template tree node and the subtree below it. The syntax is:
`%deprecated: "String with reason";`

If the XORP startup configuration contains a statement that uses a deprecated node in the template, the rtrmgr prints an error with the string with the reason, and exits. If, however, a third-party user program (*e.g.,* other than xorpsh) sends to the rtrmgr configuration that contains a deprecated statement, the rtrmgr returns an error to xorpsh, and the error message will contain the string with the reason.

2.2.15 The `%user-hidden` Command.

The `%user-hidden` command can be used to hide a template tree node and the subtree below it from the user. Such node or a subtree can be used by the rtrmgr itself for internal purpose only and is not visible to the user (*e.g.,* via xorpsh or when saving the configuration to a file). The syntax is:
`%user-hidden: "String with reason";`

However, if the XORP startup configuration contains a statement that uses an user-hidden node, the rtrmgr will accept the configuration. Similarly, if a third-party user program (*e.g.,* other than xorpsh) sends to the rtrmgr configuration that contains an user-hidden statement, the rtrmgr would accept that statement. This is an experimental feature may become permanent or may be disabled in the future.

2.2.16 The `%read-only` Command.

The `%read-only` command can be used to specify a template tree node as read-only. The syntax is:

`%read-only: "String with reason";`

or

`%read-only:;`

Only a leaf node that contains a value can be marked as read-only. If a node is marked as a read-only, then its value cannot be changed from the default template value. For example, a read-only node could be part of the startup configuration, but if its value is different from the default template value the rtrmgr will reject the configuration.

Note that by definition a read-only node is also permanent (see Section 2.2.17): it cannot be deleted directly, but it will be removed if its parent is deleted.

2.2.17 The `%permanent` Command.

The `%permanent` command can be used to specify a template tree node as a permanent node that cannot be deleted. The syntax is:

`%permanent: "String with reason";`

or

`%permanent:;`

If a node is marked as permanent, the node itself cannot be deleted directly. However, deleting the parent node will delete the permanent node as well. Also, adding or deleting children of a permanent node is allowed.

If a permanent node never should be deleted, then all its ancestors should be marked as permanent.

2.2.18 The `%order` Command.

The `%order` command provides a way to specify the ordering of multiple nodes of the same type in the configuration. For example, if no ordering is specified in the template file, such as with interfaces:

```
interfaces {
  interface @: txt {
  }
}
```

Then this template would allow the configuration for each `interface` to be displayed and configured in the order they were entered. For example, the configuration might be:

```
interfaces {
  interface fxp1 {
    vif fxp1 {
       address: 10.0.0.1
    }
  }
```

```
    interface dc0 {
      vif dc0 {
         address: 10.0.1.1
      }
    }
    interface fxp0 {
      vif fxp0 {
         address: 10.0.2.1
      }
    }
  }
```

The ordering of the `interface` sections here is arbitrary, in the order they were entered by the user.
In many cases this is what is desired, but in some cases such as firewall rules, this is not desired, and the `%order` command provides a way to enforce an ordering.
For example, a simple firewall (not the actual XORP firewall) might use a template such as:

```
firewall {
  interface @: txt {
     rule @: u32 {
       %order: sorted-numeric;
       permit @: txt;
       deny @:txt;
     }
   }
 }
```

Thus, some configured firewall rules might be:

```
firewall {
  interface fxp0 {
    rule 100 {
      permit "net 10.0.0.0/24"
    }
    rule 300 {
      deny "all"
    }
  }
}
```

The ordering here is now dictated by `rule` number, in accordance with the tt %order command. If a new `rule 200` was subsequently inserted, it would always be displayed and configured after `rule 100` and before `rule 300`.
The available parameters for the `%order` command are:

- `unsorted` - the default, ordered in the order of entry.

- sorted-numeric - sorted in increasing numeric interger order.
- sorted-alphabetic - sorted in increasing alphabetic order.

Note that if sorted-numeric is applied to a txt field, the sort order for non-numeric values is undefined, but numeric values will be sorted correctly.

2.3 Template Tree Command Actions

Template tree commands such as:

- %modinfo: start_commit <method>;
- %modinfo: end_commit <method>;
- %modinfo: status_method <method>;
- %modinfo: startup_method <method>;
- %modinfo: shutdown_method <method>;

are used to specify the mechanism to be call before any configuration change of a module, the mechanism to discover the status of a module, and so on. Template tree commands such as %create, %activate and %set are used to specify the actions that need to be performed when the router configuration is created or modified.

This section provides a complete listing of all the template tree actions that the rtrmgr supports.

2.3.1 Template Tree xrl Action

The xrl command action specifies the XRL to be executed. The XRL and its arguments must be inside quotes and it may contains variables that will be substituted with the particular values at execution time. For example, if the template tree held the following:

```
bgp-id {
    %set: xrl "$(bgp.targetname)/bgp/0.2/set_bgp_id?id:ipv4=$(@)";
}
```

Then when we set the value of bgp-id, first the rtrmgr will substitute $(bgp.targetname) with the particular value of that variable, and $(@) with the value of bgp-id. After the substitution it will call XRL bgp/0.2/set_bgp_id with argument id:ipv4 set to the value of bgp-id.

We could use xrl actions to get the value of a particular variable, store the value inside the rtrmgr and then use that value by other actions. For example, if the template tree held the following:

```
interface {
    ...
    %modinfo: start_commit xrl "$(interface.targetname)/ifmgr/0.1/
                          start_transaction->tid:u32=$(interface.TID)";
    %modinfo: end_commit   xrl "$(interface.targetname)/ifmgr/0.1/
                          commit_transaction?tid:u32=$(interface.TID)";
```

```
    ...
    TID {
        %create:;
    }

    interface @: txt {
        %create: xrl "$(interface.targetname)/ifmgr/0.1/
                 create_interface?tid:u32=$(interface.TID)&ifname:txt=$(@)";
    }

    ...
}
```

Then whenever the interface configuration is changed the `start_commit` and `end_commit` XRLs will be call before and after performing any change to the configuration of the module. The `start_commit` XRL will return the transaction ID `tid` of type `u32`. The rtrmgr will store that value internally in the `$(interface.TID)` local variable (note that this variable should be declared as a leaf node without type). Then this value can be used by other actions such as the `%create` and the `end_commit` XRL actions in the above example.

2.3.2 Template Tree `program` Action

The `xrl` command action specifies the external program to be executed. The program and its arguments must be inside quotes and it may contains variables that will be substituted with the particular values at execution time. For example, if the template tree held the following:

```
    foo {
        %set: program "/bin/echo -n '$(@)' >> /tmp/file.txt";
    }
```

Then when we set the value of `foo`, first the rtrmgr will substitute `$(@)` with the value of `foo`. After the substitution it will call program `/bin/echo` with argument `-n` and the value of `foo`. The result of this command will be appended to file `/tmp/file.txt`.

We could use `program` actions to store the `stdout` and `stderr` output of a command inside the rtrmgr and then use those values by other actions. For example, if the template tree held the following:

```
rtrmgr {
    ...
    CONFIG {
        %create:;
    }
    CONFIG_STDERR {
        %create:;
    }

    load {
        %create:;
```

```
        %set: program "/bin/cat '$(@)' ->
              stdout=$(rtrmgr.CONFIG)&stderr=$(rtrmgr.CONFIG_STDERR)";
    }
    save {
        %create:;
        %set: program "/bin/echo -n '$(rtrmgr.CONFIG)' > '$(@)'";
    }
    ...
}
```

Then whenever we change the value of variable `load`, the external program `/bin/cat` will be executed with the value of that variable as its argument. The rtrmgr will store the stdout and stderr output of that program internally inside local variables `$(rtrmgr.CONFIG)` and `$(rtrmgr.CONFIG_STDERR)` respectively (note that those variables should be declared as leaf nodes either with or without type). Then those values can be used by other actions such as the `%set` action for the `save` node in the above example.

3 The Configuration File

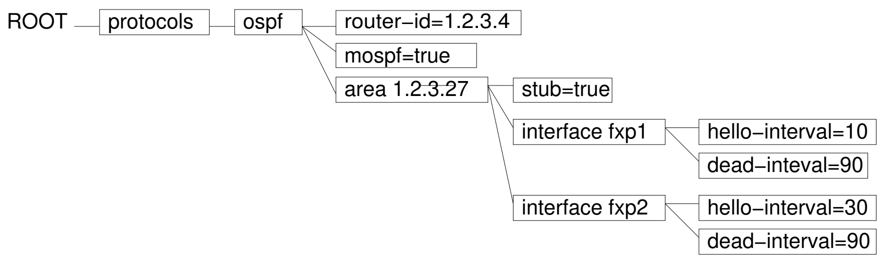
Whereas the template files inform the rtrmgr as the *possible* configuration of the router, the configuration file provides the specific startup configuration to be used by this specific router. The syntax is similar to, but not the same as, that of template files - the differences are intentional - template files are intended to be written by software developers, whereas configuration files are intended to be written by network managers. Hence the syntax of configuration files is simpler and more intuitive, but less powerful. However, both specify the same sort of tree structure, and the nodes in the configuration tree must correspond to the nodes in the template tree.

An example fragment of a configuration file might be:

```
protocols {
  ospf {
    router-id: 1.2.3.4
    mospf
    area 1.2.3.27 {
      stub
      interface fxp1 {
        hello-interval: 10
      }
      interface fxp2
    }
  }
}
```

Note that unlike in the template tree, semicolons are not needed in the configuration tree, and that line-breaks are significant.

The example fragment of a configuration file above will construct the following configuration tree from the template tree example given earlier:

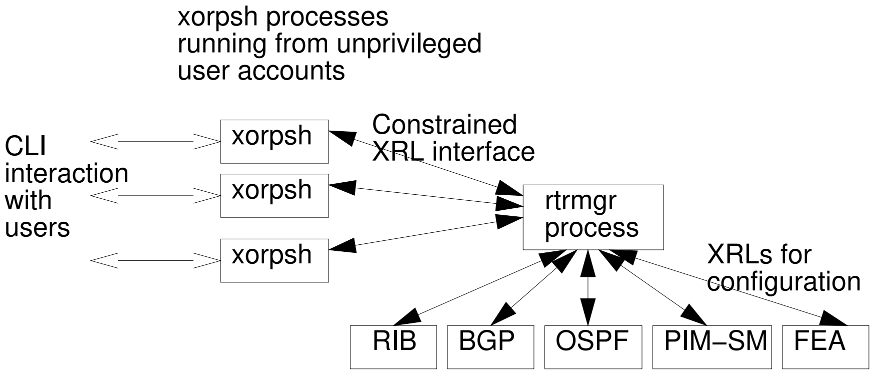


Note that configuration tree nodes have been created for `dead-interval` and (in the case of fxp1) for `hello-interval` even though this was not mentioned in the configuration file. This is because the template tree contains a default value for this leaf node. Also, in case of configuring a boolean variable (*e.g.,* of type `bool` or `toggle`) such as `mospf`, typing the variable name itself (*e.g.,* `mospf`) is equivalent to assigning it value of `true` (*e.g.,* `mospf: true`).

4 Command Line Interface: xorpsh

The rtrmgr process is the core of a XORP router - it starts and stops processes and keeps track of the configuration. To do its task, it must run as root, whereas most other XORP processes don't need privileged operation and so can be sandboxed. This makes the rtrmgr process the single most critical point from a security point of view. Thus we would like the rtrmgr to be as simple as possible[1], and to isolate it from possibly hostile input as far as is reasonable.

For these reasons we do not build a command line interface directly into the rtrmgr, but instead use an external process called `xorpsh` to interact with the user, while limiting the rtrmgr's interaction with xorpsh to simple authentication mechanisms, and exchanges of configuration tree data. Thus the command line interface architecture looks like:



The interface between the rtrmgr and a xorpsh instance consists of XRLs that the xorpsh may call to query or configure rtrmgr, and a few XRLs that the rtrmgr may asynchronously call to alert the xorpsh process to certain events.

The rtrmgr exports the following XRLs that may be called by xorpsh:

`register_client`
: This XRL is used by a xorpsh instance to register with the rtrmgr. In response, the rtrmgr provides the name of a file containing a nonce - the xorpsh must read this file and return the contents to the rtrmgr to authenticate the user.

`authenticate_client`
: Xorpsh uses this to complete the authentication process.

`get_running_config`
: Xorpsh uses this to request the current running configuration from the rtrmgr. The response is text in the same syntax as the rtrmgr configuration file that provides the rtrmgr's view of the configuration.

`enter_config_mode`
: A xorpsh process must be in configuration mode to submit configuration changes to the rtrmgr. This XRL requests that the rtrmgr allows the xorpsh to enter configuration mode. Not all users have permission to enter configuration mode, and it is also possible that a request may be refused because the configuration is locked.

[1]Unfortunately the router manager is not simple as we would like.

`get_config_users`
: Xorpsh uses this to request the list of users who are currently in configuration mode.

`apply_config_change`
: Xorpsh uses this to submit a request to change the running configuration of the router to the rtrmgr. The change consists of a set of differences from the current running configuration.

`lock_config`
: Xorpsh uses this to request an exclusive lock on configuration changes. Typically this is done just prior to submitting a set of changes.

`unlock_config`
: Unlocks the rtrmgr configuration that was locked by a previous call to `lock_config`.

`lock_node`
: Xorpsh uses this to request a lock on configuration changes to a specific config tree node. Usually this will be called because the user has made local changes to the config but not yet committed them, and wishes to prevent another user making changes that conflict. Locking is no substitute for human-to-human configuration, but it can alert users to potential problems.

  Note: node locking is not implemented yet.

`unlock_node`
: Xorpsh uses this to request a lock on a config tree node be removed.

`save_config`
: Xorpsh uses this to request the configuration be saved to a file. The actual save is performed by the rtrmgr rather than by xorpsh, but the resulting file will be owned by the user running this instance of xorpsh, and the file cannot overwrite files that this user would not otherwise be able to overwrite.

`load_config`
: Xorpsh uses this to request the rtrmgr reloads the router configuration from the named file. The file must be readable by the user running this instance of xorpsh, and the user must be in configuration mode when the request is made.

`leave_config_mode`
: Xorpsh uses this to inform rtrmgr that it is no longer in configuration mode.

Each xorpsh process exports the following XRLs that the rtrmgr can use to asynchronously communicate with the xorpsh instance:

`new_config_user`
: Rtrmgr uses this XRL to inform all xorpsh instances that are in config mode than another user has entered config mode.

`config_change_done`
: When a xorpsh instance submits a request to the rtrmgr to change the running config or to load a config from a file, the rtrmgr may have to perform a large number or XRL calls to implement the config

change. Due to the single-threaded nature of XORP processes, the rtrmgr cannot do this while remaining in the `apply_config_change` XRL, so it only performs local checks on the sanity of the request before returning success or failure - the configuration will not have actually been changed at that point. When the rtrmgr finishes making the change, or when failure occurs part way through making the change, the rtrmgr will call `config_change_done` on the xorpsh instance that requested the change to inform it of the success or failure.

`config_changed`
When multiple xorpsh processes are connected to the rtrmgr, and one of them submits a successful change to the configuration, the differences in the configuration will then be communicated to the other xorpsh instances to keep their version of the configuration in sync with the rtrmgr's version.

4.1 Operational Commands and xorpsh

Up to this point, we have been dealing with changes to the router configuration. Indeed this is the role of the rtrmgr process. However a router's command line interface is not only used to change or query the router configuration, but also to learn about the dynamic state of the router, such as link utilization or routes learned by a routing protocol. To keep it as simple and robust as possible, the rtrmgr is not involved in these *operational mode* commands. Instead these commands are executed directly by a xorpsh process itself.

To avoid the xorpsh implementation needing in-built knowledge of router commands, the information about operational mode commands is loaded from another set of template files. A simple example might be:

```
show interfaces $(interfaces.interface.*) {
        %command: "path/to/show_interfaces -i $3" %help: HELP;
        %module: fea;
        %opt_parameter: "brief" %help: BRIEF;
        %opt_parameter: "detail" %help: DETAIL;
        %opt_parameter: "extensive" %help: EXTENSIVE;
        %tag: HELP "Show network interface information";
        %tag: BRIEF "Show brief network interface information";
        %tag: DETAIL "Show detailed network interface information";
        %tag: EXTENSIVE "Show extensive network interface information";
}
show vif $(interfaces.interface.*.vif.*) {
        %command: "path/to/show_vif -i $3" %help: "Show vif information";
        %module: fea;
        %opt_parameter: "brief" %help: "Show brief vif information";
        %opt_parameter: "detail" %help: DETAIL;
        %opt_parameter: "extensive" %help: EXTENSIVE;
        %tag: DETAIL "Show detailed vif information";
        %tag: EXTENSIVE "Show extensive vif information";
}
```

This template file defines two operational mode commands: "`show interfaces`" and "`show vif`".

The "show interfaces" command takes one mandatory parameter, whose value must be the name of one of the configuration tree nodes taken from the variable name wildcard expansion `$(interfaces.interface.*)`.

Thus if the router had config tree nodes called "`interfaces interface xl0`", and "`interfaces interface xl1`", then the value of the mandatory parameter must be either `xl0` or `xl1`.

Additional optional parameters might be `brief`, `detail`, or `extensive` - the set of allowed optional parameters is specified by the `%opt_parameter` commands.

The `%command` command indicates the program or script (and its arguments) to be executed to implement this operational command - the script should return human-readable output preceded by a MIME content type indicating whether the text is structured or not[2]. If the command specification contains any positional arguments (*e.g.,* `$0`, `$1`, `$2`) they are resolved by substituting them with the particular substring from the typed command line command: `$0` is substituted with the complete string from the command line, `$1` is substituted with the first token from the command line, `$2` is substituted with the second token from the command line, The resolved positional arguments along with the remaining arguments (if any) are passed to the executable command. For example, if the user types "show interfaces xl0", the xorpsh might invoke the `show_interface` command using the Unix command line:

```
path/to/show_interfaces -i xl0
```

The pathname to a command must be relative to the root of the XORP tree. The ordering in computing the root of the tree is: (a) the shell environment XORP_ROOT (if exists); (b) the parent directory the xorpsh is run from (only if it contains the etc/templates and the xrl/targets directories); (c) the XORP_ROOT value as defined in config.h (currently this is the installation directory, and defaults to "/usr/local/xorp").

The command `%module` indicates that this operational command should only be available through the CLI when the router configuration has required that the named module has been started. If the `%module` command is missing, then this operational command is always enabled.

The command `%help` is used to specify the CLI help for each CLI command or the optional parameters. It must be on the same line as the `%command` or the `%opt_parameter` commands. If the argument after the `%help` command is in quotes, then it contains the help string itself. Otherwise, the argument is the name of the tag that contains the help string.

The command `%tag` is used to specify the help string associated with each tag. For example, statement:

```
%command: "path/to/show_vif -i $3" %help: HELP;
%tag: HELP "Show vif information";
```

is equvalent with:

```
%command: "path/to/show_vif -i $3" %help: "Show vif information";
```

Note: currently there is no security mechanism restricting access to operational mode commands beyond the restrictions imposed by Unix file permissions. This is not intended to be the long-term situation.

A Modification History

- December 11, 2002: Initial version 0.1 completed.
- March 10, 2003: Updated the version to 0.2, and the date.

[2]Only `text/plain` is currently supported.

- June 9, 2003: Updated to match XORP release 0.3.
- August 28, 2003: Updated to match XORP release 0.4.
- November 6, 2003: Updated the version to 0.5, and the date.
- July 8, 2004: Updated to match XORP release 1.0.
- January 27, 2005: Removed MFEA+MRIB-related text, because the MFEA does not deal with the MRIB information anymore.
- April 13, 2005: Updated to match XORP release 1.1.
- March 8, 2006: Updated to match XORP release 1.2.
- August 2, 2006: Updated the version to 1.3, and the date.
- March 20, 2007: Updated the version to 1.4, and the date.