A XORP module implementation: Coordinate System example

Bruno Willemaers

University Of Liège

March 2, 2010

Prerequisites

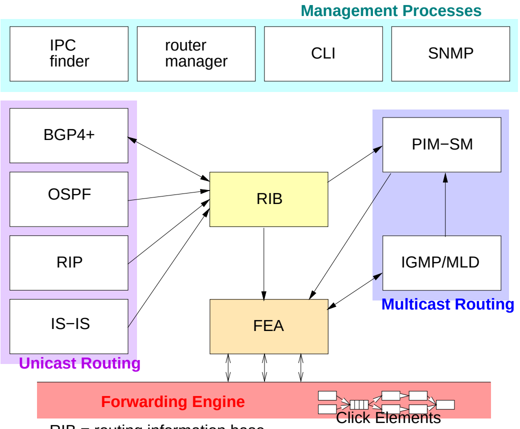
XORP (eXtensible Open Router Platform)

- is essentially written in C++.
- makes heavy use of
  - templates,
  - multiple inheritance.

Outline

Extensibility - Modularity

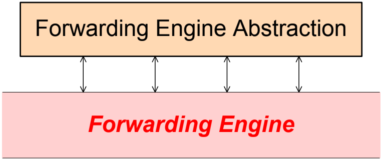
By design, XORP is flexible and modular.



RIB = routing information base
FEA = forwarding engine abstraction

XORP modules can also be *distributed* on multiple devices.

Forwarding Engine Abstraction (FEA)



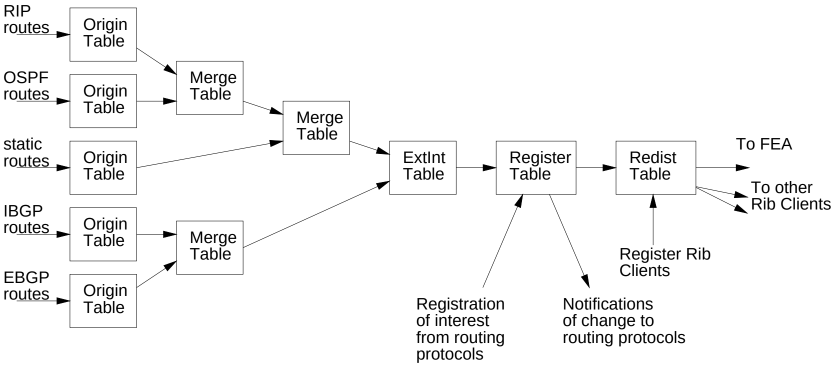
- The *Forwarding Engine* lies in the host OS kernel.
- The *Forwarding Engine Abstraction* provides a uniform interface to the underlying kernel.

Roles of the FEA:

- Interface management.
- Forwarding Table Management.
- Raw packet I/O.
- TCP/UDP socket I/O.

RIB Process

- In charge for management of the Routing Information Bases.
- By default, this process holds 4 RIBs: *Unicast* and *Multicast* RIBs for both IPv4 and IPv6.

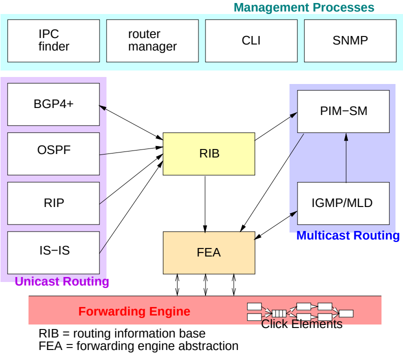


Roles of the RIB process:

- Store routes provided by the running protocols.
- Resolve conflicts for routes to identical subnets.
- Resolve next-hops to neighbors, if needed (BGP).
- Push winning **unicast** routes to the FEA.
- Permit to processes to register interest in some routing informations.
- Permit to redistribute routes from specific tables.

Inter-Process Communication (IPC)

Flexibility reachable thanks to a powerful IPC mechanism known as *XORP Resource Locators (XRLs)*.



- *IPC Finder* module is in charge for the management of this system.
- Arrows represent main (non-blocking – asynchronous) IPC calls.
- An API for each module is well-defined (*XRL Interfaces*).

XORP Resource Locator

- An XRL describes an inter-process (possibly remote) procedure call.
- Unresolved, the call is addressed to the *Finder* (whose address – `hostname:port` – must statically be defined.)

Unresolved human-readable form

`finder://fea` ← Use the Finder to reach the FEA process.
`/fti/0.1` ← Addressed to the fti (version 0.1) XRL interface.
`/add_route?` ← Call to the `add_route()` method. / Arguments: →
`net:ipv4net=10.0.0.1/8&gateway:ipv4=19.15.18.1`

- The *Finder* will resolve this call.

Resolved human-readable form

stcp://19.15.1.5:1992
```
/fti/0.1
  /add_route?
    net:ipv4net=10.0.0.1/8&gateway:ipv4=19.15.18.1
```

Module Design - Philosophy

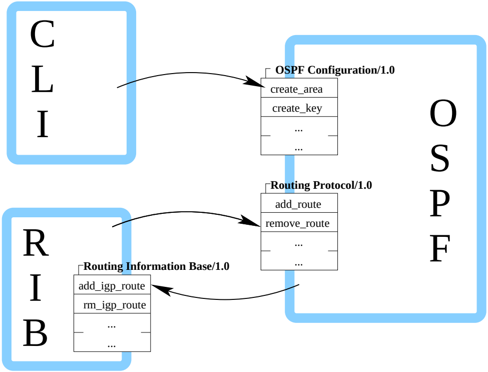
- Typically, one module = one process.

  Sometimes, two processes for both IPv4 and IPv6 versions.
- No multithreading.
- Event-driven programming.
  - Events: user actions, packet incoming, updates, etc.
  - Queued and processed ASAP by the *event loop*.
  - Timers: delayed, periodic.

⇝ Processing of an event should be as short as possible to keep the module reactive.

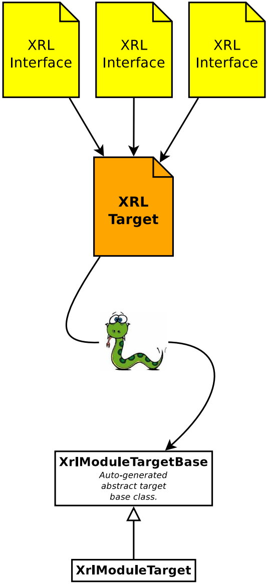
⇝ Asynchronous programming, *i.e.*, blocking calls are proscribed.

Module Design - Programming Interface



- As seen, an IPC call is described by an *XRL*.
- A *XRL Interface* is a set of methods defined to fulfill a particular function.
- A *XRL Target* is a set of XRL interfaces to fulfill a high level goal.
- Most of the time, the API of a module is a single XRL Target.

From XRL interfaces to C++



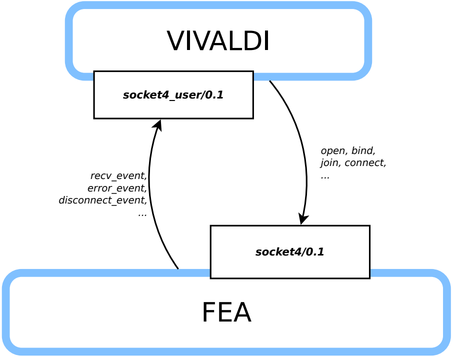
1. Definition of XRL Interfaces (`.xif` file, in `xrl/interfaces`).
2. Definition of an XRL Target (`.tgt` file, in `xrl/target`).
3. This XRL Target is automatically translated into an abstract target base class in C++.
4. A class must be derived from such a generated base class to process XRLs (IPC calls) addressed to the module.

Classes defining clients for XRL interfaces are auto-generated as well.

Outline

Socket Programming with XORP

As a first practical approach, socket programming (for IPv4) with XORP will be explained.



Socket programming involves two XRL interfaces:

1. *socket4/0.1* in the FEA.
2. *socket4_user/0.1* in our module.

The *socket4/0.1* interface defines the following types of methods:

- Creation and deletion of sockets:
  `tcp/udp_open, bind, close, ...`
- Socket management:
  `udp_join/leave_group, tcp_listen,`
  `udp_enable_recv, set_socket_option, ...`
- Sending data:
  `send, send_to, ...`

The *socket4_user/0.1* interface:

- Receiving data:
  `recv_event`
- Connection events:
  `inbound_connect_event` *(TCP)*,
  `outgoing_connect_event` *(TCP)*, `disconnect_event`
- Error event:
  `error_event`

Example: UDP Socket Creation

```
#include "xrl/interfaces/socket4_xif.hh"

template <>
bool PortBase<IPv4>
::request_udp_open_and_bind() {
  XrlSocket4V0p1Client cl(&_xrl_router);

  return cl.send_udp_open_and_bind(_socket_server_name.c_str(), // target: "fea"
                                    _xrl_router.instance_name(), // creator: "vivaldi4"
                                    _local_address,
                                    _local_port,
                                    callback(this,
                                             &PortBase<IPv4>::udp_open_and_bind_cb));
}

template <typename A>
void PortBase<A>
::udp_open_and_bind_cb(const XrlError& e, const string* psid) {
    if (e != XrlError::OKAY()) {
        set_status(SERVICE_FAILED, "Failed␣to␣open␣a␣UDP␣socket.");
        return;
    }
    _socket_id = *psid;
    set_status(SERVICE_RUNNING);
}
```

Example: Datagram reception

Reception is made through the *socket4_user* XRL interface, and specifically with this XRL:

```
XrlCmdError
socket4_user_0_1_recv_event(const string & sockid,
                            const string & if_name,
                            const string & vif_name,
                            const IPv4 & src_host,
                            const uint32_t & src_port,
                            const vector<uint8_t>& data);
```

The payload is stored within the vector referenced by `data`.

Obviously, you may need to forward the data from the XRL Target class to the object where it is actually needed.

Outline

Vivaldi XRL interface

First, define an XRL interface for the Vivaldi operations:

xrl/interfaces/vivaldi4.xif

```
/*
 * Vivaldi Coordinates System XRL interface.
 */

interface vivaldi4/0.1 {

    /**
     * Enable/disable/start/stop Vivaldi process.
     *
     * @param enable if true, then enable Vivaldi, otherwise disable it.
     */
    enable_vivaldi      ? enable:bool
    start_vivaldi
    stop_vivaldi

...
```

`xrl/interfaces/vivaldi4.xif` (ctd)

```
enable_port       ? ifname:txt \
                  & vifname:txt \
                  & addr:ipv4 \
                  & enable:bool
start_port        ? ifname:txt \
                  & vifname:txt \
                  & addr:ipv4
stop_port         ? ifname:txt \
                  & vifname:txt \
                  & addr:ipv4

get_coordinates       ? ifname:txt \
                      & vifname:txt \
                      & addr:ipv4 \
                      -> coordinates:txt
```

+ Methods to modify Vivaldi parameters (add/remove bootstrap server, probe interval, number of peers, weight of moving average, etc.)

Vivaldi API

Second, define the API – a set of XRL interfaces – for the module:

xrl/target/vivaldi4.tgt

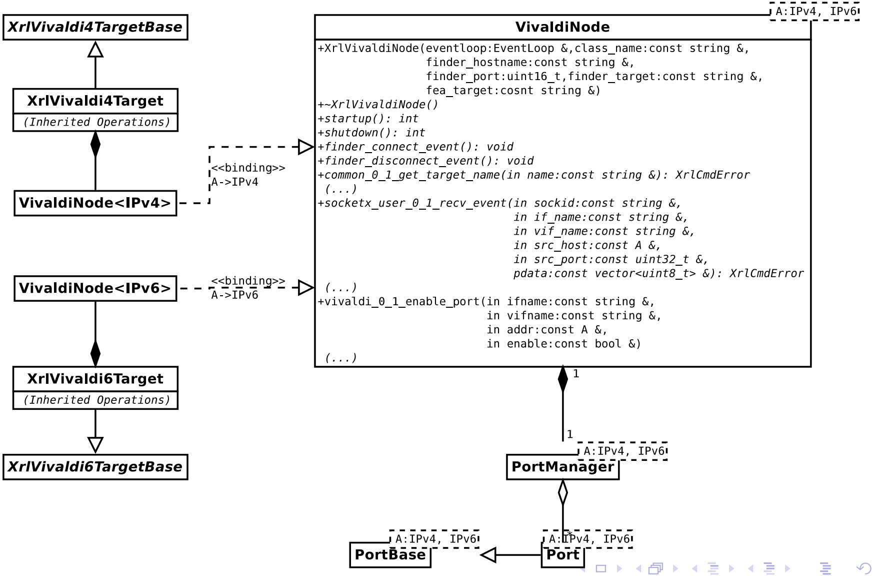
```
#include "common.xif"
#include "finder_event_observer.xif"
#include "socket4_user.xif"
#include "vivaldi4.xif"

target vivaldi4 implements   common/0.1, \
                             finder_event_observer/0.1, \
                             socket4_user/0.1, \
                             vivaldi4/0.1
```

Then, modify the script (Makefile/SCons) to generate the abstract base class for module implementation.

UML - Abstract Base Class of XRL Target

XrlVivaldi4TargetBase
#common_0_1_get_target_name(out name:string &): XrlCmdError *(...)* *#finder_event_observer_0_1_xrl_target_birth(in target_class:const string &,* *in target_instance:const string &): XrlCmdError* *(...)* *#socket4_user_0_1_recv_event(in sockid:const string &,* *in if_name:const string &,* *in vif_name:const string &,* *in src_host:const IPv4 &,* *in src_port:const uint32_t,* *in data:const vector<uint8_t> &): XrlCmdError* *(...)* *#vivaldi4_0_1_enable_vivaldi(in enable:const bool &): XrlCmdError* *(...)*

Main Class Template



Port Manager