XRL Interfaces: Specifications and Tools
Version 1.4

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

March 20, 2007

Abstract

XORP Resource Locators are the XORP projects preferred means for inter-process communication. A description of the XRL's can be found in [1]. This document describes an important aspect of the XRL world: the specification of XRL Interfaces and Targets. Following these specifications this document then describes tools that generate XRL marshalling and unmarshalling routines and assist in common case XRL handling.

1 Introduction

Up until the interface specification language and tools described here, only a low-level API was available for dispatching and handling XRL requests. The primary goal of the interface specification language and tools is to make the common case XRL code less cumbersome and more consistent. In addition, the interface specification language embeds versioning information which can be used to maintain backwards compatibility as interface develop and minimize conflicts following revisions.

2 Terminology

We define an *XRL Target* as something that XRL requests can be directed to in order to be executed. Multiple XRL targets can exist within a single process, though there will typically be one target per process. Each XRL Target has an associated name and a set of IPC mechanisms that it supports. At start-up each target registers its name and IPC mechanisms with the Finder, so other processes are able to direct requests to it.

We define an *XRL Interface* to be a set of related methods that an XRL Target would implement. Each XRL Interface is uniquely identified by a name and version number. An XRL Target will nearly always implement multiple interfaces and potentially multiple versions of particular interface. For instance, every XRL Target could implement a process information interface that would return information about the process hosting the target. Each XRL Target will implement interfaces specific to their field of operation: a routing process would implement the “routing process interface” so that the RIB process can make identity-agnostic XRL calls for tasks like redistributing a route to another routing protocol.

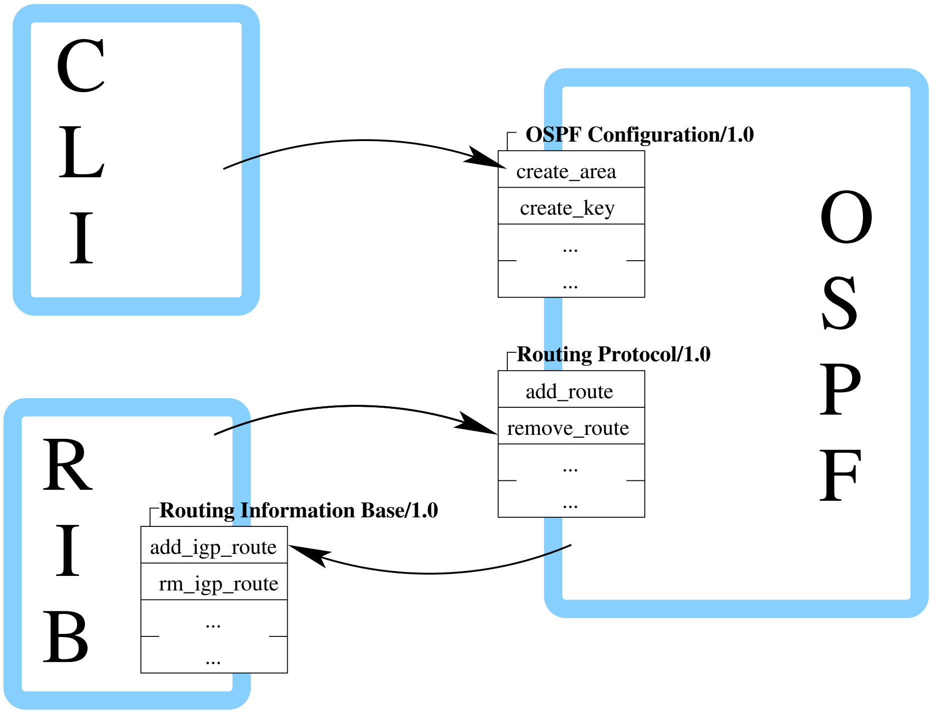

Figure 1: Example XRL Targets and clients: OSPF, RIB, and CLI. OSPF and RIB processes are XRL targets. OSPF supports XRL interfaces "OSPF Configuration/1.0" and "Routing Protocol/1.0": the CLI is a client of the "OSPF Configuration/1.0" interface and the RIB a client of the "Routing Protocol/1.0" interface. RIB supports XRL interface "Routing Information Base/1.0": the OSPF is a client of that interface.

We define an *XRL Interface Client* to be a process that accesses a particular XRL Interface.

These definitions are illustrated in figure 1.

3 Interface and Target Specifications

3.1 Interface Specifications

An interface is declared using the **interface** keyword and a set of XRL methods associated with it.

interface <*interface name*>/<*interface version*> { . . . *xrl method list* . . . }

Each item in the XRL Method list consists of a method name, its dispatch arguments, and return arguments specified in the default XRL text form. An example of this is shown in listing 1.

Listing 1: Example Xrl Interface Specification

```
/* $XORP: xorp/xrl/interfaces/test.xif,v 1.2 2006/03/26 23:59:39 pavlin Exp $ */

/*
 * This is a test file with XRL interface.
 */
```

```
interface test/1.0 {
        /**
         * Print "hello world".
         */
        print_hello_world

        /**
         * Print "hello world" and a user supplied greeting.
         *
         * @param msg greeting to be printed.
         */
         print_hello_world_and_message ? msg:txt

         /**
          * Count number of greetings available.
          */
         get_greeting_count -> num_msgs:i32

         /**
          * Get greeting.
          *
          * @param greeting_num index of greeting.
          *
          * @param greeting text of greeting.
          */
         get_greeting ? greeting_num:i32 -> greeting:txt

         /**
          * Something that always fails.
          */
         shoot_foot
}
```

Whitespace can be used to separate XRL Atoms in the description and backslashes can be used to continue XRL descriptions across multiple lines. [NB This usage of spacing is not currently supported elsewhere in the project - to be fixed.]

C-style comments are valid anywhere with interface specification files and kdoc comments can be used to describe the XRL, its arguments, and return values. Each kdoc comment is associated with the next XRL found in a specification file. All kdoc tags are valid with the exception of **@return** which is used by kdoc to describe C/C++ single return values (for XRL's return values should be document with **@param** since they are named and groups of values can be returned).

By convention, XRL Interfaces are specified in `.xif` files and these are located in the `xorp/xrl/interfaces` directory.

3.2 Target Specifications

XRL Targets are declared in *.tgt* files using:

target < *target name* > **implements** <*interface*> [, <*interface*> …]

Target specification files use the standard C include primitive to include the appropriate interface specifications. An example is shown in listing 2 and further examples can be found in the `xorp/xrl/targets` directory.

Listing 2: Example XRL Target specification.

```
/* $XORP: xorp/xrl/targets/test.tgt,v 1.2 2004/04/10 07:41:43 pavlin Exp $ */

#include "common.xif"
#include "test.xif"

target test implements          common/0.1,                          \
                                test/1.0
```

4 Tools

We use XRL Interface and XRL Target specifications to generate code that takes some of the tedium out of writing XRL related code. We have written two scripts: clnt-gen and tgt-gen that generate C++ header files and libraries and also produce a list of XRL's for each XRL Target.

4.1 clnt-gen : XRL Interface Client Generator

clnt-gen parses Xrl Interface specifications and generates a header and related library source file to simplify invoking the XRL's of that interface. Each XRL Interface specification generates an XRL Interface Client C++ class. The generated C++ class has a send method for each XRL in the XRL interface specification and takes care of argument marshalling when the XRL is dispatched and when it returns.

A fragment of generated code is shown in listing 3. This code is generated from the Xrl Interface presented in listing 1. The Xrl Interface Client C++ has a constructor that takes a XrlRouter object that holds the necessary state to correlate the XRLs being called with the return values. Each XRL specified in the XRL Interface has a send method. The send method takes the name of Xrl Target, the native C++/XORP arguments to be passed as XRL arguments, and a callback argument to be invoked when the XRL returns.

The type of the XRL callback is determined by the return type specification of the XRL. Since the syntax for the callback is awkward, a typedef is used in the class method. The relevant typedef appears immediately before each send method. When called, the first callback argument of the callback is an `XrlError` object. If the value is `XrlError::OKAY()` then return values from the XRL will be pointed to by the following arguments (*e.g.*,, const string* for string argument). Otherwise, the return value pointers will be `NULL` since there is no data available for return values when errors occur.

The interested reader should refer to the directory `xorp/xrl/tests/` for examples using the client interface.

Listing 3: Fragment of a C++ XRL Client.

```
class XrlTestV1p0Client {
public:
    XrlTestV1p0Client(XrlRouter* r) : _router(r) {}
    virtual ~XrlTestV1p0Client() {}
```

```
    /* ... */

    typedef XorpCallback2<void, const XrlError&, const string*>::RefPtr CB3;
    /**
     *  Send Xrl intended to:
     *
     *  Get greeting.
     *
     *  @param dst_xrl_target_name the Xrl target name of the destination.
     *
     *  @param greeting_num index of greeting.
     */
    void send_get_greeting(
        const char*     dst_xrl_target_name,
        const int32_t&  greeting_num,
        const CB3&      cb
    );

    /* ... */
};
```

4.2 tgt-gen : XRL Target Generator

tgt-gen takes an XRL target specification and generates list of XRL's supported by the target and a class to be sub-classed that takes care of argument marshalling and unmarshalling and sanity checks.

The XRL's for the `test/1.0` that was target presented in listing 2 are shown in listing 4. Notice that the name of each XRL has the interface and version that it belongs to prepended. This is the mechanism used to support the concept of interfaces and to support versioning.

As far as the generated base class is concerned, it implements a set of pure-virtual Xrl handler methods each of which is associated with handling a specific XRL. The implementor derives from the base class, implementing the pure-virtual methods. When an XRL request is received, it will be forwarded to the derived class's implementation handler method. The base class takes care of marshalling and unmarshalling Xrl arguments and error handling.

The design to derive a class to implement the handler functions was, in part, taken to encourage (force) people to implement handlers for all the XRL's specified for the XRL Target. In addition, implementors can add whatever state they require in dispatching XRL's to the handler class, ie pointers or references to data structures.

An example of a generated target header is shown in listing 5. Notice that the input Xrl arguments are const reference arguments to the C++ method, and the Xrl return arguments are just reference arguments.

Listing 4: Example XRL list generated from Xrl Target specification.

```
/*
 * Copyright (c) 2001-2007 International Computer Science Institute
 * See LICENSE file for licensing, conditions, and warranties on use.
 *
 * DO NOT EDIT THIS FILE - IT IS PROGRAMMATICALLY GENERATED
 *
 * Generated by 'tgt-gen'.
 *
 * $XORP: xorp/xrl/targets/test.xrls,v 1.12 2007/02/16 22:47:58 pavlin Exp $
 */

/**
 *  Get name of Xrl Target
 */
finder://test/common/0.1/get_target_name->name:txt

/**
 *  Get version string from Xrl Target
 */
finder://test/common/0.1/get_version->version:txt

/**
 *  Get status of Xrl Target
 */
finder://test/common/0.1/get_status->status:u32&reason:txt

/**
 *  Request clean shutdown of Xrl Target
 */
finder://test/common/0.1/shutdown

/**
 *  Print "hello world".
 */
finder://test/test/1.0/print_hello_world

/**
 *  Print "hello world" and a user supplied greeting.
 *
 *  @param msg greeting to be printed.
 */
finder://test/test/1.0/print_hello_world_and_message?msg:txt

/**
 *  Count number of greetings available.
 */
finder://test/test/1.0/get_greeting_count->num_msgs:i32

/**
 *  Get greeting.
 *
 *  @param greeting_num index of greeting.
 *
 *  @param greeting text of greeting.
 */
finder://test/test/1.0/get_greeting?greeting_num:i32->greeting:txt

/**
 *  Something that always fails.
 */
finder://test/test/1.0/shoot_foot
```

Listing 5: Example Xrl Target C++ Base Class

```
/*
 * Copyright (c) 2001-2007 International Computer Science Institute
 * See LICENSE file for licensing, conditions, and warranties on use.
 *
 * DO NOT EDIT THIS FILE - IT IS PROGRAMMATICALLY GENERATED
 *
 * Generated by 'tgt-gen'.
 *
 * $XORP: xorp/xrl/targets/test_base.hh,v 1.19 2007/02/16 22:47:58 pavlin Exp $
 */


#ifndef __XRL_TARGETS_TEST_BASE_HH__
#define __XRL_TARGETS_TEST_BASE_HH__

#undef XORP_LIBRARY_NAME
#define XORP_LIBRARY_NAME "XrlTestTarget"

#include "libxorp/xlog.h"
#include "libxipc/xrl_cmd_map.hh"

class XrlTestTargetBase {
protected:
    XrlCmdMap* _cmds;

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.
     */
    XrlTestTargetBase(XrlCmdMap* cmds = 0);

    /**
     * Destructor.
     *
     * Dissociates instance commands from command map.
     */
    virtual ~XrlTestTargetBase();

    /**
     * Set command map.
     *
     * @param cmds pointer to command map to associate commands with.  This
     * argument is typically a pointer to the XrlRouter associated with the
     * target.
     *
     * @return true on success, false if cmds is null or a command map has
     * already been supplied.
     */
    bool set_command_map(XrlCmdMap* cmds);

    /**
     * Get Xrl instance name associated with command map.
     */
    inline const string& name() const { return _cmds->name(); }

    /**
     * Get version string of instance.
     */
    inline const char* version() const { return "test/0.0"; }
```

```
protected:

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Get name of Xrl Target
     */
    virtual XrlCmdError common_0_1_get_target_name(
        // Output values,
        string& name) = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Get version string from Xrl Target
     */
    virtual XrlCmdError common_0_1_get_version(
        // Output values,
        string& version) = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Get status of Xrl Target
     */
    virtual XrlCmdError common_0_1_get_status(
        // Output values,
        uint32_t&       status,
        string& reason) = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Request clean shutdown of Xrl Target
     */
    virtual XrlCmdError common_0_1_shutdown() = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Print "hello world".
     */
    virtual XrlCmdError test_1_0_print_hello_world() = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Print "hello world" and a user supplied greeting.
     *
     *  @param msg greeting to be printed.
     */
    virtual XrlCmdError test_1_0_print_hello_world_and_message(
        // Input values,
        const string&   msg) = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Count number of greetings available.
     */
    virtual XrlCmdError test_1_0_get_greeting_count(
        // Output values,
        int32_t&        num_msgs) = 0;
```

```
    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Get greeting.
     *
     *  @param greeting_num index of greeting.
     *
     *  @param greeting text of greeting.
     */
    virtual XrlCmdError test_1_0_get_greeting(
        // Input values,
        const int32_t&  greeting_num,
        // Output values,
        string& greeting) = 0;

    /**
     *  Pure-virtual function that needs to be implemented to:
     *
     *  Something that always fails.
     */
    virtual XrlCmdError test_1_0_shoot_foot() = 0;

private:
    const XrlCmdError handle_common_0_1_get_target_name(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_common_0_1_get_version(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_common_0_1_get_status(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_common_0_1_shutdown(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_test_1_0_print_hello_world(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_test_1_0_print_hello_world_and_message(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_test_1_0_get_greeting_count(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_test_1_0_get_greeting(const XrlArgs& in, XrlArgs* out);

    const XrlCmdError handle_test_1_0_shoot_foot(const XrlArgs& in, XrlArgs* out);

    void add_handlers();
    void remove_handlers();
};

#endif // __XRL_TARGETS_TEST_BASE_HH__
```

A Modification History

- December 11, 2002: Initial version 0.1 completed.
- March 10, 2003: Minor changes. Updated the version to 0.2, and the date.
- June 9, 2003: Updated the version to 0.3, and the date.
- August 28, 2003: Updated the version to 0.4, and the date.
- November 6, 2003: Updated the version to 0.5, and the date.
- July 8, 2004: Updated the version to 1.0, and the date.

- April 13, 2005: Updated the version to 1.1, and the date.
- March 8, 2006: Updated the version to 1.2, and the date.
- August 2, 2006: Updated to match XORP release 1.3: fixed sample XRL-related code to match the current code. Added “Modification History” appendix.
- March 20, 2007: Updated the version to 1.4, and the date.

References

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