Seventh FRAMEWORK PROGRAMME
FP7-ICT-2007-2 - ICT-2007-1.6
New Paradigms and Experimental Facilities

SPECIFIC TARGETED RESEARCH OR INNOVATION PROJECT

Deliverable D4.2:
EUA Software Documentation

Project description
Project acronym: **ECODE** Project full title: **Experimental Cognitive Distributed Engine** Grant Agreement no.: **223936**
Document Properties
Number: FP7-ICT-2007-2-1.6-223936-D4.2
Title: EUA Software Documentation
Responsible: Laurent Mathy (ULANC)
Editor(s): Matthew Jakeman (ULANC)
Authors: Laurent Mathy, Matthew Jakeman, Steven Simpson (ULANC), Dimitri Papadimitriou (ALB)
Dissemination level: Public (PU)
Date of preparation: ***Date*** Version: 1.0

Table of Contents

1. Introduction

This document accompanies the "machine learning engine" prototype deliverable D4.2. It describes the ECODE Unified Architecture (EUA) that enables Machine Learning Engines (MLE's) to execute on software based router platform. As a foundation the EUA uses the eXtensible Open source Routing Platform (XORP) and builds upon the functionality it offers to enable Machine Learning Engines (MLE's) to operate within a router. XORP provides a software platform that is used to turn regular PC's running Linux into a router platform. It provides mechanisms to implement extensions to the router known as XORP processes. These processes can communicate with the rest of the routing platform. The EUA is be implemented in the form of a number of these XORP processes. This document further details the functionality the EUA platform provides as well as providing information for users of the software with regards to installation and general usage instructions.

This document is organised as follows. Section 2 details the installation procedure for the EUA. Section 3 details the features the EUA offers and Section 4 is structured as the software user manual.

2. Installation Procedures

This section describes the procedures for installing the EUA software. Although the software should operate on any Unix based operating system it has only been comprehensively tested on Linux, specifically Ubuntu. Hence, it is highly recommended to run the EUA on an Ubuntu based machine and some of the instructions described in this section are Ubuntu specific. For notes on installation on other Operating Systems see the file xorp/BUILD_NOTES in the source tarball.

2.1 Pre-Requisites

There are a number of libraries required in order to compile the EUA source code. Both the GNU C and C++ compilers are needed. It is also necessary to install the SSL development libraries. XORP also uses a build system called scons which is not installed by default. The packages in the list below are recommended and can be obtained from Ubuntu's ***apt*** repositories:

- gcc-4.3
- g++-4.3
- libssl-dev
- scons
- openssl
- traceroute
- iptables-dev

Once all of these libraries have been installed the source code can be compiled and installed.

2.2 Compilation and Installation

This section outlines the compilation procedures for the EUA and the TCI resolver.

2.2.1 EUA Compilation

The compilation of XORP and the included EUA enhancements is performed using the scons tool which should now have been installed. First, scons needs to be executed inside of the xorp/ directory. This will compile all of the source code. However, because the EUA uses a new implementation of an asynchronous server inside of XORP a flag needs to be passed to scons so the following command is used:

- scons enable_async_server=True

Once the compilation has successfully completed the binaries can be installed with the following command:

- sudo scons enable_async_server=True install

This command installs the compiled binaries into the usr/local/xorp/sbin directory.

More information regarding the scons build system can be obtained with the command:

- scons –help

2.2.2 TCI Resolver

The TCI resolver code is located within the resolver/ directory in the root of the source code tarball. Inside the directory, there is a build script which can be executed to compile the code.

By executing the following two commands in the resolver/ directory the code will be compiled and the compiled binaries can be found in the src/ directory created by the *build.sh* script:

- chmod +x build.sh
- ./build.sh

2.3 Configuring the System

In order to run the EUA a few settings are required on the host system. XORP requires that there is a xorp group on the system and that the user running XORP is a member of that group.

As XORP only runs as a root user the following commands are all prefixed with sudo to give the appropriate user privileges.

- sudo addgroup xorp
- sudo adduser root xorp

Once this has been completed, the session needs to be restarted for the new group and user rights to apply. Logging out of the system and back in again will accomplish this.

3. Software Features

The EUA offers a number of features to developers that enable the creation of Machine Learning Engines (MLEs) within a router. This functionality has been achieved by extending the XORP platform with two classes of XORP process (MLPs and MPs), and the realized Translation and Communication Interface (TCI) process that governs their interaction.

3.1 Remote interaction between MLEs

MLEs need raw measurements to operate on, and might need to obtain them from remote XORP routers. To support that, the EUA is built on an architecture that separates machine-learning logic from measurement, and includes a XORP process to facilitate the interaction.

Machine-Learning Processes (MLPs) are XORP processes that implement specific Machine-Learning Engines. The EUA does not impose any additional constraints on them, except that they must be XORP processes, and so should normally communicate control messages via XORP's own IPC mechanism, XRLs. Other components of the architecture exist to support MLPs by performing measurement tasks, and coordinating remote communication.

Measurement Processes (MPs) concentrate on providing basic input data to MLPs. A given MP will be concerned with a specific kind of measurement, e.g. one for round-trip times, one for bandwidth and throughput, and one for link availability. An MP may provide measurements in a single-shot mode or a push mode. For example, an MP providing round-trip times may provide an XRL command that performs several RTT measurements, and then finally returns an average after several turns. Alternatively, or additionally, it may provide an XRL command which takes a callback XRL to be invoked continually with the latest measurement. An MP is expected to support multiple clients (normally MLPs) but does not normally handle remote communication.

The TCI is a specific XORP process that governs interaction between MLPs and MPs. Rather than an MLP asking an MP directly for a specific measurement, the MLP asks its local TCI to invoke the MP. This gives the TCI the opportunity to invoke a remote MP by interacting with a remote TCI located on the same XORP router as the MP. This feature is referred to as 'dispatching', and the TCI provides the single-shot form with no prior knowledge of the XRL types supported by the MPs. For push mode, an MP is required to register its push commands with the TCI before it can be invoked.

3.1.1 Callback Techniques

The dispatch and dispatch_push methods described below require the MLP to provide XRLs through which results are delivered. For dispatch (Section 3.1.2), these results are the out-arguments of the dispatched XRL. For dispatch_push (Section 3.1.4), these are the data arguments pushed with each update.

3.1.1.1 Callback XRL Signatures

When a callback is invoked on the MLP, it will have no context to distinguish the received results from other invocations on the same XRL. The approach shown here allows the MLP's arbitrary context requirements to be met in combination with the arbitrary formats of the results which an MP can provide.

The signature of the callback XRL must include as in-parameters all parameters that the MLP needs as context, followed by all the MP's method's out-parameters. When the MLP invokes the MP, it passes a *partial* callback XRL, i.e. one containing only its own context arguments. When the callback is invoked (either by the MP or the TCI), the caller first *completes* it by appending each of the result arguments. The name cbxrl is conventionally used as the name of the string parameter that carries the callback XRL from the MLP towards the MP.

For example, when dispatch is used on and MP method with the following signature:

```
ping ? host:ipv4 & tries:u32 & period:u32 -> delay:u32;
```

then the callback method must have at least the following:

```
// In interface myif/0.1, for example
ping_result ? delay:u32;
```

That is, its in-parameters match the out-parameters of the MP method. In this case, the MLP should use the following as its cbxrl:

```
finder://mlp_name/myif/0.1/ping_result
```

Note: the string mlp_name can be obtained by calling XrlRouter::class_name(). If the MLP implements a target node, it will likely be inheriting from an XrlStdRouter, which itself is an XrlRouter.

If the MLP needs any context with the callback (e.g. the address that is being pinged), it can add it to cbxrl:

```
finder://mlp_name/myif/0.1/ping_result?addr:ipv4=4.3.2.1
```

together with a corresponding change to the callback XRL's signature:

```
// In interface myif/0.1, for example
ping_result ? addr:ipv4 & delay:u32;
```

It is possible that result arguments and context arguments will have the same name. To distinguish them, wherever a callback XRL is accepted, another string parameter is also accepted, which specifies a prefix to be used on the name of each appended result argument. The name cbpfx is conventionally used as the name of the string parameter that specifies this prefix.

To deal with XRL command errors, you may need a separate method to receive the error attributes:

```
// In interface myif/0.1, for example
ping_result ? addr:ipv4 & delay:u32;
ping_error ? addr:ipv4 & code:u32 & note:txt;
```

Of course, you must then also pass a separate errxrl alongside cbxrl:

```
finder://mlp_name/myif/0.1/ping_error?addr:ipv4=4.3.2.1
```

3.1.1.2 Hidden Callback Methods

To receive the results of a dispatch call, or the updates of a dispatch_push call, it is not necessary to add a callback method to the MLP's public interface. Further, in the case of dispatch, where an XRL command error could be returned instead of a result, it is not necessary to define two separate callback methods, i.e., one for results and one for errors. Below, we show that it is possible to inject the handler for a method into an MLP's C++ implementation, which method is not declared in any

interface, and which can cope with both result and error calls between the XORP IPC mechanism and XRL argument marshalling.

Suppose you have a class XrlMlpExampleNode to extend the generated class XrlEuaMlpExampleTargetBase. The convention may be to also extend XrlStdRouter:

```
class XrlMlpExampleNode : public XrlStdRouter,
                          public XrlEuaMlpExampleTargetBase,
                          ... {
};
```

When this class is instantiated, its XrlStdRouter component will be initialized first. This initialization is necessary as XrlEuaMlpExampleTargetBase will be passed that component's address as part of its configuration. XrlEuaMlpExampleTargetBase then registers handlers with the XrlStdRouter for all the target's methods. These handlers parse out specific arguments and pass them to your class's implementations of those methods. The idea is to inject a handler for an undeclared method to receive arguments in an untyped form (as far as C++ is concerned). For example, the handler would be declared as follows:

```
const XrlCmdError
XrlMlpExampleNode::ping_handler(const XrlArgs &in, XrlArgs *out)
{
  // Prepare to receive arguments.
  IPv4 host;
  uint32_t delay;
  string err_note;
  uint32_t err_code;

  // Do our own marshalling.
  in.get("host", host);
  try {
    in.get("delay", delay);
    // We got the result...

  } catch (const XrlArgs::BadArgs &er) {
    // Get the error code instead, but see note below.
    in.get("code", err_code);
    in.get("note", err_note);
  }

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

It might appear that the class's own constructor could arrange to inject methods not declared on the implemented interfaces, but it can't because the XrlEuaMlpExampleTargetBase will have finalized the XrlStdRouter by the time the constructor is called, so it can't receive any more methods.

Extra methods can be injected like this:

```
class XrlMlpExampleNode : public XrlStdRouter,
                          DispatchCBs, // yes, between these two classes
                          public XrlEuaMlpExampleTargetBase,
                          ... {
};
```

Then declare the class entirely in-line:

```
struct DispatchCBs { // Probably give it a more distinctive name.
  DispatchCBs(XrlCmdMap *cmds) {
    if (!cmds->add_handler("myif/0.1/ping_cb",
```

```
                          callback(this, &DispatchCBs::ping_handler))) {
      // Report an error.
    }
  }

  virtual const XrlCmdError
  ping_handler(const XrlArgs &in, XrlArgs *out) = 0;
};
```

Of course, by inheriting this class, ping_handler now has to be implemented, but this is the actual purpose of this exercise. Now if an XRL like this is supplied:

```
finder://mlp_name/myif/0.1/ping_cb?and&the&args
```

...as your cbxrl, the C++ method ping_handler can be called with the given arguments plus any results from the MP via the TCI.

Note, however, a restriction imposed by current XORP implementations. Although this technique permits both results and errors to be syntactically handled by the same function, it is not currently possible to capture the BadArgs exception that indicates an error. By omission, the XrlArgs::get functions have not been declared to throw BadArgs; consequently, C++ forbids catching of BadArgs when it happens, and abruptly terminates the process instead. Nevertheless, the technique is still valuable if two functions are defined to handle results and errors separately, or if errors cannot occur. Furthermore, future versions of XORP may include a simple fix that allows the exception to be caught.

3.1.2 The dispatch Method

This method allows an MLP (or, any XORP process) to invoke an arbitrary XRL on an MP (or any XORP process) via the TCI. Additionally, the TCI can be instructed to contact a remote TCI, and have that invoke an MP at that remote location. Results are returned to the MLP by it providing an XRL to be used as a callback. In effect, one duplex XRL invocation is turned into two simplex ones, one in each direction.

The dispatch method was provided as a stop-gap solution to the call-chaining problem encountered when attempting to implement remote dispatch within the TCI. We describe it here for completeness, but the direct_dispatch method is superior (it is simpler to use by an MLP), and should be used in preference.

3.1.2.1 Interface

The definition of the dispatch method in eua_tci/0.1 is:

```
  dispatch ? tci_id:txt & euaxrl:txt & \
             cbxrl:txt & cbpfx:txt & errxrl:txt & errpfx:txt;
```

An MLP is expected to invoke this method on its local TCI. The TCI will then invoke an XRL either on a local MP directly (if tci_id is empty), or on a remote MP via another TCI (identified by tci_id, as passed to the TCI Resolver).

euaxrl specifies the XRL that should be invoked on the MP by its local TCI. This does not vary as the MLP switches between making a local or remote dispatch call.

There are no return parameters. The MLP's local TCI passes all results back by calling an XRL formed from the template cbxrl. Again, this does not depend on whether the MLP is making a local or remote dispatch call, as it will always be invoked by its local TCI.

The actual callback XRL is a combination of cbxrl (which can include any arguments the MLP needs for context) and the out-arguments of the euaxrl invocation (i.e. the results), which are appended.

To deal with errors, the XrlCmdError is appended instead of the results in the form of two arguments, code:u32 and note:txt. The following arguments are normally blank but can be set to adjust the callback:

- If not blank, errxrl is an XRL template to be used instead of cbxrl if an error occurs.
- On success, the value of cbpfx is prefixed to the names of each result argument appended to cbxrl.
- On error, the value of errpfx is prefixed to the names of the two error arguments code and note.

3.1.2.2 Usage by MLP Implementations

To invoke an MP's method indirectly through the TCI, the MLP must set up a callback method to receive the MP's out-arguments. This XRL method must have a signature combining any context arguments required by the MLP with the out-arguments of the MP's XRL method, as covered in Section 3.1.1.1. The MLP may declare the method explicitly on one of its interfaces, or it may use the 'hidden callback' technique shown in Section 3.1.1.2.

3.1.2.3 Usage by MP Implementations

Integration with the dispatch method is trivial for MP implementations: simply provide an ordinary method with in- and out-parameters. In other terms, the MP need not take any special action.

The TCI will await the out-arguments and change them into the in-arguments of another XRL. The MP will be unaware that the TCI is doing anything special, as it will just appear as an ordinary XRL method call.

3.1.3 The direct_dispatch Method

3.1.3.1 Interface

The definition of the direct_dispatch method in eua_tci/0.1 is:

```
// in eua_tci/0.1
direct_dispatch ? tci_id:txt & euaxrl:txt -> ret:binary;
```

This method is meant as a replacement for dispatch that takes advantage of the forthcoming “asynchronous method implementations” feature of XORP, which has already been incorporated into EUA.

An MLP is expected to invoke the direct_dispatch method on its local TCI. The TCI will then invoke an XRL on either a local MP directly (if tci_id is empty), or a remote MP via another TCI (identified by tci_id, as passed to the TCI Resolver).

euaxrl specifies the XRL that should be invoked on the MP by its *local* TCI. This does not vary as the MLP switches between making a local or remote dispatch call.

Results are converted to a generic binary form before being returned to the MLP. The MLP should therefore convert ret into an XrlArgs, and read the out-arguments from that, according to the out-signature of the XRL specified by euaxrl.

If an error occurs in invoking the MP, an error will be returned from the direct_dispatch call.

3.1.3.2 Usage by MLP Implementations

The caller will have to unmarshal out-arguments itself, as they are provided as a single binary argument in order for direct_dispatch to be compatible with all MP methods. For example, if the MP method has the following signature:

```
ping ? host:ipv4 & tries:u32 & period:u32 -> delay:u32;
```

then the caller can extract the delay argument with the likes of:

```
void dispatched_ping_cb(const XrlError& xrl_error,
                        const vector<uint8_t>* ret)
{
  if (xrl_error.isOK()) {
    assert(ret);

    // 'ret' is a packed XrlArgs.
    XrlArgs args;
    args.unpack(&(*ret)[0], ret->size());
    uint32_t delay;
    args.get("delay", delay);
  } else {
    // Something went wrong, crash probe told.
  }
};
```

3.1.3.3 Usage by MP Implementations

Integration with the direct_dispatch method is trivial for MP implementations: simply provide an ordinary synchronous method implementation with in- and out-parameters, or a corresponding asynchronous implementation. In other words, the MP need not take any special action.

The TCI will await the out-arguments from the MP, and convert them to a binary form. The MP will be unaware that the TCI is doing anything special, as it will just appear as an ordinary XRL method call.

3.1.4 The dispatch_push Method

This method aims to allow an MLP to receive continuous updates from a local or remote MP via the TCI.

3.1.4.1 Interface

The definition of the dispatch_push method in eua_tci/0.1 is:

```
// in eua_tci/0.1

// to be called by MPs
register_push ? iface:txt & start:txt & stop:txt & \
                cbname:txt & pfxname:txt & \
                mediate:txt;

// to be called by TCIs
get_push ? iface:txt & name:txt \
        -> start:txt & stop:txt & \
           mediate:txt;

// to be called by MLPs
dispatch_push ? tci_id:txt & euaxrl:txt & \
                cbxrl:txt & cbpfx:txt;
```

MPs must provide push methods in pairs, one starting a push, and one stopping. Parameters are divided into:

- those which identify the data to be obtained (data-id),
- those which identify the recipient of the data (recv-id),
- those which control how the data is to be obtained, e.g. frequency (ctrl).

The first two groups, i.e., data-id and recv-id, identify an ***interest***.

Pushing takes the form of repeated calls originally made by the MP to some or all of the interested parties (as specified by recv-id arguments) with the relevant data (as specified by data-id arguments). The MP's specification promises to include certain *data* parameters in each call.

3.1.4.2 Starting and stopping a push

An MLP is expected to invoke dispatch_push method on its local TCI. The TCI will then invoke an XRL either on a local MP directly (if tci_id is empty), or on a remote MP via another TCI (identified by tci_id, as passed to the TCI Resolver).

euaxrl specifies the XRL that should be invoked on the MP by its *local* TCI. This does not vary as the MLP switches between making a local or remote dispatch call.

cbxrl must contain a *partial* callback XRL to be invoked when the MP has data to report to the MLP. Arguments already present on the XRL are simply echoed in that call, but arguments supplied by the MP are appended to *complete* the XRL, with their names prefixed with cbpfx. cbxrl and cbpfx together form the recv-id arguments.

A callback XRL can be set up in the MLP using the same techniques as for dispatch. Section 3.1.1.1 shows how the signature of the callback is determined. Section 3.1.1.2 shows how the callback XRL need not be declared in the MLP's interfaces.

If an error occurs in invoking the MP, an error will be returned from the direct_dispatch call.

An interest is **created** or **updated** when euaxrl identifies a start method, including data-id and ctrl arguments. The data-id and recv-id arguments together identify the interest. If an interest already exists with the same data-id and recv-id profile, the call is an *update*; otherwise, it is a *creation*.

An interest is **destroyed** when euaxrl identifies a stop method, including only data-id and recv-id arguments. Any existing interest with the same data-id and recv-id profile is destroyed. It is no error if no such interest exists. An interest may also be destroyed implicitly if the recipient no longer appears to be capable to receive data.

3.1.4.3 MP Registration

An MP must register its push methods with its local TCI by calling register_push. The start argument is the name of the XRL method that starts a push, while stop is the name of the method that stops it. Both methods must belong to the same interface, as identified by iface, which includes the finder target name.

cbname is the name of the parameter of the start and stop methods that specifies the XRL callback through which data would be pushed. pfxname is the name of the parameter that specifies the prefix to be placed on the names of each of the data arguments supplied by the MP. In summary, pfxname and cbname identify the recv-id parameters.

mediate is a structured string describing ctrl parameters also required when starting or updating an interest. In future developments, it may also allow the mode of mediation to be specified on these parameters. It currently has no effect, and should be left empty.

get_push retrieves parameters for the specified method name (and its partner) in the interface iface. This allows one TCI to query the information about an MP method available to another TCI.

3.2 Asynchronous XRL Implementation

The EUA is derived from a version of XORP that had a limitation in how XORP processes can implement XRL methods. XORP processes are designed to be single-threaded, operating via an event loop. An event could be the arrival of an XRL request or response, or a timed event, or the availability of data on a socket. The servicing of each event must complete before the next event is handled. This approach requires that a client invoking an XRL cannot block while waiting for an XRL response event, as it would be unable to service other events (e.g., XRL requests from other processes) while waiting. Instead, the client sets up a callback function (normally in C++) to service the response event, and then yields control back to the event loop. It might service several XRL requests from other processes while waiting for its own request to be answered.

In contrast, when an XRL request is received, the process's C++ method that implements the XRL must pass the XRL results back as it returns control to the event loop. Although it can initiate new calls while it has control, it cannot use the results of those calls to provide results to its own caller, as it cannot receive the results it needs before returning to the event loop.

3.2.1 The Call-Chaining Problem

Figure 1 shows what happens at the client and server normally. At the client, some XORP-provided entity such as the event loop (EL), invokes application code. The application decides to make a call to the server, and prepares a callback (CB) to receive the out-arguments. Then it submits the call to the local XORP stack, providing in-arguments and CB. XORP generates an exportable reference to CB, and transmits that with the in-arguments to the remote node. It then returns control immediately (i.e. without waiting for a response) to the application, which then returns control to its caller, which could be the event loop. The out-arguments are returned with the exportable CB reference. When the EL is able to, it dispatches these to the CB indicated by the reference. Importantly, whether the response arrives before or after the application returns control to the EL, the EL cannot deliver the out-arguments until that control is returned. On the server side, the in-arguments are received and dispatched to the application. The application retains control while it computes the results, then returns them as it yields control back to the EL.

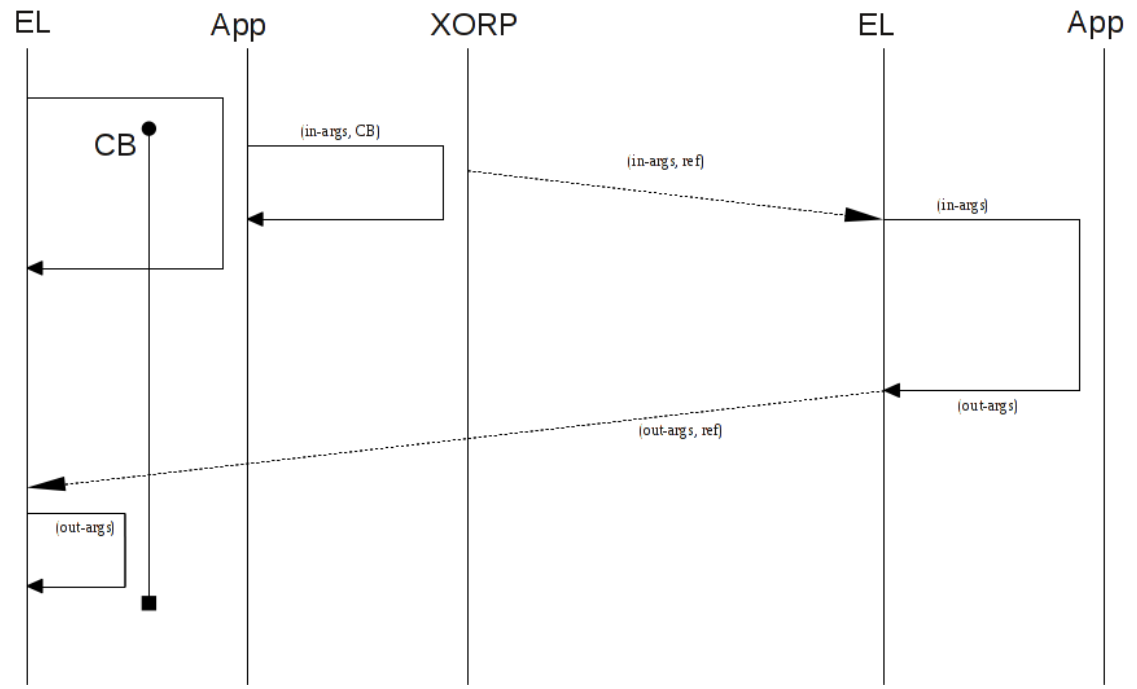

Figure 1: **Basic XORP RPC**

Figure 2 shows what happens if a node acts as both a client and server, when in order to service a call (1), the node must also make a call (2) of its own, and use this nested call's results to compute the outer call's results. Call 1's in-arguments are received and dispatched to the application. It prepares to make call 2 by creating CB2 and then submits it with the in-arguments for call 2. It must then yield control back to the EL for the out-arguments of call 2 to be dispatched to its CB2. However, in doing so, it must return the out-arguments for call 1 immediately, i.e., before it receives out-args-2. Therefore, out-args-1 cannot be computed from out-args-2.

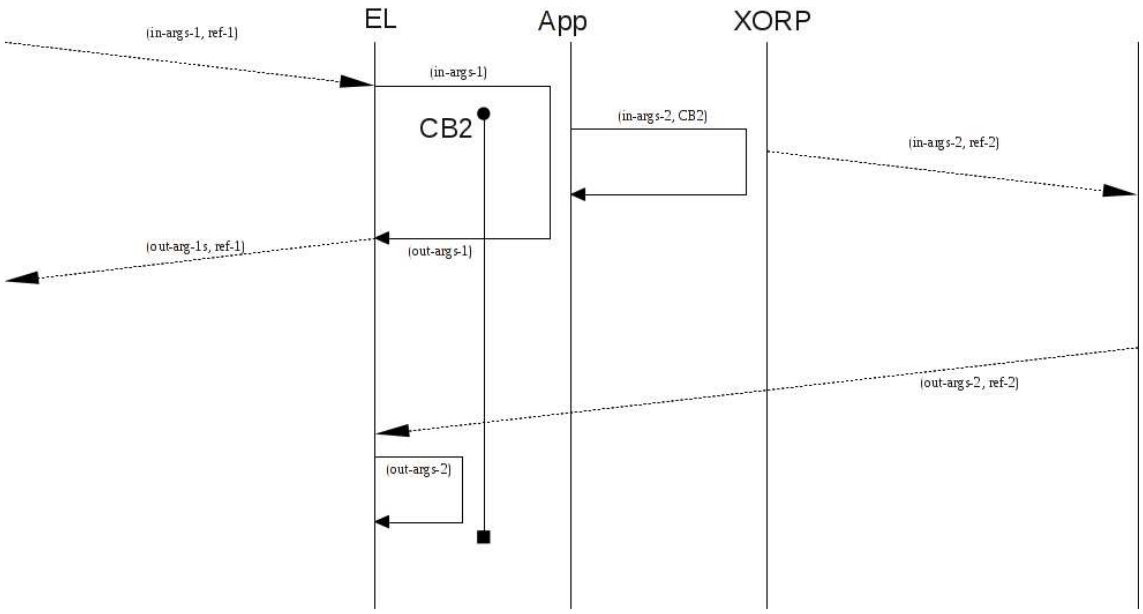

Figure 2: **Server acting as client**

Figure 3 shows one work-around for the problem. Having initiated call 2, the application *delegates*, rather than *yields*, control to EL, allowing it to delay the return of out-args-1 until out-args-2 has been delivered to CB2.

However, this can lead to other problems. The first problem occurs if a second call is received by this node, and is answered by the delegation to the event loop; it will have to be completed before the original call can be. The second is that it requires the event loop software to be re-entrant, which is not a guaranteed behavior.

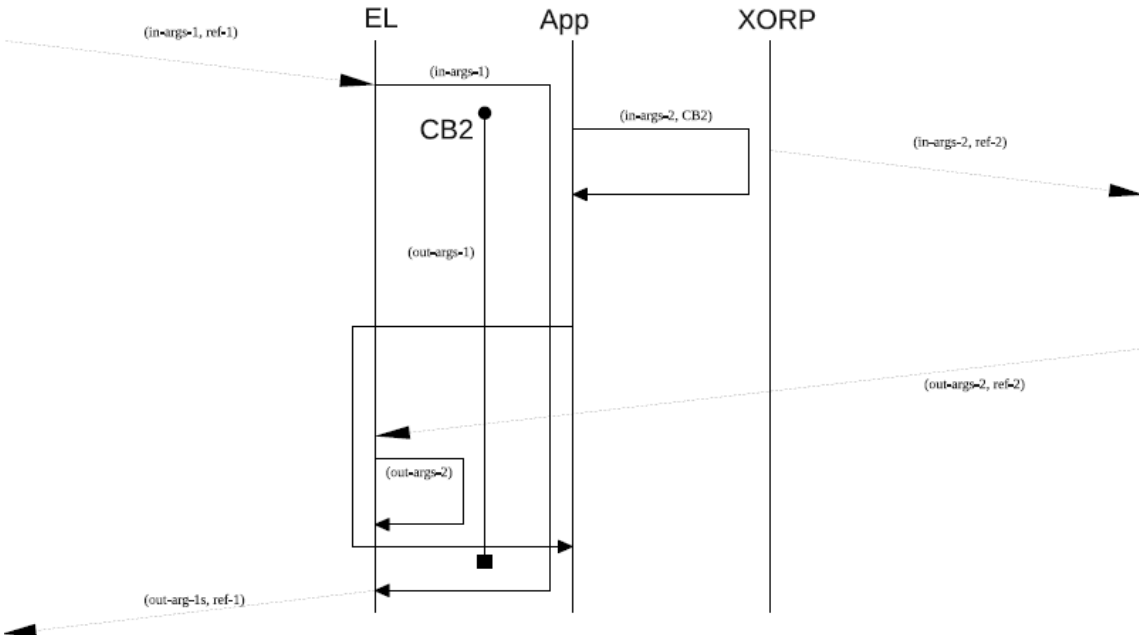

Figure 3: **Re-entrant call to event loop**

Figure 4 shows the alternative solution. The original duplex call has been split into two simplex calls; the duplex call's out-arguments become the second simplex call's in-arguments. The server returns call 1 immediately, and later receives the out-arguments it needs from the call that it initiated to service call 1. It then computes what would have been out-args-1 from out-args-2, and passes them as the in-args of the second of the simplex calls back on the original node.

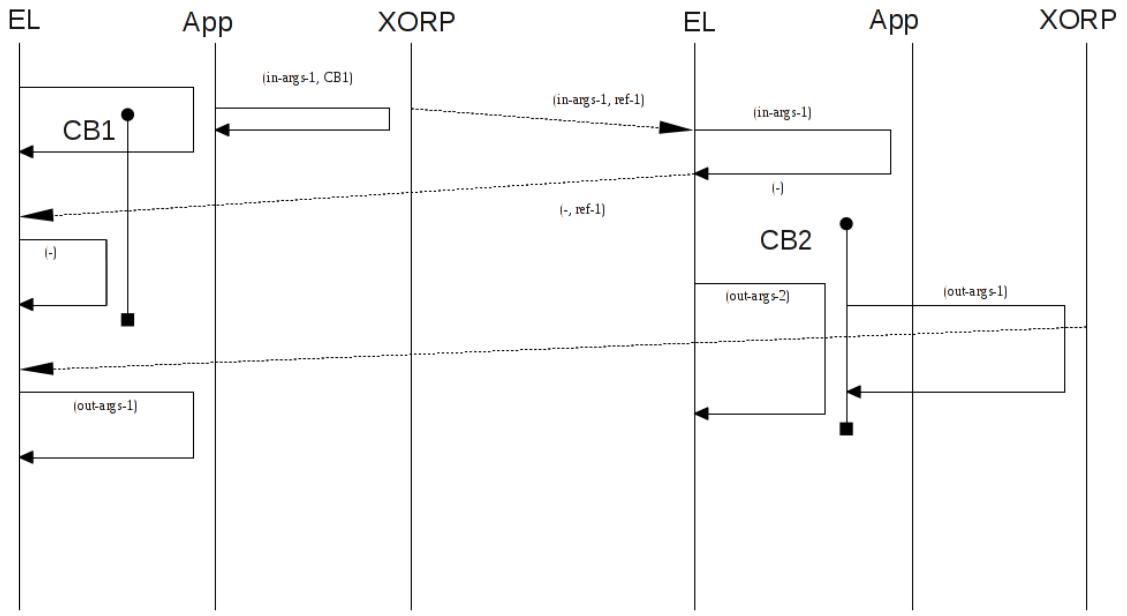

Figure 4: Explicit Callback

Figure 5 shows a preferred solution, which has been implemented by making changes to the XORP trunk, and applying these changes selectively to the EUA. The server's event loop receives the incoming call 1, and sets up a callback CB1'. It is passed with call 1's in-arguments to the application code implementing the method. The server initiates call 2 by first preparing its callback CB2, and storing CB1' as context for that callback. Then it submits call 2's in-arguments with its own callback, and can then return control to EL immediately. Call 2 is completed when out-args-2 are delivered to EL, and forwarded to CB2. CB2 computes the results of the original call from out-args-2, and passes them to CB1'. CB1' then completes the original call by passing the results back to the client.

Note: a multi-threaded solution would be difficult to consider. First, if an XRL's implementation were to invoke a new thread without any special changes to the XORP libraries, that implementation would still not be able to return control to the event loop until the results from the thread had been obtained (because of the way the XRL implementation and the XORP framework interface), so the event loop would still be blocked. Second, XORP's libraries give full control over the handling of incoming XRL requests to the event loop, which would have to be replaced with something that could dispatch to multiple threads. Every other piece of XORP code assumes that it is executing in a single-threaded, event-driven environment, and would suddenly be faced with new concurrency issues. Even if such a change was only applied to a process which needed the feature, the whole process would be affected by concurrency issues, rather than just the specific XRLs that needed to take advantage of the feature.

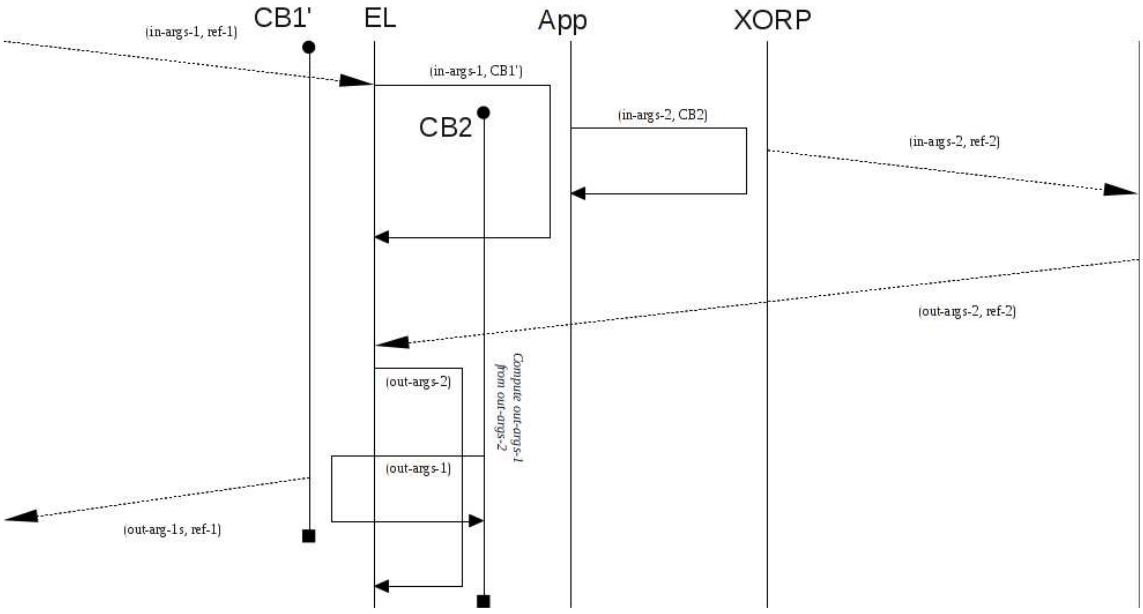

Figure 5: **Server acting as client with asynchronous reply**

3.2.2 Providing Asynchronous XRL Implementations

The feature is enabled with enable_async_server=True as an argument to scons when building and installing XORP. Its presence can be tested at compile time with the macro XORP_ENABLE_ASYNC_SERVER:

```
#ifdef XORP_ENABLE_ASYNC_SERVER
  // asynchronous implementations enabled
#else
  // ordinary XORP
#endif
```

When enabled, an XRL method can be implemented asynchronously by overriding a generated C++ method.

If the XRL method is eua_ping_mp/0.1/ping, one would normally implement that by fulfilling the abstract method eua_ping_mp_0_1_ping:

```
XrlCmdError
XrlEuaPingMpNode::eua_ping_mp_0_1_ping(const IPv4 &host,
                                       const uint32_t &tries,
                                       const uint32_t &period,
                                       uint32_t &delay)
{
  // ... Work out the delay ...

  // Provide the results and return.
  delay = ...;
  return XrlCmdError::OKAY();
}
```

Note: this method receives all in-arguments host, tries and period, and must supply the single out-argument delay before returning a status code.

It is still possible to provide that C++ method if a synchronous implementation is sufficient. However, to provide an alternative, asynchronous implementation, you must provide an additional function (overriding one generated by XORP from XRL interface files):

```
void
XrlEuaPingMpNode::async_eua_ping_mp_0_1_ping(const IPv4 &host,
                                             const uint32_t &tries,
                                             const uint32_t &period,
                                             EuaPingMp01PingCB cb)
{
  // ... Work out the delay ...

  // Provide the results and return.
  delay = ...;
  cb->dispatch(XrlCmdError::OKAY(), &delay);
}
```

This method differs from its synchronous counterpart in that it returns void instead of XrlCmdError, out-parameters are replaced with a single InterfaceVersionMethodCB object, and its name is prefixed with “async_”. This method does not have to call cb->dispatch before returning:

```
void
XrlEuaPingMpNode::async_eua_ping_mp_0_1_ping(const IPv4 &host,
                                             const uint32_t &tries,
                                             const uint32_t &period,
                                             EuaPingMp01PingCB cb)
{
  // Create a structure to hold ping info.
```

```
  ref_ptr<PingTask> info = new PingTask(host, tries, period, eventloop, cb);
  tasks.push_back(info);

  // Set it running.
  info->start();

  // Assume that PingTask will call cb->dispatch later.
}
```

Note that it is still necessary to provide a synchronous implementation, as it is declared originally as a pure virtual function:

```
XrlCmdError
XrlEuaPingMpNode::eua_ping_mp_0_1_ping(const IPv4 &host,
                                       const uint32_t &tries,
                                       const uint32_t &period,
                                       uint32_t &delay)
{
  UNUSED(host);
  UNUSED(tries);
  UNUSED(period);
  UNUSED(delay);
#ifdef XORP_ENABLE_ASYNC_SERVER
  return XrlCmdError::COMMAND_FAILED("asynchronous calls not enabled");
#else
  return XrlCmdError::COMMAND_FAILED("unreachable code");
#endif
}
```

cb->dispatch(...) can be invoked because the -> operator just yields the XORP Callback reference providing the dispatch method. However, this requires all arguments to be provided whether results are being returned:

```
  EuaPingMp01PingCB cb; // parameter
  delay = ...;
  cb->dispatch(XrlCmdError::OKAY(), &delay);
  // You have to pass OKAY!
```

or reporting an error:

```
  EuaPingMp01PingCB cb; // parameter
  cb->dispatch(XrlCmdError::COMMAND_FAILED("Failure"), NULL);
  // You have to pass NULL for each argument!
```

Furthermore, each result argument can only be passed by storing in a variable, and returning that variable's address. There is no means to pass a literal or an expression:

```
  EuaPingMp01PingCB cb; // parameter
  cb->dispatch(XrlCmdError::OKAY(), sum / successes); // error
```

Instead of using cb->dispatch, there are two methods which can be called on cb directly:

```
  EuaPingMp01PingCB cb; // parameter
  delay = ...;
  cb.respond(delay);
  // No need to pass OKAY!

  EuaPingMp01PingCB cb; // parameter
  cb.respond(sum / successes);
  // No need to pass a pointer!

  EuaPingMp01PingCB cb; // parameter
  cb.fail(XrlCmdError::COMMAND_FAILED("Out of memory!"));
  // No need to pass any NULLs!
```

4. User Guide

4.1 Configuration

XORP requires a config file to be present when it is executed to set its initial values. XORP is capable of generating a config file but for use with the EUA a sample configuration file has been provided that will initialise some standard variables within XORP for use with the EUA and help users get started within the framework.

The sample configuration file is located in the root directory of the source tarbull and is named eua-test. A directory needs to be created within /usr/local/xorp/ called etc within which the config file should be placed. XORP looks for a file named xorp.conf by default so the simplest method to get up and running is to copy the eua-test file to /usr/loca/xorp/etc/xorp.conf. The following commands (executed within the root directory of the EUA extracted tarbull) will create the directory and copy the config file to the correct destination:

- sudo mkdir /usr/local/xorp/etc
- sudo cp eua-test /usr/local/xorp/etc/xorp.conf

An example of EUA configuration file can also be found in Appendix A in this document.

4.2 Starting XORP and the EUA

There are two main programs that need to be running for the EUA to function correctly. The first is the TCI resolver which the EUA communicates with to resolve remote TCI's. The second is the EUA XORP implementation which provides the bulk of the functionality.

4.2.1 TCI Resolver

The binary for the TCI resolver can be found in the resolver/bin/ directory after following the compilation instructions in Section 2.

Running the resolver is a simple matter of starting the resolver binary. This will open a socket on the host machine allowing EUA's to communicate with it. The default port that the resolver listens on is 3490 but this can be changed by altering the PORT #define in resolver.h.

The resolver can be run on a different machine to XORP if required. In fact if a number of instances of the EUA are going to be running it is preferential to have one central resolver that all the EUA's can communicate with.

4.2.2 EUA XORP Implementation

XORP consists of two main parts. The router manager is the backend for the router and is responsible for executing the appropriate code based on commands entered into the XORP shell (the other integral part of XORP).

The simplest method to get started once the resolver has been started (and a config file for XORP is in place) is to simply start the router manager and XORP shell. The binaries for the router manager and the XORP shell are both located in the directory /usr/local/xorp/sbin/.

The router manager needs to be started first as the XORP shell needs this to be running so it can connect to it. The command to start the router manager is:

- sudo /usr/local/xorp/sbin/xorp_rtrmgr

The router manager utilises a number of 2 second delays when it loads to ensure certain tasks have fully completed before others are started. With the EUA it is very important that the router manager is allowed to fully load before the XORP shell is initiated because certain operations (such as the TCI registration) need to have completed before the XORP shell connects. Because of this it is important to wait until a line similar to the following is visible in the router managers output before proceeding:

```
[ 2011/07/29 14:05:14.676817  INFO xorp_rtrmgr:4534 RTRMGR
rtrmgr/task.cc:2242 run_task ] No more tasks to run
```

As soon as the "No more tasks to run" line has been seen the XORP shell needs to be started. The binary for this is in the same location as the router manager and is called xorpsh. The following command can be used to start it:

- sudo /usr/local/xorp/sbin/xorpsh

There are a number of environment variables that can be set to alter the behaviour of the EUA via the router manager. Because it is most often the case that sudo is used to execute commands, the environment variables are best set through the command directly as some configurations of sudo will not let it inherit from its parent shell for security reasons. If this is the case the following form should be used.

- sudo XORP_PF=t /usr/local/xorp/sbin/xorp_rtrmgr

This would execute the /usr/local/xorp/sbin/xorp_rtrmgr command and set the XORP_PF environment variable to t.

4.2.2.1 PF_INET vs PF_UNIX

XORP uses UNIX-domain sockets (PF_UNIX) by default, which do not permit inter-host communication to make the router more secure. You need to enable PF_INET sockets by setting an environment variable:

- XORP_PF=t

4.2.2.2 Binding to a Physical Interface

Even if PF_INET sockets are used, XORP still only binds to localhost/127.0.0.1, so it is still only accessible locally. To override this, two steps must be taken:

1. Tell the finder within the XORP node to use your physical interface with the -i command-line switch.
2. Tell all processes within XORP to use the same physical interface with the XORP_FINDER_CLIENT_ADDRESS environment variable.

So, if your physical interface is 10.1.18.21, use this:

- sudo XORP_PF=t XORP_FINDER_CLIENT_ADDRESS=10.1.18.21 xorp_rtrmgr -i 10.1.18.21

4.2.2.3 Access Control

Before any remote node tries to communicate with your router, it will talk to the router's finder, which performs some access control to ensure the process

trying to communicate has required permission. Tell the finder to allow specific hosts with -a ip-addr; use as many as necessary on the xorp_rtrmgr command. You can also use -n subnet for an address range.

4.2.2.4 TCI Resolver

When the TCI starts, it immediately connects to a resolver at TCP address localhost:3490. In practice, a resolver will have to be set up and made visible to all routers, and the default overridden by setting TCI_RESOLVER:

- TCI_RESOLVER=our-resolver.ecode.eu

If the port number is not the default of 3490 this also needs to be included. If for example the port being used is 4800 the following command could be used:

- TCI_RESOLVER=our-resolver.ecode.eu:4800

4.2.2.5 TCI Name

When the TCI contacts the resolver, it registers itself under the default name TCI_ID. This will need to be overridden in any multi-TCI deployment, by setting the TCI_ID environment variable. For example:

- TCI_ID=lancs_tci

4.2.2.6 Full Example Router Manager Command

The following is an example command setting all of the environment variables when starting the router manager.

- sudo XORP_PF=t PING_TCI= TCI_RESOLVER=fake-resolver.ecode.eu XORP_FINDER_CLIENT_ADDRESS=192.168.0.9 /usr/local/xorp/sbin/xorp_rtrmgr -i 192.168.0.9 -n 192.168.0.0/16

This command will start the router with the following options:

- XORP Protocol Family as TCP
- PING_TCI is used by the ping test. If blank, it performs a local dispatch or direct_dispatch. Set it to the same as TCI_ID (whose default value is TCI_ID), and it will do a remote call on itself. In this example it is blank so a local dispatch will be performed
- The TCI Resolver is set to fake-resolver.ecode.eu, as no port has been specified the default port 3490 will be used.
- XORP will use the physical interface which the IP address 192.168.0.9 is bound to to listen on.
- The Finder will run on the interface 192.168.0.9 as specified by the *-i* flag.
- The subnet 192.168.0.0/16 will be allowed to communicate with the finder as specified by the -n flag.

4.3 Using the EUA via the XORP Shell

The XORP shell allows users to configure the router manager and processes that it is running. This includes the EUA and associated XORP processes. This section describes how to enable and configure these processes.

Before any commands can be issued within the XORP shell it is necessary to first enter the configuration mode. This can be achieved by typing the configure command when the shell has started. Within the shell it is possible to see what commands are available at any time with the ? command.

4.3.1 Enabling and Configuring the TCI

4.3.1.1 Enabling the TCI

Once configuration mode has been entered, the TCI requires enabling via the XORP shell before it can be utilised. This can be performed by issuing the following commands:

- set eua tci enable true
- commit

The first command tells the shell the command to execute.

The second command commits any previously uncommitted commands and executes them. It is important to remember to commit as without it the commands won't be executed.

4.3.1.2 TCI Configuration Options

There are 4 variables which can be set via the XORP shell to configure the TCI. The four commands allow the following functionality to be performed.

1. Maximum number of MLP's the TCI will allow to connect
2. Maximum number of MP's the TCI will allow to connect
3. The priority of MLP's
4. The priority of MP's

The format for commands one and 2 is as follows:

1. set eua tci mlp-pritory <MLP ID> prio <Priority Unsigned Integer Value>
2. set eua tci mp-pritory <MP ID> prio <Priority Unsigned Integer Value>

After either of these commands has been used it is important to commit using the *commit* command within the shell. The format of commands 3 and 4 from the list above is as follows:

1. set eua tci max-mlp <Max MLP Unsigned Integer Value>
2. set eua tci max-mp <Max MP Unsigned Integer Value>

Again after either of these commands has been issued it is important to commit them.

4.3.2 Inbuilt Test MP's and MLP's

The EUA comes with some inbuilt processes as examples of MLP's and MP's. They also serve as a method to test some of the TCI's functionality. The two processes that are bundled with the EUA are the Ping MP and a testing MLP, Test MLP. The Ping MP offers some rudimentary ping functionality and the testing MLP contains some functionality for testing different areas of the TCI.

To enable the processes the following commands are used:

- Ping MP - set eua ping_mp enable true
- Test MLP - set eua mle_test enable true

Once either of these commands has been issued it needs to commit as previously described.

The Ping MP has no other commands associated with it as it is simply used by other EUA XORP processes to perform measurements.

The test MLP has some functionality that can be enabled via the shell for testing different aspects of the TCI. The different commands that can be used on the test MLP are as follows (all must be prefixed with set eua mle_test when inputting the commands to the XORP shell):

- test_discover_mp - Test the TCI discover_mp functionality on the PING-MP MP
- test_dispatch - Test the TCI dispatch functionality
- test_get_mps - Test the TCI get mps functionality
- test_get_ping_methods - Test the TCI get_mp_methods functionality on the PING-MP MP

Although the purpose of these commands is primarily for testing elements of the TCI they have been included with the EUA as they give a good basis for users to see what goes on behind the scene when they are executed. This is true both for how the router manager reacts to the commands (aided by the debug output from the router manager) and to the code that is used to implement them. Examining the code for these commands gives a good basis for writing commands in user XORP EUA processes.

Appendix A – Sample Configuration File

```
* XORP configuration file
*
* Configuration format: 1.1
* XORP version: 1.8-CT
* Date: 2010/10/21 00:23:39.145335
* Host: sim1
* User: root
*/

eua {
    mle_test {
        enable: false
    }
    mp_test {
        enable: false
    }
    ping_mp {
        enable: false
    }
    tci {
        enable: false
        max-mp: 16
        max-mlp: 16
    }
}
rtrmgr {
    config-directory: "/etc/xorp"
    load-file-command: "fetch"
    load-file-command-args: "-o"
    load-ftp-command: "fetch"
    load-ftp-command-args: "-o"
    load-http-command: "fetch"
    load-http-command-args: "-o"
    load-tftp-command: "sh -c 'echo Not implemented 1>&2 && exit 1'"
    load-tftp-command-args: ""
    save-file-command: "sh -c 'echo Not implemented 1>&2 && exit 1'"
    save-file-command-args: ""
    save-ftp-command: "sh -c 'echo Not implemented 1>&2 && exit 1'"
    save-ftp-command-args: ""
    save-http-command: "sh -c 'echo Not implemented 1>&2 && exit 1'"
    save-http-command-args: ""
    save-tftp-command: "sh -c 'echo Not implemented 1>&2 && exit 1'"
    save-tftp-command-args: ""
}
```