XORP Libxorp Library Overview

Version 0.4

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

August 28, 2003

1 Introduction

The *libxorp* library contains a set of classes for basic XORP functionality such as IP addresses and subnets, timers, event loops, etc. It is used by virtually every other XORP component, and its main purpose is to simplify the implementation of those components.

1.1 Overview

Currently, the libxorp library contains the following classes and components (in alphabetical order):

- *asnum.hh: class AsNum*: A class for storing an AS number used by protocols such as BGP.
- *asyncio.hh: class AsyncFileReader, class AsyncFileWriter*: Asynchronous file transfer classes.
- *buffer.hh: class Buffer*: A class for storing buffered data.
- *c_format.hh: c_format()*: A macro that creates a C++ string from a C-style printf(3)-formatted string.
- *callback.hh:* Callback mechanism.
- *config_param.hh: template class ConfigParam*: A class for storing a configuration parameter.
- *debug.h*: Provides facility for generating debug messages.
- *ether_compat.h*: Ethernet manipulation compatibility functions.
- *eventloop.hh: class EventLoop*: Event loop class for coordinated operations between timers and I/O operations on file descriptors.
- *exceptions.hh*: Standard XORP C++ exceptions.
- *heap.hh: class Heap*: A heap that implements a priority queue.
- *ipnet.hh, ipv4net.hh, ipv6net.hh, ipvxnet.hh: class IPv4Net, class IPv6Net, class IPvXNet*: Implementation of classes for basic subnet addresses (for IPv4, IPv6 and dual IPv4/6 address family respectively).

- *ipv4.hh, ipv6.hh, ipvx.hh: class IPv4, class IPv6, class IPvX*: Implementation of classes for basic IP addresses (for IPv4, IPv6 and dual IPv4/6 address family respectively).
- *mac.hh: class Mac, class EtherMac*: Containers for MAC types.
- *nexthop.hh*: Classes that contain routing next-hop information.
- *ref_ptr.hh: template class ref_ptr*: Reference counted pointer class.
- *ref_trie.hh*: Implementation of a trie to support route lookups. Based on trie.hh, but with reference-counted storage supporting delayed deletion.
- *selector.hh*: I/O multiplexing interface.
- *status_codes.h*: Process states status codes.
- *time_slice.hh: class TimeSlice*: A class for computing whether some processing is taking too long.
- *timer.hh, class XorpTimer*: XORP timer facility.
- *timeval.hh: class TimeVal*: A class for storing time values (similar to *struct timeval*).
- *token.hh*: Token related definitions.
- *transactions.hh*: Facility for transaction operations.
- *trie.hh*: Implementation of a trie to support route lookups.
- *utility.h*: Contains various mini-utilities (mostly compiler-related helpers).
- *utils.hh*: Contains various utilities (*e.g.,* to delete a list or array of pointers and the objects pointed to).
- *vif.hh: class Vif, class VifAddr*: Virtual interface and virtual interface address classes.
- *xlog.h*: Provides facility for log messages generation.
- *xorp.h*: The XORP main include file that should be included by all XORP C and C++ files.

Each of the components is described in Section 2.

2 Components Description

This section contains a brief description of each of the components of the *libxorp* library. This description is for informative purpose only. The source code for each component is the ultimate source for programming reference, and implementation details.

2.1 asnum.hh

This file contains *class AsNum*: a class for storing an AS number used by protocols such as BGP. This class can be used to store an AS number that can be either 16 or 32 bits. Originally, the AS numbers were defined as 16-bit unsigned numbers. Later the "extended" AS numbers were introduced, which are unsigned 32-bit numbers.

2.2 asyncio.hh

This file contains asynchronous file transfer classes. These utilize XORP EventLoop and its SelectorList to read or write files asynchronously. The user creates an AsyncFile{Reader,Writer} and adds a buffer for reading or writing with add_buffer(). A callback provided with each buffer is called every time I/O happens on the buffer. Reading or writing only begins when start() is called, and normally continues until there are no buffers left.

From the developer's point of view, the following classes are of interest: *class AsyncFileReader, class AsyncFileWriter*.

2.3 buffer.hh

This file contains *class Buffer*: a class for conveniently storing and accessing buffered data. Currently it has limited applicability.

2.4 c_format.hh

This file contains *c_format()*: a macro that creates a C++ string from a C-style printf(3)-formatted string. It takes the same arguments as printf(3), but %n is illegal and will cause abort to be called.

In practice c_format() is a nasty macro, but by doing this we can check the compile time arguments are sane and the run time arguments.

2.5 callback.hh

This file contains an implementation of a callback mechanism. XORP is an asynchronous programming environment and as a result there are many places where callbacks are useful. Callbacks are typically invoked to signify the completion or advancement of an asynchronous operation.

XORP provides a generic and flexible callback interface that utilizes overloaded templatized functions for for generating callbacks in conjunction with many small templatized classes. Whilst this makes the syntax a little unpleasant, it provides a great deal of flexibility.

XorpCallback objects are objects created by the callback() function which returns a reference pointer to a newly created callback object. The callback is invoked by calling the dispatch() method on that object.

For more details on the callback mechanism, and for usage examples, see the beginning of *callback.hh*. Note that this file is auto-generated by callback-gen.py (a Python script), therefore it should never be edited.

2.6 config_param.hh

This file contains the implementation of *template class ConfigParam*: a class for storing a configuration parameter.

This class can be used to store the value of a configuration parameter. Such parameter has a current and a default value. The *ConfigParam* class has the facility to add a callback that is invoked whenever the value of the configuration parameter is changed.

2.7 debug.h

This file provides facility for debug messages generation. More specifically, it defines the `debug_msg()`, the macro responsible for generating debug messages. It takes the same arguments as printf(3). For example:

```
debug_msg("The number is %d\n", 5);
```

For more details see the comments inside that file.

2.8 ether_compat.h

This file contains Ethernet-related manipulation compatibility functions. For example, it includes the appropriate system files, and declares functions `ether_aton()` and `ether_ntoa()` (implemented locally in *ether_compat.c*) if the system is missing the corresponding `ether_aton(3)` and `ether_ntoa(3)`.

2.9 eventloop.hh

This file defines *class EventLoop*. It is used to co-ordinate interactions between a TimerList and a SelectorList for XORP processes. All XorpTimer and select operations should be co-ordinated through this interface.

2.10 exceptions.hh

This file contains *class XorpException*: a base class for XORP C++ exceptions. It contains also all standard XORP C++ exceptions. An example of such exception is *class InvalidFamily* which is thrown if the address family is invalid (for example, by an IPvX constructor when invoked with an invalid address family).

2.11 heap.hh

This file contains *class Heap*. Heap implements a priority queue, mostly used by Timer objects. This implementation supports removal of arbitrary objects from the heap, even if they are not located at the top.

2.12 ipnet.hh, ipv4net.hh, ipv6net.hh, ipvxnet.hh

These files contain the declaration of the following classes: *class IPv4Net, class IPv6Net, class IPvXNet*, which are classes for basic subnet addresses (for IPv4, IPv6 and dual IPv4/6 address family respectively). IPvXNet can be used to store a subnet address that has either IPv4 or IPv6 address family.

Most of the implementation is contained in file *ipnet.hh*, which contains a *template class IPNet*. The IPv4Net, IPv6Net, and IPvXNet classes are derived from that template.

2.13 ipv4.hh, ipv6.hh, ipvx.hh

These files contain the declaration for the following classes: *class IPv4, class IPv6, class IPvX*, which are classes for basic IP addresses (for IPv4, IPv6 and dual IPv4/6 address family respectively). IPvX can be used to store an address that has either IPv4 or IPv6 address family.

2.14 mac.hh

This file declares the following classes: *class Mac, class EtherMac*. The first class is a generic container for any type of MAC. The second class is a container for Ethernet MAC address.

2.15 nexthop.hh

This file declares a number of classes that can be used to contain routing next-hop information. For example, *class NextHop* is the generic class for holding information about routing next hops. NextHops can be of many types, including immediate neighbors, remote routers (with IBGP), discard interfaces, encapsulation endpoints, etc. NextHop itself doesn't really do anything useful, except to provide a generic handle for the specialized subclasses. The specialized subclasses are:

- IPPeerNextHop is for next hops that are local peers.
- IPEncapsNextHop is for "next hops" that are non-local, and require encapsulation to reach. An example is the PIM Register Encapsulation.
- IPExternalNextHop An IP nexthop that is not an intermediate neighbor.
- DiscardNextHop is a discard interface.

2.16 ref_ptr.hh

This file declares *template class ref_ptr*: reference counted pointer class.

The ref_ptr class is a strong reference class. It maintains a count of how many references to an object exist and releases the memory associated with the object when the reference count reaches zero. The reference pointer can be dereferenced like an ordinary pointer to call methods on the reference counted object.

At the time of writing the only supported memory management is through the new and delete operators. At a future date, this class should support the STL allocator classes or an equivalent to provide greater flexibility.

2.17 ref_trie.hh

This file implements a trie to support route lookups. The implementation is template-based, and is based on the code in trie.hh. From deleloper's point of view, templates RefTrie, RefTrieNode, RefTriePreOrderIterator, and RefTriePostOrderIterator are the most important. Those templates should be invoked with two classes, the basetype "A" for the search Key (which is a subnet, `IPNet<A>`), and the Payload.

RefTrie differs from Trie (and its associated classes) in that the RefTrieNode includes a reference count of how many RefTrieIterators are pointing at it. If a RefTrieNode is deleted, but has a non-zero reference count, deletion will be delayed until the reference count becomes zero. In this way, additions and deletions to the RefTrie cannot cause a RefTriePreOrderIterator or RefTriePostOrderIterator to reference invalid memory, although a deletion and subsequent addition can cause the payload data referenced by an iterator to change.

2.18 selector.hh

This file contains the I/O multiplexing interface. The particular class of interest is *class SelectorList*.

A SelectorList provides an entity where callbacks for pending I/O operations on file descriptors may be registered. The callbacks are invoked when one of the select methods is called and I/O is pending on the particular descriptors.

2.19 status_codes.h

This file contains the enumerated *ProcessStatus* codes that a XORP process should report to the XORP router manager (*rtrmgr*) [1]. The file itself contains a detailed explanation of the process states (valid transaction between states, triggering events, actions, etc).

2.20 time_slice.hh

This file declares *class TimeSlice*. This class can be used to compute whether some processing is taking too long time to complete. It is up to the program that uses TimeSlice to check whether the processing is taking too long, and suspend processing of that task if necessary.

2.21 timer.hh

This file declares the XORP timer facility. The only class of interest from a developer's point of view is *class XorpTimer*.

2.22 timeval.hh

This file contains implementation of *class TimeVal* for storing time values (similar to *struct timeval*). *TimeVal* implements the appropriate constructors and numerous helper methods (*e.g.,* Less-Than and Addition operators, etc).

2.23 token.hh

This file contains various token-related definitions. Token is a sequence of symbols separated from other tokens by some pre-defined symbols. In this implementation, the separators are the is_space(3) and '—' characters. The facilities in that file are to copy tokens, removing them from a token line, etc. Currently, this file is used only by the CLI, therefore in the future it may be moved to the CLI itself.

2.24 transactions.hh

This file contains facility for transactions. A transaction consists of a sequence of transaction operations, each of which is a command. The TransactionManager class provides a front-end for creating, dispatching, and destroying transactions.

2.25 trie.hh

This file implements a trie to support route lookups. The implementation is template-based. From deleloper's point of view, templates Trie, TrieNode, TriePreOrderIterator, and TriePostOrderIterator are the most important. Those templates should be invoked with two classes, the basetype "A" for the search Key (which is a subnet, `IPNet<A>`), and the Payload.

2.26 utility.h

This file contains various mini-utilities. Those utilities are mostly compiler-related helpers; *e.g.,* compile-time assertion, *UNUSED(var)* macro to suppress warnings about unused functions arguments, etc.

2.27 utils.hh

This file contains various helper utilities. Currently, the only two utilities are template functions to delete a list or array of pointers and the objects pointed to.

2.28 vif.hh

This file declares the following classes: *class Vif, class VifAddr*.

Class Vif holds information about a virtual interface. A Vif may represent a physical interface, or may represent more abstract entities such as the Discard interface, or a VLAN on a physical interface. VifAddr holds information about an address of a virtual interface. A virtual interface may have more than one VifAddr.

2.29 xlog.h

This file provides facility 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_MAX excluded):

  ```
  typedef enum {
      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). An example:

```
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.

2.30 xorp.h

This is the XORP main include file that should be included by all XORP C and C++ files. This file itself includes a number of frequently used system header files, defines several commonly used values, etc.

A Modification History

- December 11, 2002: Version 0.1 completed.
- March 10, 2003: Updated to match XORP version 0.2 release code; add information about RefTrie; cleanup.
- June 9, 2003: Updated to match XORP version 0.3 release code.
- August 28, 2003: Updated to match XORP version 0.4 release code.

References

[1] XORP Router Manager Process (rtrmgr). XORP technical document. http://www.xorp.org/.