XORP Router Manager Process (rtrmgr)

Version 0.1

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

December 11, 2002

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.

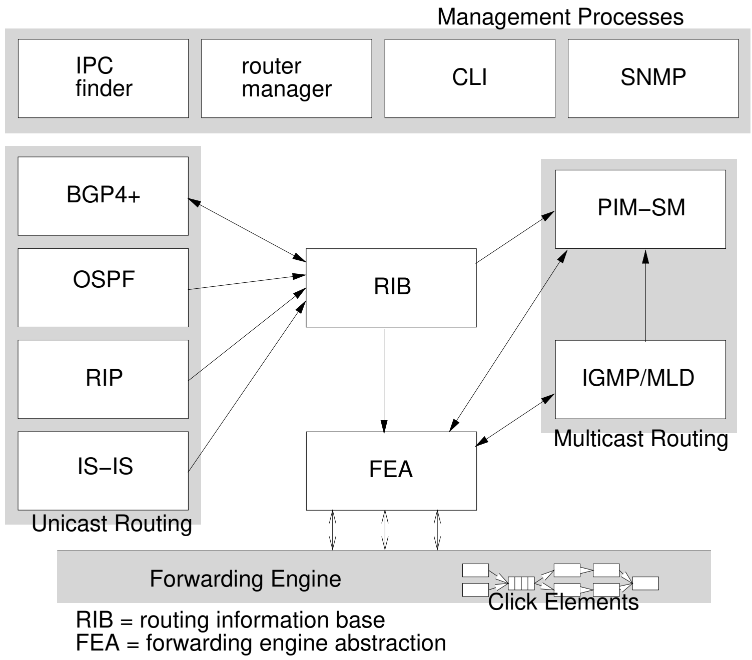


Figure 1: Overview of XORP processes

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*.

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 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 XRLs that need to be called to configure the process just started. These XRLs are then called, one after another, with the successful completion of one XRL triggering the calling of the next. Some processes may require calling a transaction start XRL before configuration, and a transaction complete XRL 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

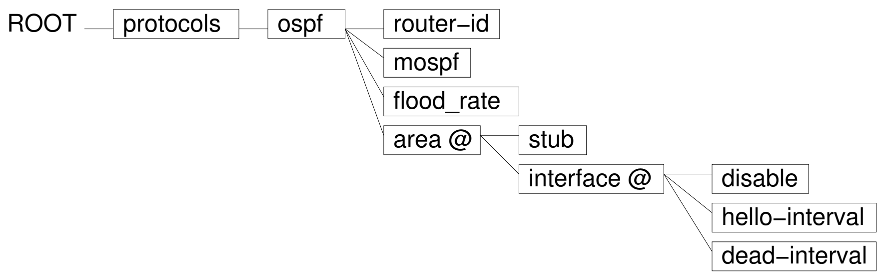
The router manager reads a directory of template files to discover the configuration options that the router supports. A fragment of such a configuration file might look like:

```
protocols {
  ospf {
    router-id: ipv4;
    mospf: toggle = false;
    flood_rate: int;
    area @: ipv4 {
      stub: toggle = false;
      interface @: text {
        disable: toggle = false;
        hello-interval: uint = 30;
        dead-interval: uint = 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. Here 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 `uint` (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";
  router-id {
    %set: xrl "ospf/ospf/0.1/set_router_id?id:u32=$(@)";
    %get: xrl "ospf/ospf/0.1/get_router_id->id:u32";
  }
  area @ {
    %create: xrl "ospf/ospf/0.1/add_or_configure_area?
                  area_id:u32=$(area.@)&is_stub:bool=$(@.stub)";
    %delete: xrl "ospf/ospf/0.1/delete_area?area_id:u32=$(area.@)";
  }
}
```

The first three 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 that this functionality is provided by the module called `ospf`, that this module depends on the module called `rib`, and that the software in `../ospfd/xorp/ospfd` is the software to run to provide this module.

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/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 the variable `$(@)`. All variables take the form `$(...)`.

The variable `$(@)` means the value of the current node, so if 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.

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".

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, and to indicate which other modules this software depends on being running.

- 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.

2.1 Template Tree Node Types

The following types are currently supported for template tree nodes:

`uint`
: Unsigned 32 bit integer

`int`
: 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.

`ipv4_prefix`
: An IPv4 address and prefix length in the conventional format. E.g.: `1.2.3.4/24`.

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

`ipv6_prefix`
: An IPv6 address and prefix in the conventional format. E.g.: `fe80::1/64`

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

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.

`%modinfo startcommit` *XRL*
: The `startcommit` subcommand takes one additional parameter, and gives the XRL to call before performing any change to the configuration of the module.

`%modinfo endcommit` *XRL*
: The `endcommit` subcommand takes one additional parameter, and gives the XRL to call to complete any change to the configuration of the module. startcommit and endcommit are optional. They provide a way to make batch changes to a module configuration as an atomic operation.

2.2.2 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.
- If the action is `xrl`, then the second parameter gives the XRL to call to create the new configuration tree instance of this template tree node.

2.2.3 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.
- If the action is `xrl`, then the second parameter gives the XRL to call 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*.

2.2.4 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.
- If the action is `xrl`, then the second parameter gives the XRL to call to return the list.

2.2.5 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.
- If the action is `xrl`, then the second parameter gives the XRL to call to delete the configuration tree instance of this template tree node.

2.2.6 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.
- If the action is `xrl`, then the second parameter gives the XRL to call to set the value of configuration tree instance of this template tree node.

2.2.7 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.
- If the action is `xrl`, then the second parameter gives the XRL to call to unset the value of configuration tree instance of this template tree node.

2.2.8 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.
- If the action is `xrl`, then the second parameter gives the XRL to call to get the value of configuration tree instance of this template tree node.

2.2.9 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 remaining parameters are a list of possible allowed values for this variable.

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 `text`, 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 @: text {
  %allow: $(@) "inet" "inet6";
  address @: ipv4 {
    %allow: $(family.@) "inet";
    broadcast: ipv4;
  }
  address @: ipv6 {
    %allow: $(family.@) "inet6";
  }
}
```

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.

3 The Configuration File

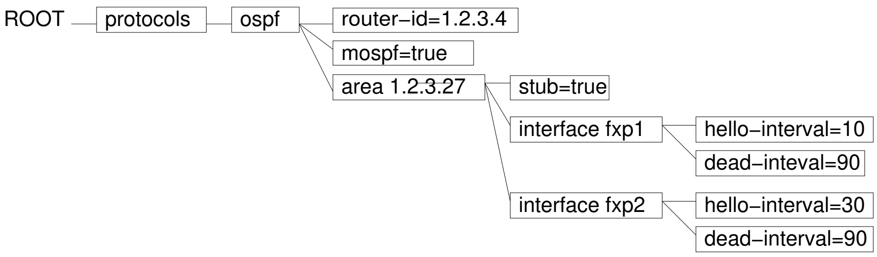
Whereas the template files inform the rtrmgr as the *possible* configuration of the router, the configuration file provides the specific startup configuration to be used by this specific router. The syntax is similar to, but not the same as, that of template files - the differences are intentional - 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:

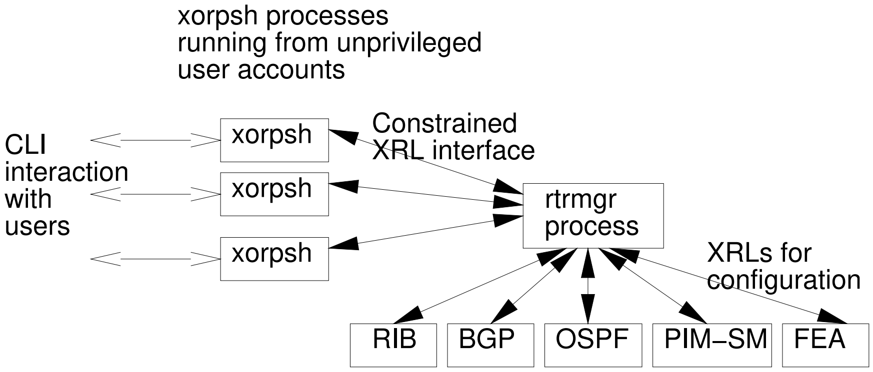


Note that configuration tree nodes have been created for `dead-interval` and (in the case of fxp1) for `hello-interval` even though this was not mentioned in the configuration file. This is because the template tree contains a default value for this leaf node.

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 it's 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 yet implemented.

`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 interface $(interfaces.interface.*) {
        %command: show_interface;
        %module: fea;
        %opt_parameter: brief;
        %opt_parameter: detail;
        %opt_parameter: extensive;
}
show vif $(interfaces.interface.*.vif.*) {
        %command: show_vif;
        %module: fea;
        %opt_parameter: brief;
        %opt_parameter: detail;
        %opt_parameter: extensive;
}
```

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

The "show interface" 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 to be executed to implement this command - the script should return human-readable output preceded by a MIME content type indicating whether the text is structured or not[2]. The entire command line typed by the user is passed into the command. Thus the xorpsh might invoke the `show_interface` command using the Unix command line:

```
show_interface show interface xl0 brief
```

The command `%module` indicates that this command should only be available through the CLI when the router configuration has required that the named module has been started.

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.

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