Smart Proxies in LuaOrb: Automatic Adaptation and Monitoring

Noemi Rodriguez , Hélcio Mello

[1]PUC-Rio, R. Marquês de São Vicente, 225, Gávea, Rio de Janeiro — RJ, Brazil

Abstract. *The LuaOrb project uses reflective features offered by the Lua programming language and by CORBA itself to create a platform that combines simplicity and flexibility in its support for dynamic behavior. In this paper we describe ongoing work on smart proxies. Smart proxies, based on simple QoS descriptions and event-based monitoring facilities, substitute traditional stubs, but automatically react to changes in run-time conditions, The paper also discusses support for dynamic stubs, which can implement generic adaptation procedures or performance-enhanced access to servers. Finally, we discuss an example of dynamic adaptation with smart proxies.*

1. Introduction

Over the last years, the need for adaptation in the context of mobile computing and embedded systems has led to much work on new middleware platforms that allow their own structure to be dynamically redefined [rm0, 2000, ref, 2003]. In this work we explore a slightly different approach: how to support adaptation with the reflective facilities available in a more traditional middleware platform like CORBA. As we have already discussed in [Batista et al., 2003], the CORBA specification contains a series of reflective mechanisms that allow programs to query their run-time environments and determine current properties and services. The dynamic invocation interface (DII) and the dynamic skeleton interface (DSI) allow the structure of applications and services to be dynamically redefined. However, due to the complexity of using and understanding these mechanisms, few systems seem to take advantage of these facilities.

We have been studying for some time how the use of an interpreted language with reflective features can help alleviate this problem. The LuaOrb binding [Cerqueira et al., 1998] explores reflective characteristics of CORBA and of the Lua programming language [Celes et al., 1996] to create an environment in which CORBA objects can be dynamically accessed in the same way as any other Lua object. Libraries such as LuaRep and LuaTrading [Batista et al., 2003] use this binding to provide easy access to CORBA information repositories. Using LuaOrb and these libraries, the programmer can build applications that monitor the environment and adapt to its conditions by locating servers that are adequate to their needs.

In this work, we add further facilities to this platform: more specifically, the support for smart proxies, which act as stubs with internal adaptation behavior. Smart proxies with standard adaptation procedures can easily be created, dynamically, from simple QoS descriptions, but the (smart) programmer is also given the choice to code specific adaptation actions. This is in line with our belief that a scripting language such as Lua is well suited for different classes of users. Technically advanced users can use all the expressiveness of the language (and, in this case, of the libraries and tools) to code adaptation procedures. Less sophisticated programmers can use the standard adaptation procedures or the ones created by administrators for their specific applications and environment.

The use of an interpreted language such as Lua also allowed us to explore downloadable stubs (often mentioned here as *dstub* for short), similar to the ones used in Java RMI, and their uploadable counterparts (also called *ustubs* in this paper). Although it is possible to use them without any extra support, those stubs can also be combined with LuaProxy, yielding an alternative, dynamic adaptation approach.

The paper is organized as follows. Section 2 reviews the existing platform, describing the LuaOrb binding and associated libraries. Section 3 discusses our implementation of smart proxies. The following section describes support for dynamic stubs. In Section 5, we present an example of adaptation through smart proxies. Finally, sections 6 and 7 contain some final remarks and compare our work with similar ones found in the literature.

2. Existing Environment

This work builds on several existing tools, most of them aimed at providing easy-to-use interface layers above CORBA, as well as convenient means of performing adaptation-specific activities, like monitoring.

2.1. LuaOrb

LuaOrb[Cerqueira et al., 1998] is a binding of the CORBA[Group, 1999] specification to the interpreted language Lua[Celes et al., 1996]. One of the main design aspects of Lua is its simplicity and flexibility. In Lua, variables are neither declared nor typed, although the values they hold belong to a particular type, such as *string*, *number*, etc. Hence, the same variable can be used to hold values of different types at different moments.

Two types in Lua deserve special attention: *table* and *function*. Tables implement associative arrays, that is, their indices can be values of arbitrary types, not only numbers as in ordinary vectors. A table can thus be used as a general-purpose container, and is often used to emulate traditional object-oriented features.

Functions, in turn, are first-order values, i.e. they can be passed as arguments to or be used as return values from other functions, and even be stored in a table. By doing so one can implement an “object” by storing its methods (Lua functions) as fields of a table.

Lua also supports *meta-tables*, a mechanism for handling some events like indexing non-existent fields of a table. When such events happen, a user-defined *meta-method* is called to handle them. For example, for non-existing fields in a table the meta-method could return a default value (eg. zero in sparse matrices). A detailed example of meta-tables usage is beyond the scope of this paper, but they are the key to implementing proxies in LuaOrb. LuaOrb implements a CORBA proxy by means of a table that employs the meta-table mechanism to automatically and transparently fetch the remote method or attribute in question. From the point of view of those who use the proxy, it behaves as the remote object itself.

The traditional CORBA programming sequence consists in creating an IDL, compiling it to generate stubs and adding user-specific code. Because of the interpreted nature of LuaOrb, however, there is no way to generate compiled stubs. Instead, LuaOrb uses DII (Dynamic Invocation Interface) to invoke remote methods, building the necessary data structures at run time. The meta-tables mechanism mentioned in this section shields the users from low-level DII details, encouraging them to build more dynamic applications.

2.2. LuaRep & LuaTrading

The Interface Repository and the Trading Service are key CORBA services in the task of providing the user with support for adaptive programming. The complexity of their interfaces makes their usage rather cumbersome; the programmer must make several method calls, create complex data structures and sometimes perform some sort of post-processing of the results.

LuaRep and LuaTrading[Batista et al., 2003] provide the user with mirror repositories to the CORBA services They are implemented as Lua tables associated with meta-methods that transparently retrieve or update the information within those repositories. Many tasks can be carried out by simply reading from or assigning to a table. For instance, in order to create a new service type, it suffices to write its description in the table that mirrors the service types repository.

The meta-methods of those mirror repositories perform all low-level activities such as translating user-supplied high-level arguments (e.g. "long" into "tk_long"). Also, additional methods that act on those repositories have been designed to use default arguments. For instance, the following line of code imports all service offers of *FOO* service type:

```
offers = importServiceOffers {type = "FOO"}
```

With similar simplicity it is possible to iterate over the Interface Repository, obtain and modify interfaces descriptions as well as create new modules and interfaces.

2.3. LuaMonitor

In order to have applications that dynamically adapt to runtime conditions it is clearly necessary to provide means for them to detect changes in the environment.

One approach towards that end is to use entities that continually inspect the value of a given *property*, called *monitors* herein. The LuaMonitor library [Moura et al., 2002] tackles this issue in the context of LuaOrb, maintaining a dynamic approach. It supports the dynamic definition of objects that support basic periodic monitoring features, *aspects* and *event observers*.

Aspects can be thought of as secondary properties that derive from the main ones and possibly from other aspects as well. One possible use for aspects is to keep track of statistical data about the main property, like average, variance, etc. Whenever the monitored value is updated, so are all of the monitor's aspects.

An event observer is an object responsible for taking notifications of certain events. In practice, when an application wishes to be notified of a particular event, it registers an observer at the monitor watching the property in question. When the event happens the observer is notified and the application may react accordingly.

For example, suppose a monitor has been designated to watch the temperature of a boiler, and that the control application wishes to be notified of excessive temperature increases. Assuming the maximum acceptable temperature is 110^oC, the application simply registers itself as an observer at the monitor under the condition "temperature $>$ 110". When that expression evaluates to *true*, that is, the temperature raises above 110^oC, the observer is notified my means of a *notifyEvent ()* method call.

The application is then able to react to the temperature increase, for instance by increasing the amount of cold water flowing through the boiler, reducing the amount of power supplied to it, etc.

Another approach for monitoring a property would be assigning a thread for polling it periodically. Doing so would not be very efficient, because it would require one extra thread for each property being monitored. Even if a single thread were to poll all desired properties, there would be a significant amount of traffic in the network due to the polling messages and their replies.

Moreover, most of the replies would be of little use, because many applications are not interested in every single change in the monitored property. Instead, they usually want to know only when they exceed a certain threshold or drop below it. Finally, there would also be the need to cope with the complexity of multithread programming and the inherent synchronization issues.

LuaMonitor runs asynchronally with respect to the client. Hence, while a monitor periodically polls a property, the client may perform any other task. Only upon notification the observer needs to worry about the monitored property. Also, because monitors often run at the same host as the server which holds the property there are no polling remote calls, thus saving network bandwidth.

The monitored property (or any of its aspects) can also be used as a dynamic property in a trader service offer. That combination of monitoring and use of trading service is very powerful. Typically, the exported service offers contain both the dynamic property (or aspect) being monitored and a reference to its monitor. When a client imports such offers it obtains access to the monitor and can register observers to be notified of relevant changes of the property in question. Upon notification, the client may handle the event as desired, for instance querying the trader for a new offer.

3. Smart Proxies

In distributed computing, the interaction between a client and a server usually takes place by means of a proxy. This proxy is a local object from the client's point of view, which forwards all method calls to the remote object. Ordinary proxies behave in a rather "static" fashion; they seldom support changing the remote object they refer to.

Under appropriate run-time conditions the above approach will yield acceptable results. Many real applications, however, are liable to unexpected changes in the environment they run. Servers can be encumbered with an overwhelming quantity of both local and remote requests, or even crash; networks can experience momentaneous traffic peaks, partitions and so on. It is very difficult (or even impossible) to foresee all such hazards at compile-time and properly handle them. Because of that applications should dynamically adapt to overcome those problems. LuaProxy is the implementation of a "smart" proxy; one capable of reacting to unpredicted conditions as the ones discussed above. While a conventional proxy usually takes just a reference to the remote object, LuaProxy constructor needs extra arguments that specify what the desired QoS levels are and how to adapt to relevant changes in available resources. These data structures are discussed in the next sections.

3.1. QoS Descriptors

Each application may require different constraints over a set of QoS parameters, such as delay, bandwidth, etc. QoS descriptors allow programmers to specify lists of constraints, called *QoS attributes*. Typical examples of such attributes are "delay $<$ 30", "bandwidth $>$ 80", and so on. Optionally, they may also be followed by a comma and a priority, like in "delay $<$ 30, high", where a delay under 30 ms is required with a high priority.

A QoS descriptor can also inherit from others, so as to facilitate descriptor reuse. The constructor syntax makes that straightforward: it takes a sequence of strings (the QoS attributes) and the descriptor parents. For instance, if `d1` and `d2` are valid descriptors then

```
d3 = QoSDescriptor (d1, d2, "delay < 30")
```

creates a new descriptor that inherits from both `d1` and `d2` and adds the atribute "delay $<$ 30".

The rules concerning the inheritance mechanism are simple: if an attribute is absent at a given descriptor then it is recursively sought for in the descriptor's parents, in a depth-first algorithm. The parents are inspected in the same order they were supplied to the constructor.

QoS descriptors provide a simple preliminary solution for QoS specification. The inheritance mechanism described here still does not take into account possible conflicts such as "delay $<$ 20" and "delay $>$ 30". More complex applications may need additional features, and are the target of future research, as discussed in Section 7.

3.2. Adaptation Procedure

LuaProxy's adaptation procedure is usually triggered by the violation of a QoS parameter. The monitor associated with that parameter notifies LuaProxy's observer, which in turn adds the violation to an event queue.

The next time the proxy is invoked, it handled events in the queue, and must then adapt to the QoS violation. The default adaptation behavior is to select another server from the *trading service* (Section 3.2.1).

In order to benefit from service-specific features, LuaProxy accepts a table that maps each possible event to an adaptation procedure, called *QoS implementation*. If a given event is not mapped, then it is handled by the default adaptation routine.

By means of a QoS implementation the user may customize the adaptation procedure for a particular application. Because QoS implementations can be organized as libraries, new ones can easily be created by inheritance, in the same fashion of QoS descriptors. The derived QoS implementation can override the adaptation procedure for any event or define new ones as desired.

As an example of user-specific adaptation routines lies in video on demand. If the server runs out of bandwidth, its monitor will signal the *bandwidth_decrease* event, thus triggering LuaProxy adaptation. The QoS implementation could handle that event by requiring the video server to diminish the frame rate or the color density. That would reduce the bandwidth requirements of the application without the user's intervention. Of course, the adaptation procedure may deem such reconfiguration unacceptable (for instance, if the video is already playing under this scarce-resource scenario) and choose to replace the video server instead.

3.2.1. Selection algorithm

When the user's QoS constraints can no longer be met and all adaptation routines fail, the LuaProxy *select ()* method is invoked to search for a new server in the trader. Here, LuaTrading (Section 2.2) plays an important support role, as service offers can be imported with a single line of code, and the low-level issues (such as the use of iterators) are properly handled.

The trader is queried according to the QoS descriptor (Section 3.1) specified by the user. The query constraint is the conjunction of all its QoS attributes (devoid of their priorities). For instance, if the QoS descriptor holds "delay $<$ 20, high" and "bandwidth $>$ 70, low" as its attributes the query constrait will be "delay $<$ 20 and bandwidth $>$ 70". Because a QoS descriptor may hold several attributes, it is possible that no available service offer meets them all. In that case, the trader is queried again, but without the low priority attributes. If the query still does not return any offers, a final attempt will be made to satisfy only the high-priority attributes. If again no server is found, an exception is raised.

In order to select the best suitable server, the query is set to sort its output so that the first offer yields the maximum *quality*. This quality is the result of the evaluation of a *quality expression*, such as “5 $\times$ bandwidth”, which yields greater results for higher bandwidth values, thus valuing servers with most available bandwidth. The default quality expression used by LuaProxy is a weighted sum of the quality of each QoS parameter, according to the priority of each parameter (see Section 3.1). The quality of a parameter, in turn, is an increasing function of the parameter in question if it was requested to be greater than a specific value, and a decreasing function otherwise. For example, if “bandwidth” was required to be *greater than* a certain value in a QoS descriptor then its *increase* will likely make the offer more attractive. Likewise, if a parameter (such as “delay”) is expected to be *under* a given limit then its *decrease* will probably make the offer look more suitable to the service in question.

Thus, one possible quality expression would be “5 $\times$ bandwidth + 100 / delay”, because the higher the bandwidth and the lesser the delay, the higher the quality expression will evaluate to. Of course, specific applications may need a better tuning of those constants. The user is allowed to override how those formulae are generated for specific QoS parameters, but that is beyond the scope of this paper.

4. Dynamic Stubs

Although LuaProxy itself can already achieve dynamic adaptation to some degree, additional features based on code mobility are also being investigated in our work. This section introduces dynamic stubs that allow client and server to transfer code between each other at run-time, according to *code on demand* and *remote evaluation* paradigms [Fuggetta et al., 1998].

4.1. Downloadable stubs

Although QoS implementation objects allow the user to have practically full control over the adaptation procedure, it is possible that a server may have valuable information about implementation details hidden behind the IDL interfaces. To take advantage of their peculiar features, remote servers may provide their clients with *downloadable stubs* (or simply *dstubs* for short).

To provide this facility, a server must implement the *LuaProxy::DStubProvider* interface (partially shown in Figure 1). LuaProxy detects whether the server supports dstubs by invoking the standard *_is_a ()* pseudo-operation against the *LuaProxy::DStubProvider* interface. If a dstub is available then it is automatically fetched and installed.

The stubs are simple structures that hold two fields: an enumeration that specifies the stub language and a sequence of octets that contains the stub code. The *language* field allows the stub to be implemented in any language that can interface with Lua, like C and Java. Based on that field, LuaProxy will perform the necessary steps to integrate the stub, such as loading the code as a C dynamic library or treating the octets as a chunk of Lua code. The current LuaProxy implementation deals with Lua implementations only.

After being downloaded, the stub is merged into the smart proxy at run time and acts as a broker to the remote object. From then on, all method calls will be intercepted by it. The stub may choose to either forward the call directly to the server or perform some pre- or post-processing activity. Possible uses include providing extra features such as caching, compression of transmitted/received data, etc.

```
module LuaProxy {
   enum LangEnum {LUA, C, JAVA};
   typedef sequence<octet> OctetSeq;

   struct Stub {
      LangEnum Language;
      OctetSeq Encoding;
   };

   interface DStubProvider {
      Stub getStub (in LangEnum lang);
   };
};
```

Figure 1: IDL for supporting downloadable stubs.

```
MyServant = {
   foo = function (self)
      print ("Inside foo ()")
   end,

   getStub = function (self, lang)
      return {
         Language = "LUA",
         Encoding = [[{
            foo = function (self)
               print ("Inside dstub.")
               self._proxy:foo ()
            end
         }]]
      }
   end
}
```

Figure 2: Dstubs usage example.

Figure 2 shows a simple example of a dstub. When a proxy is created for the *MyServant* object, the dstub is fetched and installed in that proxy. As can be seen in the figure, the dstub overrides the *foo ()* method, prints a message and then invokes the original method.

More interestingly, the downloadable stub can also be designed to actively participate in the adaptation procedure, i.e. intercept LuaProxy's *adapt ()* method. By doing so the stub may decide whether to invoke its own adaptation routine for a given event or leave it to be handled by LuaProxy. Another possibility is to allow state transfer when LuaProxy switches to a new server. A stub can accomplish that by overriding LuaProxy's method for server selection (*_select ()*). It should then select a new server itself, transfer the state from the former to the new server, and finally return the new server to LuaProxy. Section 5 demonstrates both of these techniques in a case study.

4.2. Uploadable Stubs

It is well known that communication latency is usually responsible for the greatest part of the execution time of remote calls. When several method calls must be issued on the same remote object, significant performance increase can be achieved by uploading code to the server. This technique is often referred to as *remote evaluation* [Fuggetta et al., 1998], and could also be employed, for instance, when a program runs where resources are deemed scarce.

Adaptation routines could benefit from this approach by uploading performance-critical code to the server, thus reducing the reaction time between the QoS violation and the corresponding system adaptation. The uploaded code would then either override methods in the server or simply extend its interface with the uploaded methods.

Uploadable stubs (*ustubs*) have therefore been developed to endow LuaProxy

```
module LuaProxy {
   interface UStubHolder {
      long insertStub (in Stub stub, in string ifname, out Object extobj);
      void removeStub (in long id);
   };
};
```

Figure 3: IDL for supporting *uploadable* stubs.

```
interface Matrix
{
   double getAij (in unsigned long i, in unsigned long j);
   long getn ();
};
```

Figure 4: *Matrix* interface.

adaptation routines with code upload capability. The target server must implement the *LuaProxy::UStubHolder* interface, shown in Figure 3. In order to make supporting ustubs easier, LuaProxy offers a default implementation servers may inherit from.

The *insertStub ()* method uploads and installs a stub in the remote object, returning an ID for eventual stub removal. The server remains intact, but the uploaded stub encapsulates it, yielding an *extended object*. Lua's meta-table mechanism is set to automatically forward a method call to the original server in case the ustub does not override it. Finally, a reference to the resulting extended object is returned as the out parameter.

The ustub may define methods originally not present in the server, extending its interface. In this case the extended object interface should inherit from both the server interface (so that the extended object still provides the former server functionality) and the one that holds the extensions. The latter is supplied as the second argument of *insertStub ()*. LuaProxy automatically creates a new interface that inherits from those two at runtime. If it is not necessary to extend the server's interface (just override its methods) then an empty string can be used instead.

Figure 6 shows how to extend a server with a ustub. The server implements the *Matrix* interface shown in Figure 4. Assuming p initially holds a reference to such server, the ustub augments its interface with a method to get its main diagonal (Figure 5). The resulting extended object is then used to get the diagonal with a single method call. Without the ustub, it would be necessary to invoke *getAij ()* n times.

It is important to note that the reference to the original server remains valid, as it probably should. If it did not, other clients of that server would also be affected by the upload, which is usually not desired. Should several clients need to interact with the extended object, its reference could easily be shared by other means (e.g. CORBA naming service).

The server extensions provided by ustubs offer a new adaptation approach for *QoSImplementation* objects. They can create extended servers as needed, and the ustub behavior is neither known nor noticed by the server. Among the possibilities, the server's interface could be augmented with data compression or encryption features, as shown in the next section.

```
interface NewMethods {
   typedef sequence<double> DoubleSeq;
   DoubleSeq getMainDiagonal ();
};
```

Figure 5: Exemple of interface extension.

```
ustub = {
   Language = "LUA",
   Encoding = [[{
      getMainDiagonal = function (self)
         local diag = {}
         for i = 1, self:getn () do
            table.insert (diag, self:getAij (i, i))
         end
         return diag
      end
   }]]
}

id, obj = p:insertStub (ustub, "NewMethods")
obj = luaorb.narrow (obj)
diag = obj:getMainDiagonal ()
p:removeStub (id)
```

Figure 6: Using the extended object.

5. Case Study

In order to test LuaProxy, a small application program has been created: *LuaEmpire*. It is a very simple strategy game where players can build spaceships to attack enemy fleets or conquer planets. The goal of the game is to conquer all enemy planets without losing your own.

In order to join a game, one logs in a remote server that hosts an ongoing match. The player can then issue commands (such as ship construction) in a graphical interface. The interface then translates user interaction (like pressing buttons) into remote method calls.

Those methods update the server state, i.e. a snapshot of the solar system and its fleets. The changes (updates) are periodically sent back to all players logged in, so that they can update their internal states and redraw any changed objects on the player's screen.

This application uses LuaProxy to find a suitable server running a match and to adapt to resource shortage during gameplay. The fluctuation of QoS parameters is simulated by a test script, so that the monitors in question trigger the adaptation procedures.

The test scenario proposed here consisted of two servers and two clients. Both servers support dstub and ustub features, but one of them initially had more available resources, so that both players selected it as their initial server.

During the match, the players created some ships and moved part of them to somewhere in space. In the meantime the server's available bandwidth was explicitly reduced, making the clients automatically request that the state updates be compressed. Because the server's interface had originally no support for compression at all, an ustub has been installed for that purpose.

Next, the players' hosts had their response times raised (simulating, for instance, the user running additional processes) so that they could not redraw the players' screen as often as before. The adaptation mechanism reacted to this event by buffering and merging every two consecutive updates into a single one, which was then delivered to the application. That task was carried out by means of a dstub, without user intervention.

As a final test, the server in use was manually shut down, forcing LuaProxy to select another one. Server crash detection is currently done by means of the standard CORBA pseudo-operation *_non_existent ()*. Although we acknowledge it is not a good fail detector, it served our prototyping purposes well.

```
module LuaEmpire {
   interface Client {
      oneway void update (in string changes);
   };

   interface Server {
      string login (in string nick, in Client cb);
      void clock ();
      void loadState (in string state);
   };
};
```

Figure 7: Part of LuaEmpire IDL.

Once again, the application-specific behavior of the downloadable stub played an important role. After the new server had been selected, its initial state was set to reproduce the last game state held by one of the players. After a negligible delay, the crashing server was replaced with another one that carried out the computation task from the point it was interrupted.

5.1. Code Details

The most important tasks the clients must perform are creating an instance of LuaProxy and redrawing the game screens as the server reports changes in the solar system. Creating the proxy is straightforward: the user simply states the desired interface (*LuaEmpire::Server* in this case) and the QoS requirements.

In our test we required the bandwidth to be greater than 56 kbps and the response delay to be lower than 10 ms. The proxy can then be constructed as follows (units are implicit):

```
d = QoSDescriptor ("bandwidth > 56", "delay < 10")
proxy = LuaProxy ("LuaEmpire::Server", d, qosimpl)
```

The above proxy will both work like an ordinary one and show adaptive behavior. The *qosimpl* argument holds the QoSImplementation to be used, possibly got somewhere else, like a library.

5.1.1. Game Architecture

Figure 7 shows the most relevant methods of LuaEmpire IDL. It defines two interfaces: *Client* and *Server*. The latter contains methods to allow players to login, interact with the solar system (creating and moving ships), etc. In order to join a match (*login ()*), a player must supply a callback object that implements the *update ()* method. The server uses this method to report changes in the state of the solar system. *login ()* also returns an initial state to the player.

Alternatively, *loadState ()* can be used to resume a match from the last known state (held by any player). It should be noted that all states and updates are encoded as strings. They hold Lua table descriptions that are interpreted by the clients, yielding a list of objects in the solar system and their attributes. For instance, if a solar system contained a single object the game state could be represented as “{[1] = *object_description*}”.

Transmitting data in that way has two major advantages. First, it is possible to update the state of a player or load the state of a server with a single method call, instead of one call for each object. Moreover, this data is not a simple raw sequence of bytes; it is structured data organized as a stringified Lua table, which is easier to handle. Second, compressing long data streams yields better compression ratios than several chunks ones

```
function login (self, nick, callback)
   callback.formerUpdate = callback.update
   callback.update = self.newUpdate
   callback.CompressUpdates = false

   self.Callback = callback
   return self._servant:login (nick, callback)
end

function setUpdateCompression (self, on)
   self.Callback.CompressUpdates = on
end

function newUpdate (self, changes)
   if self.CompressUpdates then
      changes = compress (changes)
   end

   return self:formerUpdate (changes)
end
```

Figure 8: Ustub code uploaded to LuaEmpire.

one by one. An analogous benefit would be achieved if encryption were used, because a malicious user would have to additionally identify objects boundaries within the stream.

The *clock ()* method is automatically called by a control process and serves as a synchronization point for players, like short game turns. All methods invoked by players only affect server state when *clock ()* is called. At that time, all objects changed since last turn are reported back to players by means of *Client::update ()* method.

5.1.2. Adaptation Code

Before any player-related task is carried out, initialization regarding adaptation procedures takes place. When a new server is imported from the trader, LuaProxy notifies its QoSImplementation object, so that it can install a ustub in the new server and remove the one installed in the former server.

At that time, the server is augmented with a method that switches the compression of game updates on and off. Additionally, the ustub overrides the *login ()* method in order to capture the player's callback object. The ustub then logs in on behalf of the player, but it intercepts the callback forthcoming *update ()* method calls, to compress game data if that feature is activated by then. Figure 8 summarizes the ustub code.

Dstubs also play an important role in LuaEmpire. When LuaProxy detected the servers supported them, it fetched and installed the dstubs as expected. The dstub also overrides the *login ()* method to capture the player's callback object. Its *update ()* method is redefined to adapt to excessive client-side delay by buffering server updates. For each two updates, they are merged into a single one, which in turn is passed to the player's original *update ()* method. Hence the player's screen is redrawn only once every two updates. A simplified version of the dstub code is shown in Figure 9.

6. Related Work

The idea of a smart proxy is not new. Similar approaches have been proposed in several works. The BBN Quality Objects (QuO) architecture [Bakken et al., 1995] from which we borrowed some ideas, defines entities that map to the ones in LuaProxy to some degree.

In LuaProxy, the desired QoS requirements are stated by QoS descriptors (Section 3.1) and adaptation is triggered by LuaMonitor notifications. The smart proxy then adapts

```
SkipOddFrames = false; SeqNumber = 0; LastUpdate = nil
oldUpdate = nil     -- Holds user’s update () method.

-- Intercepts login ()
login = function (self, nick, callback)
   -- Replaces the callback update () method
   oldUpdate = callback.update
   callback.update = newUpdate

   -- Invokes server
   return self._proxy:login (nick, callback)
end

-- Replaces the callback update ()
newUpdate = function (self, changes)
   SeqNumber = SeqNumber + 1

   if SkipOddFrames and isOdd (SeqNumber) then
      -- Buffers this update
      LastUpdate = changes
   else
      -- Merges the two updates and passes the result to the old update ()
      oldUpdate (merge (LastUpdate, changes))
   end
end
```

Figure 9: Dstub code provided by LuaEmpire servers.

to reported conditions the next time the user issues a method call on it. Likewise, in QuO, *system condition objects* measure QoS parameters and *contract objects* evaluate them into *active QoS regions*. QuO *delegates* then perform the necessary adaptation based on those regions.

However, QuO contracts are specified in CDL (*Contract Description Language*) and then converted into Java code, which in turn must be compiled against application-specific code. Thus, if the contract changes, its code must be regenerated and recompiled. The separation of concerns and the interpreted nature of Lua yield more run-time flexibility to LuaProxy, allowing QoS descriptors, implementations and monitors to be bound to the smart proxy or even be modified at run time. Actually, the adaptation procedures themselves are allowed to dynamically modify the QoS descriptor in use as they see fit (for instance, in order to reduce the requested bandwidth if data compression is activated). If manual changes to LuaProxy elements (QoS descriptors and implementations) are required, not even the running application needs to be halted; a few commands in a Lua console will do.

Both QuO and LuaProxy aim at providing the user with Quality of Service (QoS) features, although LuaProxy is still in an initial phase of development. It is important to note that QuO offers a complete infrastructure for QoS provisioning that can support applications with rigid requirements of reliability. We have focused on offering flexibility and ease of use, but in an environment more fit to applications with less stringent requirements.

The downloadable stubs have partly been inspired by JINI [Edwards, 1999]. JINI does not interact directly with CORBA, but it is possible to make a CORBA object visible to a JINI client[Newmarch, 2000]. In short, a JINI service is created to behave as a CORBA client. This sevice is then able to forward method calls between a CORBA object and a JINI client. LuaProxy, in turn, is CORBA-native and thus needs no bridges to interact with other CORBA objects. Also, a downloadable stub may be written in any language that interacts with Lua, not only Java.

The work published in [Silva et al.,] describes a framework implemented in Java to support adaptation in applications for mobile devices. It employs a monitoring mechanism similar to LuaMonitor that triggers adaptation routines, but the adaptation is also initiated if certain client-server interaction patterns are detected, as observed by interceptors.

Another approach [Chang and Karamcheti, 2000] measures the effects of various adaptation mechanisms on a single application before it is deployed. Based on the results, the application can automatically select which adaptation technique is the best suited to react to a given change in the execution environment. LuaProxy QoS implementations currently require the developer to decide which adaptation approach to use when more than one applies to the same resource shortage event. On the other hand, applications that use LuaProxy do not need to undergo any performance testing before they are deployed; moreover, because any modification to the application may potentially invalidate the performance tests, LuaProxy seems to be more adequate for projects that either evolve rapidly or are prototyping-based.

We are currently unaware of any project that gathers a monitoring/adaptation mechanism with code mobility in *both* directions and implements that in a flexible environment like ours, although similar mature proposals built on a more static fashion (like QuO) already exist.

7. Final Remarks

Although LuaProxy is still in a initial state of development, we believe it is a promising approach for supporting adaptation. The creation of smart proxies from QoS descriptors and implementations, as well as the server-specific code from the downloadable stubs, allow simple adaptation techniques to be used with hardly any cost to the programmer. The knowledgeable programmer, on the other hand, maintains the possibility of writing her own adaptation scripts.

Additionally, uploadable stubs may extend the server interface with new methods, as well as intercept existing ones. If an adaptation script employs ustubs it will be able to benefit from features such as encryption and compression, even if the original server was not designed to offer such facilities. With support for ustubs, the client could also, for instance, dynamically switch its behavior between that of a “fat” and that of a “thin” client.

It will be necessary to develop more self-adaptive applications to further evaluate the LuaProxy facilities. We are interested in looking at mobile applications, where downloadable stubs can be specially important in interacting with different servers as a system moves among different geographical areas.

It is also interesting to contrast the approach of adaptation through smart proxies, which is based on a “greedy” behavior of each application, with policy-driven approaches [Lupu et al., 2003], in which a global system view is enforced. For the latter, there is the need for mechanisms which allow the structure of the application to be externally manipulated. The use of CCM (CORBA Component Model) [omg, 2002] is already being explored in this context, for instance for introducing dynamic support for fault tolerance [Favarim et al., 2003]. We intend to use the LuaCCM tool [Maia et al., 2004], which provides a binding between Lua and CCM, and experiment with the integration of both approaches.

8. Acknowledgments

Both authors have been supported by CNPq (the Brazilian Research Council).

References

(2000). *Workshop on Reflective Middleware*. held in conjunction with Middleware'2000.

(2002). *CORBA Component Model - Version 3.0*. Object Management Group, Needham, EUA. document: formal/2002-06-65.

(2003). *2nd Workshop on Reflective and Adaptive Middleware*. Held in conjunction with Middleware'2003.

Bakken, D., Schantz, R., and Zinky, J. (1995). Overview of quality of service for distributed objects. Utica, New York. BBN Technologies, Fifth IEEE Dual Use Conference.

Batista, T., Cerqueira, R., and Rodriguez, N. (2003). Enabling reflection and reconfiguration in CORBA. In *2nd Workshop on Reflective and Adaptive Middleware*, pages 125–129, Rio de Janeiro, Brazil. held in conjunction with Middleware'2003.

Celes, W., Figueiredo, L., and Ierusalimschy, R. (1996). Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652.

Cerqueira, R., Ierusalimschy, R., and Rodriguez, N. (1998). Using reflexivity to interface with CORBA. In *International Conference on Computer Languages 1998*, pages 39–46, Chicago, IL. IEEE, IEEE.

Chang, F. and Karamcheti, V. (2000). Automatic configuration and run-time adaptation of distributed applications. In *Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 11–20, Pittsburg, Pennsylvania.

Edwards, W. K. (1999). *Core JINI*. Prentice Hall.

Favarim, F., Fraga, J., and Siqueira, F. (2003). Fault-tolerant CORBA components. In *2nd Workshop on Reflective and Adaptive Middleware*, pages 144–148, Rio de Janeiro, Brazil. held in conjunction with Middleware'2003.

Fuggetta, A., Picco, G., and Vigna, G. (1998). Understanding code mobility. In *IEEE Transactions on Software Engineering*, volume 24.

Group, O. M. (1999). The common object request broker: Architecture and specification. OMG document formal/99-10-07. v2.3.1.

Lupu, E., Lymberopoulos, L., and Sloman, M. (2003). An adaptive policy-based framework for network services management. *Journal of Networks and Systems Management*, 11(3):277–303.

Maia, R., Cerqueira, R., and Rodriguez, N. (2004). An infrastructure for development of dynamically adaptable distributed components. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, Proceedings, Part II*, volume 3292 of *Lecture Notes in Computer Science*, pages 1285–1302, Agya Napa, Cyprus.

Moura, A., Ururahy, C., Cerqueira, R., and Rodriguez, N. (2002). Dynamic support for distributed auto-adaptive applications. In *Proceedings of AOPDCS - Workshop on Aspect Oriented Programming for Distributed Computing Systems (held in conjunction with IEEE ICDCS 2002)*, pages 451–456, Vienna, Austria.

Newmarch, J. (2000). *A Programmer's Guide to JINI Technologies*. APress.

Silva, F., Endler, M., and Kon, F. Developing adaptive distributed applications: a framework overview and experimental results.