An Introduction to Writing a XORP Process

Version 1.5

XORP, Inc.
http://www.xorp.org/
feedback@xorp.org

July 22, 2008

Contents

1 Introduction

This document is intented for a developer who wishes to write a XORP process, but doesn't know where to start. We'll walk through a simple XORP process, discussing how to define and use XRL interfaces, and how the bits fit together.

This is a first pass at such a document. We're bound to have missed things that are not obvious when you're starting out. Please provide us feedback as to how much help this document is; what really helped, what's missing, and what isn't explained properly.

We'll assume that you have copies of four other XORP design documents:

- XORP Design Overview[1]
- XORP Libxorp Library Overview[3]
- XORP Inter-Process Communication Library Overview[2]
- XRL Interfaces: Specifications and Tools[4]

These are available from the XORP web server. You should probably have read these through quickly so you're aware what additional information is available before reading this document further. It's recommended to read them in the order above.

We will assume you are familiar with what an XRL request is, the overall structure of the processes on a XORP router, and with C++.

2 Overview

In this document we'll work through by example the structure of a simple XORP process. We've chosen the *static_routes* process as an example. At the time of writing, this document is in sync with the source code for static_routes, but this is not guaranteed to always be the case.

static_routes is a very simple XORP process. To a first approximation, it receives XRL configuration requests from the *xorp_rtrmgr* to set up static routing entries, stores the entries, and communicates them to the RIB using XRLs.

This makes it a good example, because it exports an XRL interface to other processes (typically the xorp_rtrmgr) and calls XRLs on the XRL interface of another XORP process (the RIB). But it doesn't do all that much else, so there are few files and the code is quite readable.

The source code for the *static_routes* process is found in the `xorp/static_routes` subdirectory of the XORP source tree.

We'll walk through the main pieces of static_routes in the following order:

- The XRL interface of static_routes.
- Implementing the XRL interface of static_routes.
- The main loop of static routes.
- Calling XRLs on the RIB.
- Compiling the source code

3 The XRL Interface of *static_routes*

XRL interfaces are defined by a `.xif` file (pronounced *dot-ziff*). `xif` stands for XRL InterFace. All `.xif` files reside in `xorp/xrl/interfaces`.

The relevant file for us is `xorp/xrl/interfaces/static_routes.xif`. The first part of this file is shown in Listing 1.

Listing 1: The start of `xorp/xrl/interfaces/static_routes.xif`

```
/*
 * Static Routes XRL interface.
 */

interface static_routes/0.1 {

        /**
         * Enable/disable/start/stop StaticRoutes.
         *
         * @param enable if true, then enable StaticRoutes, otherwise
         * disable it.
         */
        enable_static_routes    ? enable:bool
        start_static_routes
        stop_static_routes

        /**
         * Add/replace/delete a static route.
         *
         * @param unicast if true, then the route would be used for unicast
         * routing.
         * @param multicast if true, then the route would be used in the
         * MRIB (Multicast Routing Information Base) for multicast purpose
         * (e.g., computing the Reverse-Path Forwarding information).
         * @param network the network address prefix this route applies to.
         * @param nexthop the address of the next-hop router for this route.
         * @param metric the metric distance for this route.
         */
        add_route4      ? unicast:bool & multicast:bool & network:ipv4net \
                        & nexthop:ipv4 & metric:u32

        add_route6      ? unicast:bool & multicast:bool & network:ipv6net \
                        & nexthop:ipv6 & metric:u32

...
}
```

The file `static_routes.xif` defines all the XRLs that are part of the `static_routes` XRL interface. These are XRLs that other processes can call on the *static_routes* process.

The format of the file is basically the keyword `interface` followed by the name and version of this particular interface, followed by a list of XRLs. In this case the name of the interface is `static_routes`, but this does not have to be the same as the name of the process. The version number is `0.1`. Version numbers are generally increased when a change is made that is not backwards compatible, but the precise value has no important meaning.

The list of XRLs is demarked by braces { ... }, and one XRL is given per line. Blank lines and comments are allowed, and a backslash before the newline can be used to split a long XRL over multiple lines to aid readability.

Thus the first XRL in this file is:
`static_routes/0.1/enable_static_routes?enable:bool`

When this XRL is actually called, it would look like:
`finder://static_routes/static_routes/0.1/enable_static_routes?enable:bool=true`

The `finder` part indicates that the XRL is an abstract one - we don't yet know what the transport parameters are. The first `static_routes` indicates the name of the target process, and the second `static_routes` is the name of the interface, taken from the XIF file. A process can support more than one interface, and an interface definition can be used by more than one process, hence the duplication in a process as simple as *static_routes*.

4 Using the `static_routes` XRL Interface

Now we have seen how the XRLs comprising the static_routes interface are defined, we shall examine how processes actually use them. For any particular interface, there are two types of user:

- The process that calls the XRLs and gets back responses. This is called the XRL *caller*.
- The process on which the XRL is called, and which generates responses. This is called the XRL *target*.

XORP provides scripts which can generate C++ code to make life much easier for both these parties.

4.1 Generating stub code for the caller

If we examine the file `Makefile.am` (the automake Makefile) in `xorp/xrl/interfaces`, we find the fragment in Listing 2.

Listing 2: Fragment from `xorp/xrl/interfaces/Makefile.am`

```
################################################################################
# Client Interface related
################################################################################

# BGP MIB traps
noinst_LTLIBRARIES = libbgpmibtrapsxif.la
libbgpmibtrapsxif_la_SOURCES = bgp_mib_traps_xif.hh bgp_mib_traps_xif.cc

...

# StaticRoutes Interface
noinst_LTLIBRARIES += libstaticroutesxif.la
libstaticroutesxif_la_SOURCES = static_routes_xif.hh static_routes_xif.cc

...

################################################################################
# Static Pattern Rules
################################################################################

SCRIPT_DIR=$(top_srcdir)/xrl/scripts
CLNTGEN_PY=$(SCRIPT_DIR)/clnt-gen

@PYTHON_BUILD@%_xif.cc %_xif.hh $(srcdir)/%_xif.hh $(srcdir)/%_xif.cc: \
         $(srcdir)/%.xif $(CLNTGEN_PY)
@PYTHON_BUILD@  $(PYTHON) $(CLNTGEN_PY) $<
```

This adds `libstaticroutesxif.la` to the list of libraries that should be built, and indicates that the source files for this library are `static_routes_xif.hh` and `static_routes_xif.cc`

The last part is pretty cryptic, but basically is a generic rule that says that files ending with `_xif.cc` and `_xif.hh` will be generated from files ending with `.xif` using the python script called `clnt-gen`.

So what actually happens here is that the file `static_routes.xif` is processed by `clnt-gen` to produce `static_routes_xif.hh` and `static_routes_xif.cc`, which are then compiled and linked into the library `libstaticroutesxif.la`. Any process that wants to call the *static_routes* interface can link with this library.

So what functionality does this library provide? Listing 3 shows a fragment from the machine-generated file `static_routes_xif.hh`. Between them, `static_routes_xif.hh` and `static_routes_xif.cc` define the machine-generated class `XrlStaticRoutesV0p1Client` and its complete implementation.

Listing 3: Fragment from `xorp/xrl/interfaces/static_routes_xif.hh`

```
class XrlStaticRoutesV0p1Client {
public:
    XrlStaticRoutesV0p1Client(XrlSender* s) : _sender(s) {}
    virtual ~XrlStaticRoutesV0p1Client() {}

...

    typedef XorpCallback1<void, const XrlError&>::RefPtr AddRoute4CB;
    /**
     *  Send Xrl intended to:
     *
     *  Add/replace/delete a static route.
     *
     *  @param dst_xrl_target_name the Xrl target name of the destination.
     *
     *  @param unicast if true, then the route would be used for unicast
     *  routing.
     *
     *  @param multicast if true, then the route would be used in the MRIB
     *  (Multicast Routing Information Base) for multicast purpose (e.g.,
     *  computing the Reverse-Path Forwarding information).
     *
     *  @param network the network address prefix this route applies to.
     *
     *  @param nexthop the address of the next-hop router for this route.
     *
     *  @param metric the metric distance for this route.
     */
    bool send_add_route4(
        const char*     dst_xrl_target_name,
        const bool&     unicast,
        const bool&     multicast,
        const IPv4Net&  network,
        const IPv4&     nexthop,
        const uint32_t& metric,
        const AddRoute4CB&      cb
    );

...

}
```

The constructor for `XrlStaticRoutesV0p1Client` takes a pointer to an `XrlSender` as its parameter. Typically this is actually an `XrlRouter` - we'll come to this in more detail later.

Then for every XRL defined in `static_routes.xif` there is a method to be called on an instance of `XrlStaticRoutesV0p1Client`. The example we'll look at here is `send_add_route4()`, although there are many more methods defined in `static_routes.xif`.

If you compare the method `send_add_route4()` in Listing 3 with the XRL `add_route4` in Listing 1, it should be pretty clear where this comes from. Basically, when you call `XrlStaticRoutesV0p1Client::send_add_route4` with all the parameters (`unicast`, `nexthop`, etc). set appropriately, the XRL `add_route4` will be called. You don't need to concern yourself with how the parameters are marshalled into the right syntax for the XRL, or how the XRL is actually transmitted, or even how the target process is discovered. But you do need to set the `target_name` parameter to the same thing that the `static_routes` process sets it to, otherwise the XRL *finder* won't be able to route your XRL to its destination. Often the target name will be the same as the name of the process - in this case *static_routes*- but if there are multiple instances of the interface then you'll need to figure out which target name to use.

You'll also notice that some of the parameters for XRL functions are not native C++ types. In this case, `network` is of type `IPv4Net` and `nexthop` is of type `IPv4`. Classes instantiating the these additional types are found in `libxorp` and are used throughout XORP.

The final parameter is `const AddRoute4CB& cb`.
Earlier in the Listing we can see that this is defined as:
`typedef XorpCallback1<void, const XrlError&>::RefPtr AddRoute4CB;`
This defines `AddRoute4CB` to be a *callback* which returns type `void` with one parameter of type `const XrlError&`.

But what exactly is a `callback`?

Well, what we want is to call the `send_add_route4()` method to send an XRL request to the *static_routes* process, and then to go off and do other things while we're waiting for the response to come back. In a multi-threaded architecture, this might be achieved by having `send_add_route4()` block until the response is ready, but XORP is deliberately *not* a multi-threaded architecture. Thus what happens is that `send_add_route4()` will return immediately. It will return false if a local error occurs, but will normally return true before the XRL has actually been sent. Some time later the response will come back from the *static_routes* process, and we need a way to direct the response to the right class instance that is expecting it. This is achieved in XORP through the use of *callbacks*.

A callback is created using the `callback()` function from libxorp. We'll discuss this in more detail when we look at how the *static_routes* process sends changes to the RIB in Section 6. For now, it suffices to say that a callback must be created and passed into `send_add_route4()`, and that this is how the response from the `add_route4()` XRL is returned to the right place.

4.2 Generating stub code for the target

The other side to the XRL story is how the XRL target implements the XRLs. To illustrate this, we will look at how the *static_routes* process implements the XRL interface defined in `static_routes.xif`. A XORP process can implement more than one interface. In fact most XORP processes implement a special-purpose interface and also the `common` interface, which provides XRLs to query basic version and status information about a target process.

To see what interfaces a particular target process supports we must look in the `xorp/xrl/targets` directory. Listing 4 shows the entire contents of `static_routes.tgt`. This file defines that the XRL target called `static_routes` implements the two interfaces `common/0.1` and `static_routes/0.1`.

In the *static_routes* process, we'd prefer not to have to write all the code to unmarshall XRLs into C++, and marshall the response back into an XRL response, so again we use machine-generated C++ stubs to free the programmer from having to do most of the tedious work. Listing 5 shows a number of fragments from `xorp/xrl/targets/Makefile.am` related to the `static_routes` target.

In Listing 5, the first important point is that `static_routes.tgt` is added to the list of `tgt_files`. From each `.tgt` file, a `.xrls` file will be generated using the python script `tgt-gen` according to the magic at the bottom of the listing.

In the case of `static_routes.tgt`, the file `static_routes.xrls` will be generated. This file simply contains a listing of all the fully expanded XRLs supported by the `static_routes` XRL target.

The next important point to note from Listing 5 is that we have specified that we want to build a library called `libstaticroutesbase.la`. This is going to be the library that the *static_routes* process links with to get access to all the stub code to implement the target part of this interface.

Finally there's the directive to build `libstaticroutesbase.la` from the machine-generated source files `static_routes_base.hh` and `static_routes_base.cc`, and that these files depend on the files `common.xif` and `static_routes.xif`.

So, what does `libstaticroutesbase.la` actually provide? Listing 6 shows some extracts from `static_routes_base.hh`. Basically `libstaticroutesbase.la` defines a class called `XrlStaticRoutesTargetBase` which will be used to receive XRL requests.

The constructor for `XrlStaticRoutesTargetBase` takes a single parameter which is typically the `XrlRouter` for the process. An `XrlRouter` is an object that is bound to an *EventLoop* and which sends and receives XRL requests. Each process has its own *EventLoop*. In Section 5 we'll look at what the EventLoop does. In any event, once an instance of `XrlStaticRoutesTargetBase` has been created with a pointer to a working `XrlRouter`, then the process is ready to receive XRL requests for the `static_routes` interface. But first we have to actually write some code.

If we look in at Listing 6, we see that the method `static_routes_0_1_add_route4()` has been defined. However the method is a *pure virtual*, which means that it is defined here, but there is no implementation of this in `XrlStaticRoutesTargetBase`. So how do we actually make use of this?

The general idea is that the stub generation code knows the syntax for this target interface, so it generates all the code needed to check that incoming requests match the defined syntax and handle errors if they don't. But the stub generation code has no idea what this interface actually *does*. We need to supply an implemenation for `static_routes_0_1_add_route4()` that actually does what we want when this XRL is called.

Listing 4: Contents of `xorp/xrl/targets/static_routes.tgt`

```
#include "common.xif"
#include "finder_event_observer.xif"
#include "policy_backend.xif"
#include "static_routes.xif"

target static_routes implements common/0.1,                              \
                                finder_event_observer/0.1,                \
                                policy_backend/0.1,                       \
                                static_routes/0.1
```

Listing 5: Extracts from `xorp/xrl/targets/Makefile.am`

```
################################################################################
# Xrl Target related
################################################################################

# Add your target file here
tgt_files               = bgp.tgt
...
tgt_files              += static_routes.tgt

...

# Automatically compute the list of the *.xrls files
xrls_files              = $(tgt_files:%.tgt=%.xrls)

...

# Add your target's library here
noinst_LTLIBRARIES       = libbgpbase.la
noinst_LIBRARIES         = libbgpbase.a
...
noinst_LTLIBRARIES      += libstaticroutesbase.la

...

# StaticRoutes
libstaticroutesbase_la_SOURCES = static_routes_base.hh static_routes_base.cc
$(srcdir)/static_routes_base.hh $(srcdir)/static_routes_base.cc:        \
                $(INTERFACES_DIR)/common.xif                            \
                $(INTERFACES_DIR)/finder_event_observer.xif             \
                $(INTERFACES_DIR)/policy_backend.xif                    \
                $(INTERFACES_DIR)/static_routes.xif

...

################################################################################
# Implicit Rules and related
################################################################################
```

```
SCRIPT_DIR=$(top_srcdir)/xrl/scripts
TGTGEN_PY=$(SCRIPT_DIR)/tgt-gen

# If this code is commented out, please upgrade to python2.0 or above.

@PYTHON_BUILD@$(srcdir)/%_base.hh $(srcdir)/%_base.cc %_base.hh %_base.cc   \
@PYTHON_BUILD@$(srcdir)/%.xrls: $(srcdir)/%.tgt $(TGTGEN_PY)
@PYTHON_BUILD@      $(PYTHON) $(TGTGEN_PY) -I$(INTERFACES_DIR) $<
```

Listing 6: Extracts from xorp/xrl/targets/static_routes_base.hh

```
class XrlStaticRoutesTargetBase {

...

public:
    /**
     * Constructor.
     *
     * @param cmds an XrlCmdMap that the commands associated with the target
     *             should be added to.  This is typically the XrlRouter
     *             associated with the target.
     */
    XrlStaticRoutesTargetBase(XrlCmdMap* cmds = 0);

...

protected:

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Add/replace/delete a static route.
     *
     *  @param unicast if true, then the route would be used for unicast
     *  routing.
     *
     *  @param multicast if true, then the route would be used in the MRIB
     *  (Multicast Routing Information Base) for multicast purpose (e.g.,
     *  computing the Reverse-Path Forwarding information).
     *
     *  @param network the network address prefix this route applies to.
     *
     *  @param nexthop the address of the next-hop router for this route.
     *
     *  @param metric the metric distance for this route.
     */
    virtual XrlCmdError static_routes_0_1_add_route4(
        // Input values,
        const bool&      unicast,
        const bool&      multicast,
        const IPv4Net&   network,
        const IPv4&      nexthop,
        const uint32_t&  metric) = 0;

...

}
```

Listing 7: Extracts from xorp/static_routes/xrl_static_routes_node.hh

```
class XrlStaticRoutesNode : public StaticRoutesNode,
                            public XrlStdRouter,
                            public XrlStaticRoutesTargetBase {
public:
    XrlStaticRoutesNode(EventLoop&      eventloop,
                        const string&   class_name,
                        const string&   finder_hostname,
                        uint16_t        finder_port,
                        const string&   finder_target,
                        const string&   fea_target,
                        const string&   rib_target);

...

protected:
    //
    // XRL target methods
    //

...

    XrlCmdError static_routes_0_1_add_route4(
        // Input values,
        const bool&     unicast,
        const bool&     multicast,
        const IPv4Net&  network,
        const IPv4&     nexthop,
        const uint32_t& metric);

...

private:
...
    XrlRibV0p1Client    _xrl_rib_client;
...
}
```

So now we come at last to the implementation of the *static_routes* process. This is in the xorp/static_routes directory.

We have created a file called xrl_static_routes_node.hh to define our class that actually implements the code to receive and process XRLs. An extract from this is shown in Listing 7. We have defined our own class called XrlStaticRoutesNode which is a child class of StaticRoutesNode, XrlStdRouter and XrlStaticRoutesTargetBase classes. We'll ignore the StaticRoutesNode class in this explanation, because it's specific to the *static_routes* process, but the important thing is that XrlStaticRoutesNode is a child of the XrlStaticRoutesTargetBase base class that was generated by the stub compiler, and a child of the XrlStdRouter base class.

The constructor for our XrlStaticRoutesNode class takes a number of parameters which are specific to this particular implementation, but it also takes a number of parameters that are used in the constructor of the XrlStdRouter base class.

We also see from Listing 7 that our `XrlStaticRoutesNode` class is going to implement the `static_routes_0_1_add_route4()` method from the stub compiler which was a pure virtual method in the base class.

Listing 8: Extracts from `xorp/static_routes/xrl_static_routes_node.cc`

```
...
#include "static_routes_node.hh"
#include "xrl_static_routes_node.hh"

...

XrlStaticRoutesNode::XrlStaticRoutesNode(EventLoop&     eventloop,
                                         const string&  class_name,
                                         const string&  finder_hostname,
                                         uint16_t       finder_port,
                                         const string&  finder_target,
                                         const string&  fea_target,
                                         const string&  rib_target)
    : StaticRoutesNode(eventloop),
      XrlStdRouter(eventloop, class_name.c_str(), finder_hostname.c_str(),
                   finder_port),
      XrlStaticRoutesTargetBase(&xrl_router()),
      ...
     _xrl_rib_client(&xrl_router()),
      ...
{
  ...
}

...

XrlCmdError
XrlStaticRoutesNode::static_routes_0_1_add_route4(
    // Input values,
    const bool&         unicast,
    const bool&         multicast,
    const IPv4Net&      network,
    const IPv4&         nexthop,
    const uint32_t&     metric)
{
    string error_msg;

    if (StaticRoutesNode::add_route4(unicast, multicast, network, nexthop,
                                     "", "", metric, error_msg)
        != XORP_OK) {
        return XrlCmdError::COMMAND_FAILED(error_msg);
    }

    return XrlCmdError::OKAY();
}
```

In Listing 8, we see an extract from `xorp/static_routes/xrl_static_routes_node.cc` where we have actually implemented the `XrlStaticRoutesNode` class.

The constructor for `XrlStaticRoutesNode` passes a number of arguments to the `XrlStdRouter` base class, and then passes to the constructor for the `XrlStaticRoutesTargetBase` base class a pointer to this `XrlStdRouter` base class (the return result for method `xrl_router()`). In addition, it initializes a lot of its own state (not shown). Note that if we were implementing a module that does not receive any XRLs (*i.e.,* it won't use the equivalent of `XrlStaticRoutesTargetBase`), then we must call `XrlStdRouter::finalize()` after `XrlStdRouter` has been created.

The complete implementation of `XrlStaticRoutesNode::static_routes_0_1_add_route4()` is shown. In this case, most of the actual work is done elsewhere, but the general idea is clear. This is where we actually receive and process the incoming XRL request.

Once we have processed the request, we need to return from this method. If this XRL had actually taken any return values, there would have been parameters to the `static_routes_0_1_add_route4` method that were not `const` references, and we would simply have set the values of these variables before calling `return` to pass the values back to the XRL caller. In the case of *static_routes* however, none of the XRLs return any values other than success or failure. We return `XrlCmdError::OKAY()` if all is well, or `XrlCmdError::COMMAND_FAILED(error_msg)` if something went seriously wrong, passing back a human-readable string for diagnostic purposes.

In general, if an error response needs to return machine-readable error information, it is often better to return `XrlCmdError::OKAY()` together with return parameters to indicate that an error occurred and what actually happened, because if `COMMAND_FAILED` is returned, the return parameter values are not passed up to the caller application.

5 The Main Loop

So far we've looked at how to define an XRL interface, how to compile the C++ stubs for that interface, and how to define the actual code that implements that interface. Now we need to look at the main loop of a XORP process to see how these pieces all come together.

In Listing 10 the main pieces of `xorp/static_routes/xorp_static_routes.cc` are shown. These comprise the entire initialization part and main loop of our *static_routes* process.

First come the #includes. Convention indicates that the first of these (`static_routes_module.h`) is a header file defining the module name and version - this information is used by later includes which will complain if this information is not available. The content of `static_routes_module.h` is very simple. It must define `XORP_MODULE_NAME` and `XORP_MODULE_VERSION`:

Listing 9: Listing of `xorp/static_routes/static_routes_module.h`

```
#ifndef XORP_MODULE_NAME
#define XORP_MODULE_NAME        "STATIC_ROUTES"
#endif
#ifndef XORP_MODULE_VERSION
#define XORP_MODULE_VERSION     "0.1"
#endif
```

Then we include the functionality from `libxorp` that we'll need:

- `libxorp/xorp.h`: generic headers that should always be included.
- `libxorp/xlog.h`: XORP logging functionality. The convention is to use XLOG macros to log warnings and error messages, so we can redefine how logging if implemented in future without re-writing the code that uses logging. See Section 8 for more information about the XLOG facility.
- `libxorp/debug.h`: XORP debugging functionality.
- `libxorp/callback.hh`: XORP callback templates, needed to pass a handle into event handling code to be called later when an event occurs.
- `libxorp/eventloop.hh`: the main XORP eventloop.
- `libxorp/exceptions.hh`: standard exceptions for standard stuff - useful as a debugging aid.

Finally we include the definition of the class that implements the `static_routes` XRL interface target class we just defined.

In the processes `main()` function, we intialize the `xlog` logging functionality. Then (not shown) we handle command line arguments.

The main part of this process occurs within a single `try/catch` statement. The `catch` part then handles any of the xorp standard exceptions that might be thrown. It is not intended that any unhandled exceptions actually get this far, but if they do, then `xorp_catch_standard_exceptions()` will ensure that appropriate diagnostic information is available when the process expires. This is not required, but it is good coding practice.

The actual main loop that does all the work is in `static_routes_main()`.

First, the `EventLoop` is created. Every XORP process should have precisely one `EventLoop`. All processing in a XORP process is event-driven from the eventloop. When the process is idle, it will be blocked in `EventLoop::run()`. When an XRL request arrives, or an XRL response arrives, or a timer expires, or activity occurs on a registered file handle, then an event handler will be called from the eventloop.

Next we create an `XrlStdRouter`. This is the object that will be used to send and receive XRLs from this process. We pass it the `EventLoop` object, information about the host and port where the XRL finder is located, and the XRL target name of this process: in this case `"static_routes"`.

Then we create an instance of the `XrlStaticRoutesNode` class we defined earlier to receive XRLs on the `static_routes` XRL target interface. Inside this object there will be the corresponding `XrlStdRouter` object for sending and receiving XRLs from this process. We pass to `XrlStaticRoutesNode` the following:

- The `EventLoop` object.
- The XRL target name of this process: in this case `"static_routes"`.
- Information about the host and port where the XRL finder is located.
- Information about the names of other XRL targets we need to communicate with: the Finder, the FEA, and the RIB.

Before we proceed any further, we must give the XrlStdRouter time to register our existence with the Finder. Thus we call `wait_until_xrl_router_is_ready()`.

Listing 10: Extracts from xorp/static_routes/xorp_static_routes.cc

```
//
// XORP StaticRoutes module implementation.
//

#include "static_routes_module.h"

#include "libxorp/xorp.h"
#include "libxorp/xlog.h"
#include "libxorp/debug.h"
#include "libxorp/callback.hh"
#include "libxorp/eventloop.hh"
#include "libxorp/exceptions.hh"

#include "xrl_static_routes_node.hh"

...

static void
static_routes_main(const string& finder_hostname, uint16_t finder_port)
{
    //
    // Init stuff
    //
    EventLoop eventloop;

    //
    // StaticRoutes node
    //
    XrlStaticRoutesNode xrl_static_routes_node(
        eventloop,
        "static_routes",
        finder_hostname,
        finder_port,
        "finder",
        "fea",
        "rib");
    wait_until_xrl_router_is_ready(eventloop,
                                   xrl_static_routes_node.xrl_router());

    // Startup
    xrl_static_routes_node.startup();

    //
    // Main loop
    //
    while (! xrl_static_routes_node.is_done()) {
        eventloop.run();
    }
}
```

```
int
main(int argc, char *argv[])
{
    int ch;
    string::size_type idx;
    const char *argv0 = argv[0];
    string finder_hostname = FinderConstants::FINDER_DEFAULT_HOST().str();
    uint16_t finder_port = FinderConstants::FINDER_DEFAULT_PORT();

    //
    // Initialize and start xlog
    //
    xlog_init(argv[0], NULL);
    xlog_set_verbose(XLOG_VERBOSE_LOW);         // Least verbose messages
    // XXX: verbosity of the error messages temporary increased
    xlog_level_set_verbose(XLOG_LEVEL_ERROR, XLOG_VERBOSE_HIGH);
    xlog_add_default_output();
    xlog_start();

...

    //
    // Run everything
    //
    try {
        static_routes_main(finder_hostname, finder_port);
    } catch(...) {
        xorp_catch_standard_exceptions();
    }

    //
    // Gracefully stop and exit xlog
    //
    xlog_stop();
    xlog_exit();

    exit (0);
}
```

Finally we're ready to go. We set our internal state as ready, and enter a tight loop that we will only exit when it is time to terminate this process. At the core of this loop, we call `EventLoop::run()` repeatedly. `run()` will block when there are no events to process. When an event is ready to process, the relevant event handler will be called, either directly via a `callback` or indirectly through one of the XRL stub handler methods we defined earlier. Thus if another process calls the `finder://static_routes/static_routes/0.1/add_route4` XRL, the first we'll know about it is when `XrlStaticRoutesNode::static_routes_0_1_add_route4()` is executed.

6 Calling XRLs on the RIB

So far we have seen how we define an XRL interface, how we implement the target side of such an interface, and how the main loop of a XORP process is structured. In the case of *static_routes*, we can now receive XRLs informing us of routes. The *static_routes* process will do some checks and internal processing on these routes (such as checking that they go out over a network interface that is currently up). Finally it will communicate the remaining routes to the RIB process for use by the forwarding plane. We will now examine how we send these routes to the RIB.

If we look in `xorp/xrl/interfaces` we find the file `rib.xif` which defines the XRLs available on the `rib` interface. Listing 11 shows some extracts from this file. As we've been following through the `add_route4` XRL, we'll again look at that here. We'll also look at the `lookup_route_by_dest4` XRL because this is an example of an XRL that returns some data, although this particular XRL is not actually used by the *static_routes* process. It is also worth noting in passing that the RIB requires a routing protocol (such as *static_routes*) to call `add_igp_table4` before sending routes to the RIB, or the RIB will not know what to do with the routes.

As we saw with the `static_routes.xif` file, the `rib.xif` file is processed by a python script to produce the files `rib_xif.hh` and `rib_xif.cc` in the `xorp/xrl/interfaces` directory which are then compiled and linked to produce the `libribxif.la` library. This library provides a class definition which does all the work of marshalling C++ arguments into XRLs, sending the XRL to the RIB process, receiving the response, and calling the relevant callback in the caller process with the response data.

Listing 12 shows some extracts from `rib_xif.hh` so we can see what the C++ interface to this library looks like. The library implements a class called `XrlRibV0p1Client`. To use this code, we must first create an instance of this class, calling the constructor and supplying a pointer to an `XrlSender`. Typically such an XrlSender is an instance of an `XrlRouter` object.

In Listing 7 we can see that our implementation of class `XrlStaticRoutesNode` actually defined an instance of `XrlRibV0p1Client` called `_xrl_rib_client` as a member variable, so this object is created automatically when our main loop creates `xrl_static_routes_node` in Listing 10. In Listing 8 we can see that we passed `xrl_router` into the constructor for `_xrl_rib_client`.

So, once everything else has been initialized, we'll have access to `_xrl_rib_client` from within `xrl_static_routes_node`. Now, how do we make use of this generated code? The answer is simple: to send a route to the RIB we simply call `_xrl_rib_client.send_add_route4()` with the appropriate parameters, and the generated library code will do the rest. We can see this in Listing 13, where this code is actually used.

The only real complication here is related to how we get the response back from the XRL. Recall that `_xrl_rib_client.send_add_route4()` will return immediately with a local success or failure response, before the XRL has actually been transmitted to the RIB. Thus we need to pass a *callback* in to `send_add_route4()`. This callback will wrap up enough state so that when the response finally returns to the XrlRouter in the *static_routes* process, it will know which method to call on which object with which parameters so as to send the response to the right place.

We can see in `XrlRibV0p1Client` (Listing 7) that the type of the callback is:

```
XorpCallback1<void, const XrlError&>::RefPtr
```

This defines a callback function that returns `void` and which takes one argument of type `const XrlError&`. If we look in Listing 13 we seen that the method `XrlStaticRoutesNode::send_rib_route_change_cb()` fits exactly these criteria. This is the method we are going to use to receive the response from our XRL request.

Listing 11: Extracts from xorp/xrl/interfaces/rib.xif

```
interface rib/0.1 {
...
        /**
         * Add/delete an IGP or EGP table.
         *
         * @param protocol the name of the protocol.
         * @param target_class the target class of the protocol.
         * @param target_instance the target instance of the protocol.
         * @param unicast true if the table is for the unicast RIB.
         * @param multicast true if the table is for the multicast RIB.
         */
        add_igp_table4          ? protocol:txt                          \
                                & target_class:txt & target_instance:txt\
                                & unicast:bool & multicast:bool
...
        /**
         * Add/replace/delete a route.
         *
         * @param protocol the name of the protocol this route comes from.
         * @param unicast true if the route is for the unicast RIB.
         * @param multicast true if the route is for the multicast RIB.
         * @param network the network address prefix of the route.
         * @param nexthop the address of the next-hop router toward the
         * destination.
         * @param metric the routing metric.
         * @param policytags a set of policy tags used for redistribution.
         */
        add_route4      ? protocol:txt & unicast:bool & multicast:bool  \
                        & network:ipv4net & nexthop:ipv4 & metric:u32   \
                        & policytags:list
        replace_route4  ? protocol:txt & unicast:bool & multicast:bool  \
                        & network:ipv4net & nexthop:ipv4 & metric:u32   \
                        & policytags:list
        delete_route4   ? protocol:txt & unicast:bool & multicast:bool  \
                        & network:ipv4net

...
        /**
         * Lookup nexthop.
         *
         * @param addr address to lookup.
         * @param unicast look in unicast RIB.
         * @param multicast look in multicast RIB.
         * @param nexthop contains the resolved nexthop if successful,
         * IPv4::ZERO otherwise.  It is an error for the unicast and multicast
         * fields to both be true or both false.
         */
        lookup_route_by_dest4 ? addr:ipv4 & unicast:bool & multicast:bool \
                -> nexthop:ipv4
}
```

Listing 12: Extracts from `xorp/xrl/interfaces/rib_xif.hh`

```
class XrlRibV0p1Client {
public:
    XrlRibV0p1Client(XrlSender* s) : _sender(s) {}
...
    typedef XorpCallback1<void, const XrlError&>::RefPtr AddRoute4CB;
    /**
     *  Send Xrl intended to:
     *
     *  Add/replace/delete a route.
     *
     *  @param dst_xrl_target_name the Xrl target name of the destination.
     *  @param protocol the name of the protocol this route comes from.
     *  @param unicast true if the route is for the unicast RIB.
     *  @param multicast true if the route is for the multicast RIB.
     *  @param network the network address prefix of the route.
     *  @param nexthop the address of the next-hop router toward the
     *  destination.
     *  @param metric the routing metric.
     *  @param policytags a set of policy tags used for redistribution.
     */
    bool send_add_route4(
        const char*     dst_xrl_target_name,
        const string&   protocol,
        const bool&     unicast,
        const bool&     multicast,
        const IPv4Net&  network,
        const IPv4&     nexthop,
        const uint32_t& metric,
        const XrlAtomList&      policytags,
        const AddRoute4CB&      cb
    );
...
    typedef XorpCallback2<void, const XrlError&, const IPv4*>::RefPtr LookupRo
uteByDest4CB;
    /**
     *  Send Xrl intended to:
     *
     *  Lookup nexthop.
     *
     *  @param dst_xrl_target_name the Xrl target name of the destination.
     *  @param addr address to lookup.
     *  @param unicast look in unicast RIB.
     *  @param multicast look in multicast RIB.
     */
    bool send_lookup_route_by_dest4(
        const char*     dst_xrl_target_name,
        const IPv4&     addr,
        const bool&     unicast,
        const bool&     multicast,
        const LookupRouteByDest4CB&   cb
```

```
    );
}
```

Listing 13: Extracts from xorp/static_routes/xrl_static_routes_node.cc

```
void
XrlStaticRoutesNode::send_rib_route_change()
{
    bool success = true;
...
    StaticRoute& static_route = _inform_rib_queue.front();
...
    //
    // Send the appropriate XRL
    //
    if (static_route.is_add_route()) {
        if (static_route.is_ipv4()) {
            if (static_route.is_interface_route()) {
...
            } else {
                success = _xrl_rib_client.send_add_route4(
                    _rib_target.c_str(),
                    StaticRoutesNode::protocol_name(),
                    static_route.unicast(),
                    static_route.multicast(),
                    static_route.network().get_ipv4net(),
                    static_route.nexthop().get_ipv4(),
                    static_route.metric(),
                    static_route.policytags().xrl_atomlist(),
                    callback(this, &XrlStaticRoutesNode::send_rib_route_change_cb));
                if (success)
                    return;
            }
        }
...
}

void
XrlStaticRoutesNode::send_rib_route_change_cb(const XrlError& xrl_error)
{
    switch (xrl_error.error_code()) {
    case OKAY:
        //
        // If success, then send the next route change
        //
        _inform_rib_queue.pop_front();
        send_rib_route_change();
        break;

    case COMMAND_FAILED:
        //
        // If a command failed because the other side rejected it,
        // then print an error and send the next one.
        //
        ...
```

```
        break;

    case NO_FINDER:
    case RESOLVE_FAILED:
    case SEND_FAILED:
        ...
        break;

    case BAD_ARGS:
    case NO_SUCH_METHOD:
    case INTERNAL_ERROR:
        ...
        break;

    case REPLY_TIMED_OUT:
    case SEND_FAILED_TRANSIENT:
        ...
        break;
    }
}
```

We actually create the callback using the call:
`callback(this, &XrlStaticRoutesNode::send_rib_route_change_cb)`
In the context of Listing 13, `this` refers to a pointer to the current instance of `XrlStaticRoutesNode`. So, what this callback does is to wrap a pointer to the method `send_rib_route_change_cb()` on the current instance of `XrlStaticRoutesNode`. Later on, when the response returns, the XrlRouter will call the `send_rib_route_change_cb()` method on this specific instance of `XrlStaticRoutesNode` and supply it with a parameter of type `const XrlError&`.

In the implementation of `send_rib_route_change_cb()` we can see that we check the value of the `xrl_error` parameter to see whether the XRL call was actually successful or not. If the return error code is `OKAY` we send the next route change. Otherwise, we take different actions based on the error type.

6.1 Returning values in XRLs

Because the *static_routes* process is so simple, none of the XRLs it calls actually return any information in the response. However, it's rather common that we want to make a request of a target and get back some information. This is quite easy to do, but just requires a different callback that can receive the relevant parameters.

In Listing 11 we saw that the XRL `lookup_route_by_dest4` returns one value of type `ipv4` called `nexthop`. XRLs can actually return multiple parameters - this is merely a simple example.

In Listing 12 we can see that the callback we need to suuply to `send_lookup_route_by_dest4()` is of type:
`XorpCallback2<void, const XrlError&, const IPv4*>::RefPtr`
This is just like the callback we have already seen, except that the method the callback will call must take two arguments. The first must be of type `const XrlError&` and the second must be of type `const IPv4*`. Although *static_routes* has no such callback method, if it did it might look like the function `lookup_route_by_dest4_cb` in Listing 14. The callback itself to be passed into `send_lookup_route_by_dest4()` is created in exactly the same way as the one we passed into `send_add_route4()`.

Listing 14: Hypothetical callback for `send_lookup_route_by_dest4()`

```
void
XrlStaticRoutesNode::lookup_route_by_dest4_cb(const XrlError& xrl_error,
                                              const IPv4* nexthop)
{
    if (xrl_error == XrlError::OKAY()) {
        printf("the nexthop is %s\n", nexthop->str().c_str());
    }
...
}
```

7 Compiling the Source Code

If any of the Makefile.am or configure.in files are modified, then the *./bootstrap* script in the XORP top-level directory must be executed first. It will run the appropriate autotools to generate the corresponding Makefile.in files and the *configure* script. After that the *./configure* script must be run followed by *gmake*.

Note that XORP assumes certain versions of the autoconf/automake/libtool versions have been installed. Those versions are listed in the README in the top-level XORP directory. If the installed versions are different, then the result is unredicted so it is best to install versions that are as close as possible to those listed in the README.

The *bootstrap* script itself assumes that the autotools executable programs have certain names. E.g.,:

```
ACLOCAL=${ACLOCAL:-"aclocal110"}
AUTOCONF=${AUTOCONF:-"autoconf261"}
AUTOHEADER=${AUTOHEADER:-"autoheader261"}
AUTOMAKE=${AUTOMAKE:-"automake110"}
LIBTOOLIZE=${LIBTOOLIZE:-"libtoolize"}
```

If the default names don't match, then set the following variables in the shell environment to the appropriate names before running *./bootstrap*:

- *ACLOCAL*
- *AUTOCONF*
- *AUTOHEADER*
- *AUTOMAKE*
- *LIBTOOLIZE*

8 The XLOG Logging Facility

The XORP XLOG facility is used for log messages generation, similar to syslog. The log messages may be output to multiple output streams simultaneously. Below is a description of how to use the log utility.

- The xlog utility assumes that `XORP_MODULE_NAME` is defined (per module). To do so, you must have in your directory a file like "foo_module.h", and inside it should contain something like:

  ```
  #define XORP_MODULE_NAME "BGP"
  ```

  This file then has to be included by each *.c and *.cc file, and MUST be the first of the included local files.

- Before using the xlog utility, a program MUST initialize it first (think of this as the xlog constructor):

  ```
  int xlog_init(const char *process_name, const char *preamble_message);
  ```

  Further, if a program tries to use xlog without initializing it first, the program will exit.

- To add output streams, you MUST use one of the following (or both):

  ```
  int xlog_add_output(FILE* fp);
  int xlog_add_default_output(void);
  ```

- To change the verbosity of all xlog messages, use:

  ```
  xlog_set_verbose(xlog_verbose_t verbose_level);
  ```

  where "verbose_level" is one of the following (`XLOG_VERBOSE_MAX` excluded):

  ```
  typedef enum {
      XLOG_VERBOSE_LOW = 0,        /* 0 */
      XLOG_VERBOSE_MEDIUM,         /* 1 */
      XLOG_VERBOSE_HIGH,           /* 2 */
      XLOG_VERBOSE_MAX
  } xlog_verbose_t;
  ```

  Default value is `XLOG_VERBOSE_LOW` (least details). Larger value for "verbose_level" adds more details to the preamble message (e.g., file name, line number, etc, about the place where the log message was initiated).

  Note that the verbosity level of message type `XLOG_LEVEL_FATAL` (see below) cannot be changed and is always set to the most verbose level (`XLOG_VERBOSE_HIGH`).

- To change the verbosity of a particular message type, use:

```
void xlog_level_set_verbose(xlog_level_t log_level,
xlog_verbose_t verbose_level);
```

where "log_level" is one of the following (XLOG_LEVEL_MIN and XLOG_LEVEL_MAX excluded):

```
typedef enum {
    XLOG_LEVEL_MIN = 0,          /* 0 */
    XLOG_LEVEL_FATAL = 0,        /* 0 */
    XLOG_LEVEL_ERROR,            /* 1 */
    XLOG_LEVEL_WARNING,          /* 2 */
    XLOG_LEVEL_INFO,             /* 3 */
    XLOG_LEVEL_TRACE,            /* 4 */
    XLOG_LEVEL_MAX
} xlog_level_t;
```

Note that the verbosity level of message type XLOG_LEVEL_FATAL cannot be changed and is always set to the most verbose level (XLOG_VERBOSE_HIGH).

- To start the xlog utility, you MUST use:

```
int xlog_start(void);
```

- To enable or disable a particular message type, use:

```
int xlog_enable(xlog_level_t log_level);
int xlog_disable(xlog_level_t log_level);
```

By default, all levels are enabled. Note that XLOG_LEVEL_FATAL cannot be disabled.

- To stop the logging, use:

```
int xlog_stop(void);
```

Later you can restart it again by xlog_start()

- To gracefully exit the xlog utility, use

```
int     xlog_exit(void);
```

(think of this as the xlog destructor).

Listing 15 contains an example of using the XLOG facility.

Listing 15: An example of using the XLOG facility

```
int
main(int argc, char *argv[])
{
    //
    // Initialize and start xlog
    //
    xlog_init(argv[0], NULL);
    xlog_set_verbose(XLOG_VERBOSE_LOW); // Least verbose messages
    // Increase verbosity of the error messages
    xlog_level_set_verbose(XLOG_LEVEL_ERROR, XLOG_VERBOSE_HIGH);
    xlog_add_default_output();
    xlog_start();

    // Do something

    //
    // Gracefully stop and exit xlog
    //
    xlog_stop();
    xlog_exit();

    exit (0);
}
```

Typically, a developer would use the macros described below to print a message, add an assert statement, place a marker, etc. If a macro accepts a message to print, the format of the message is same as printf(3). The only difference is that the xlog utility automatically adds `'\n'`, (i.e. end-of-line) at the end of each string specified by `format`:

- `XLOG_FATAL(const char *format, ...)`
  Write a FATAL message to the xlog output streams and abort the program.
- `XLOG_ERROR(const char *format, ...)`
  Write an ERROR message to the xlog output streams.
- `XLOG_WARNING(const char *format, ...)`
  Write a WARNING message to the xlog output streams.
- `XLOG_INFO(const char *format, ...)`
  Write an INFO message to the xlog output streams.
- `XLOG_TRACE(int cond_boolean, const char *format, ...)`
  Write a TRACE message to the xlog output stream, but only if `cond_boolean` is not 0.
- `XLOG_ASSERT(assertion)`
  The XORP replacement for assert(3), except that it cannot be conditionally disabled and logs error messages through the standard xlog mechanism. It calls `XLOG_FATAL()` if the assertion fails.
- `XLOG_UNREACHABLE()`
  A marker that can be used to indicate code that should never be executed.

- `XLOG_UNFINISHED()`
  A marker that can be used to indicate code that is not yet implemented and hence should not be run.

9 The *rtrmgr* Template Files

TODO: add description how to write rtrmgr template files.

For the time being, the developer can check the "XORP Router Manager Process (rtrmgr)" document for information about the template semantics, and can use file `xorp/etc/templates/static_routes.tp` as an example.

A Modification History

- July 19, 2004: Initial version 1.0 completed.
- April 13, 2005: Updated to match XORP release 1.1: Added the XLOG logging facility section.
- March 8, 2006: Updated to match XORP release 1.2: Updated some of the sample source code.
- August 2, 2006: Updated to match XORP release 1.3: The XRL-related sample code is modified to match the original code. Miscellaneous cleanup.
- March 20, 2007: Updated to match XORP release 1.4: Updated some of the sample source code.
- July 18, 2007: Added Section 7: Compiling the Source Code.
- July 22, 2008: Updated to match XORP release 1.5: No significant changes.

References

[1] XORP Design Architecture. XORP technical document. http://www.xorp.org/.

[2] XORP Inter-Process Communication Library. XORP technical document. http://www.xorp.org/.

[3] XORP Libxorp Library Overview. XORP technical document. http://www.xorp.org/.

[4] XRL Interfaces: Specification and Tools. XORP technical document. http://www.xorp.org/.