Exploiting Sequential Libraries on a Cluster of Computers

Miguel Barreiro Jose Luis Freire Victor M. Gulias Juan J. Sanchez
{enano,freire,gulias,juanjo}@lfcia.org
LFCIA, Department of Computer Science
University of A Coruña, SPAIN

Abstract

In this paper, a methodology to exploit legacy sequential libraries on a multicomputer is shown. The process of integrating such libraries is exposed as a slave pool design pattern. The proposed solution is implemented using the concurrent functional programming language Erlang, and it has been applied to extend a C++ sequential XSLT library to perform clustered XSLT processing on a Beowulf, which can be the supporting architecture for a heavy-loaded web server offering dynamic content creation from XML documents.

1 Introduction

There are a lot of situations where efficient and well-tested sequential libraries are available. These legacy libraries are normally written using low-level programming languages (C, C++, Fortran), thus modifying the libraries to exploit a cluster of computers does not seem to be a good approach.

In this paper, a methodology to exploit legacy sequential libraries on a multicomputer is shown. The process of integrating such libraries is exposed as a well-known pattern of cluster computing: a generic master/slave approach. The proposed solution is implemented using the concurrent functional programming language Erlang: the abstraction features of functional languages can be used to implement generic software that can be reused many times, while its concurrent and distributed built-in primitives ease the implementation of the distributed control.

This approach has been applied to extend a C++ sequential XSLT library to perform clustered XSLT processing on a Beowulf, which can be the supporting architecture for a heavy-loaded web server offering dynamic content creation from XML documents, with encouraging results.

The paper is structured as follows. The following section is devoted to the key concepts needed to understand the ideas presented (design patterns, functional programming, distributed Erlang). Section 3 explains the slave pool pattern and how it can be implemented in Erlang. Section 4 shows a direct application of the slave pool pattern to extend a sequential XSLT library to exploit distributed execution on a Beowulf cluster configured as an HTTP server. Finally, we conclude and present further work.

2 Background

2.1 Design Patterns

As long as engineering disciplines mature, handbooks that describe successful solutions to known problems can be written. Such collection of proven solutions constitutes a *pattern language* [4, 10]. A *design pattern* is an attempt to describe successful solutions to common software problems, thus helping people reuse such successful practices, communicating better, and reasoning about what they do and why. A pattern represents designer background about some recurring topic, so there is nothing new about patterns since they just capture experience. A pattern usually includes its intent, a motivating example of a problem in which the pattern can be applied, the abstract structure, participants and their collaboration, and an implementation example, among others.

2.2 Functional Programming

Declarative languages, in particular those whose underlying model of computation is the *function* (*functional languages* [6]), are characterized as having no implicit state and thus the emphasis is placed entirely on programming with *expressions*. Functions are treated as first-class objects, are allowed to be recursive, higher order, and polymorphic. The main property of these languages is *referential transparency*, which guarantees that a function application may be freely substituted for its definition because of the absence of side effects. That allows to reason formally about programs and to develop applications from formal specifications. Moreover, the abstraction of functional languages is used to build programming patterns that can be easily adapted to a cluster of computers. Therefore, functional languages are well suited for elegant symbolic computation.

Even though the advantages of using functional languages are clear, they demand far more resources (time, space) than non-declarative languages and, hence, a great deal of effort has to be done to implement efficient functional evaluation exploting, for example, the parallelism observed in their side-effect-free expressions.

2.3 The Erlang Functional Language

Erlang [1] is a distributed and concurrent functional programming language developed by Ericsson for telecommunication applications. The language has no constructs inducing side effects to an implicit store, as found in other almost-functional languages such as ML, with the exception of communications among threads (*processes*, in Erlang terminology), and evaluates expressions eagerly, as opposed to other functional languages such as HASKELL that use lazy evaluation.

Values in Erlang (i.e., non-reducible expressions) range from numbers and *atoms* (symbolic constants, lower-case in Erlang syntax) to complex data structures (lists, tuples) and functional values, which are treated as first-class citizens. A function is defined by a set of equations, each stating a different set of constraints based primarily on the structure of the arguments (*pattern-matching*). Iterative control flow is carried out by using function recursion. Lists, the most important data structure in functional languages, are written `[a,b,c]` with `[]` being the empty list, and a list whose first element is `X` and whose rest is `Xs` is denoted `[X|Xs]`. Thus, to define a function that tests for membership in a list, we can write:

```
member(X,[])      -> false;
member(X,[X|Xs]) -> true;
member(X,[Y|Ys]) -> member(X,Ys).
```

Functions are grouped into modules, and a subset of those functions can be exported, declaring both function name and arity, to be used by other modules.

```
-module(mymod).
-export([member/2]).
```

Hence, `mymod:member(3, [1,2,3])` reduces (i.e., evaluates) to the atom `true`, while `mymod:member(5, [1,2,3])` reduces to `false`. Besides lists, Erlang programmers also can use *tuples*, similar to C structures, which are constructed in arbitrary but finite length by writing `{A,B,...,C}`. Lists and tuples can hold any valid Erlang value, from numbers and atoms to lists, tuples and even functional values.

As other modern functional languages, interesting features such as higher-order functions (functional parameters or functional results), on-the-fly function creation, dynamic typing (even though a static type checker is also available), list comprehensions, and a powerful collection of libraries which constitutes the Open Telecom Platform (OTP) that includes a distributed database, graphical interfaces, an ASN.1 compiler, a CORBA object request broker, COM and Java interfaces among others. In fact, these libraries make Erlang one of the notable success exceptions in real world applications [11, 12].

2.4 Concurrent Erlang

What makes Erlang different from other functional languages is its support for concurrency and distribution. With Erlang's available primitives for concurrency, it resembles formal calculi such as Milner's CCS [9] or Hoare's CSP [5]. The idea behind the scenes is clear: concurrent programming is far simpler in a functional language than in a traditional imperative one because of the absence of side-effects.

A new thread is created by using the built-in function `spawn`. Given a module M, a function F with arity N exported from module M, and a list of arguments $[A_1, A_2, \ldots, A_N]$, the expression `spawn`$(M, F, [A_1, A_2, \ldots, A_N])$ starts a new process in charge of computing $M : F(A_1, A_2, \ldots, A_n)$. Once evaluated, `spawn` returns the *process identifier*, `Pid`, of the newly created lightweight process.

In order to allow interaction among processes, a couple of asynchronous message passing primitives are available:

- `Pid ! Msg`

  `Msg` is asynchronously sent to the process Pid without blocking the sending process. If process `Pid` exists, the message is stored in its *mailbox*. Any valid Erlang value can be sent to other process, including complex data structures containing lists, tuples, functions, and process identifiers.

- `receive`
  Pat_1 `->` $Expr_1$`;`
  `...`
  Pat_M `->` $Expr_M$
  `end`

  It searches the process mailbox looking for a message that matches one of the patterns $Pat_1, \ldots, Pat_M$ sequentially. If no available message exists, the process blocks until such message arrives.

Using this concurrency constructs, a *server* can be easily defined as a recursive function that receives a request from a client and sends the response back to the client. The *state* of the server is plumbed as a function parameter. For instance, the following code defines a server which adds `Inc` to client request parameter.

```
-module(inc_server).
-export([loop/1]).

loop(Inc) ->
  receive
    {From, X} -> From ! X+Inc,
                 loop(Inc)
  end.
```

Observe that as part of the request, the client includes its own identity to receive the server response. To create the server process with *Inc=1*, we should evaluate `spawn(inc_server, loop, [1])`. For example, the client API can be defined as:

```
client(X) ->
  ServerPid ! {self(), X},
  receive
    Response -> Response
  end.
```

where $ServerPid$ is the identity of the server process, and `self` is a built-in function that returns the identity of the running process.

Higher-order functions can be used to generalize the basic server pattern and to avoid the repetition of the same structure at different places. In this case, the algorihm to compute the response given a request is encapsulated into a function F, that also delivers the state of the server for the next iteration.

```
-module(server).
-export([start/2, loop/2]).

start(F, State) ->
  spawn(server, loop, [F,State]).

loop(F, State) ->
  receive
   {From, Req} ->
     {Resp, NewState} = F(Req, State),
     From ! Response,
     loop(F, NewState)
  end.
```

This skeleton can be specialized then to build the aforementioned `inc_server` by creating an on-the-fly function:

```
server:start(fun (X,Inc) ->
                    {X+Inc,Inc}
             end, 1).
```

2.5 Distributed Erlang

The natural extension of Erlang to a distributed framework is to have more than one node (possibly at different hosts) running Erlang processes. In order to create a process on a remote node, `spawn(`$Node, M, F, Args$`)` should be used, adding $Node$ as an extra parameter with the location for the newly created thread. The nice feature is that all the concurrency constructs are semantically equivalent in this distributed framework (even though communications among remote processes are less efficient, of course).

3 Slave Pool Pattern Rationale

3.1 Intent

Exploit a sequential library on a cluster of computers by distributing multiple requests among nodes running the library and providing a single access point for clients that must not be aware of any clustering issue.

3.2 Motivation

A web server that generates its contents from XML documents by performing XSL transformations. The web server receives many concurrent requests from client browsers and we want to increase response time and overall throughput. There are efficient (and complex) low-level C/C++ libraries for performing such processing, but in order to implement a scalable solution that library should be exploited on a cluster of computers.

The proposed solution, presented in section 4, is a generic master/slave framework, where master process implements the unified API for clients and the slaves are running on the nodes of the cluster. Slaves provide services based on an adaption of the XSLT library.

3.3 Applicability

The solution can be applied when:

- There is an interesting sequential software library which is going to be heavily used by multiple concurrent clients, and
- The requests are independent and the cost of solving each request is significant.

3.4 Structure and Participants

- *master process*
  - provides single access point for clients

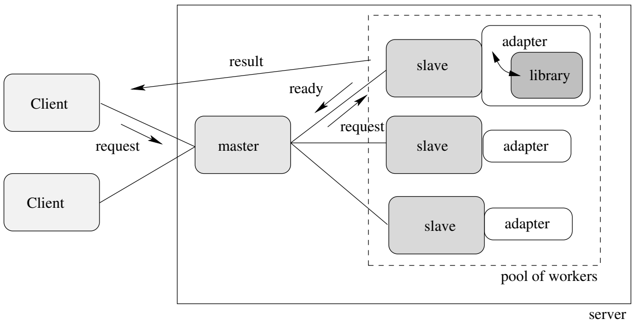

Figure 1. Solution structure and participants

- queues requests from clients
- knows idle clients
- dispatches requests to slaves

- *slave process*
  - adapts legacy library
  - implements slave-side interaction with master
  - solves client requests by interacting with legacy library
- *adapter*
  - Provides an abstract API for slave
  - Only piece of library-dependent software
- *legacy library*
  - implements the needed service
- *client*
  - needs the service provided by the legacy library
  - only interacts with master process API

3.5 Collaborations

1. The client submits a request to the master process through the unified API
2. As soon as the master identifies an idle slave process, the request is dispatched to it
3. The slave, using the adapted library, solves the request and sends the result to the client process, which asks the master for more work.

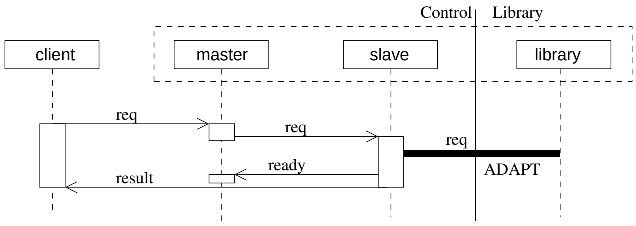

Figure 2. Interaction among actors

3.6 Implementation

Erlang language is going to be used in the implementation of the distributed control (master-slave interaction) and its low-level interface is used to adapt the legacy library. For the sake of simplicity, Erlang fault-tolerance mechanisms and special robustness features have been left out in the paper.

Adapter. The first task to be done is to adapt the legacy library. The goal is to provide a uniform interface to be used by the slave. The interface includes:

- *Initialization*. Function that takes an argument (it can be a complex structure) and initializes the library. It returns a service function and an initial state (for example, a handler for a resource used by the library).

```
Init(Arg) -> {Adapt, State}
```

- *Service function*. It interacts with the legacy library by taking the actual request and the current state and delivering a pair with the response and the new state.

```
Adapt(Req, State) -> {NewState, Resp}
```

There are different approaches in Erlang to implement the adapter; the easiest way is to implement a C port that links low-level libraries running on a separate operating system process to the Erlang virtual machine. Ports provide the basic mechanism for communication with the external world, from Erlang's point of view. They provide a byte-oriented interface to an external program.

This is the only library-dependant code in the pattern implementation.

Slave. Each slave process is a generic server similar to the one shown in section 2.4. The recursive function that receives the requests from the master, interacts with the library adapter using the service function, and delivers the response and the notification for the master. It can be implemented as follows.

```
loop(Master, Adapt, State) ->
  receive
    {From, Req} ->
      {NewState, Resp} = Adapt(Req, State),
      Master ! {self(), ready}
      From   ! {ok, Resp},
      loop(Master, Adapt, NewState)
  end.
```

The slave module is completed with the proper initialization. Besides creating the slave process, the adapter initialization should be performed so that the slave has access to both the adaptation function and to the initial state.

```
-module(slave).
-export([start/4, init/3]).

start(Node, Master, AdaptInit, AdaptArg) ->
  spawn(Node, slave, init,
        [Master, AdaptInit, AdaptArg]).

init(Master, AdaptInit, AdaptArg) ->
  {Adapt, State} = AdaptInit(AdaptArg),
  loop(Master, Adapt, State).

loop(Master, Adapt, State) -> ...
```

Master. The master process should queue temporarily the incoming client requests until a slave is able to handle them. This information, part of the state of the master server, is going to be represented as a list of pairs $\{ClientPID, Request\}$, and new requests should be placed at the end of the list to implement a FIFO policy. In addition, the master should remember the identity of all idle slaves, so a list with the PID of such slaves is also stored as part of master state. Whenever the master gets a request and a idle slave, a new task can be dispatched. This condition can be simply tested by trying to match both lists with lists with at least one element:

```
loop([ASlave|Slaves], [ATask|Tasks]) ->
  dispatch(ASlave, ATask),
  loop(Slaves, Tasks);
...
```

The dispatching of a task implies an interaction with the slave chosen, that is:

```
dispatch(ASlave, Task) ->
  ASlave ! Task.
```

In the event that the previous pattern does not match, the situation in which there is no task or there is no idle slave has been reached. Master process should then wait for incoming messages in order to change this condition. Master process interacts not only with clients, but also with slaves greedy for new work. Hence, the receive loop for the master process must distinguish between slave and client interaction by using pattern-matching, as expected:

```
loop(Slaves, Tasks) ->
 receive
  {From, ready} ->
       loop(Slaves ++ [From], PendingTasks);
  {From, Req} ->
       loop(Slaves, Pending++[{From,Req}])
 end.
```

Slaves can be dynamically added to the pool by sending to the master process a $\{SlavePID, ready\}$ message. Perhaps it is easier for many settings to let the master start some slaves at some nodes; the following function starts an slave in every node in the input list $Nodes$ by using a *list comprehension*, an Erlang notation similar to the used in mathematics for defining sets.

```
create_slaves(Nodes, AdaptIni, AdaptArg) ->
  [ slave:start(Node,
                self(),
                AdaptIni, AdaptArg)
         || Node <- Nodes ]
```

Thus, the initialization of the master process, registering its identity for public usage and spawning a group of slaves, can be defined as:

```
-module(master).
-export([start/3, init/3]).

start(Nodes, AdaptInit, AdaptArg) ->
  spawn(master, init, [Nodes,
                       AdaptInit,
                       AdaptArg]).

init(Nodes, AdaptIni, AdaptArg) ->
  register(master, self()),
  loop(create_slaves(Nodes,
                     AdaptIni,
                     AdaptArg),
       []).
```

Client API. Once a master process has been created, any client can send it a library-dependant request and wait for server response. For example:

```
...
  master ! {self(), {myreq,3}},
  receive
    {ok, Response} ->
      ... process Response ...
  end
...
```

However, this is an error-prone approach because the interaction protocol used is exposed to the client. It is better to create an specific client API encapsulating that:

```
myreq(N) ->
  master ! {self(), {myreq, N}},
  receive
    {ok, Response} ->
      Response
  end.
```

Now, `myreq(3)` is equivalent to the previous block of code but the client can see the interaction as a plain function call.

4 A Case Study: Clustered XSLT Processing

As long as information can be retrieved from different sources, it is advisable to provide a general mechanism to adapt this content to the specific features of such output devices. Contrary to when style information is hard-coded into the content, separation of style from content allows for the same data to be presented in different ways. This enables:

- *reuse of fragments of data*

  same content should look different in different contexts.

- *multiple output formats*

  different media (paper, online), different sizes (manuals, reports), different classes of output devices (workstations, hand-held devices).

- *styles tailored to the reader's preference*

  print size, color, and so on; easier support for blind or visually impaired users.

- *standardized styles*

  corporate stylesheets can be applied to the content at any time.

- *freedom from style issues for content authors*

  technical writers do not need to be concerned with layout issues because the correct style can be applied later.

- *easy interface with third-party software*

  Simple, standard interface for data interchange with other applications.

In order to achieve such goals, an application can produce its results as XML documents which can be transformed at a later stage according to an XSL stylesheet. Erlatron, a subsystem of the monitoring tool MONET [8], can be used to perform such distributed XSLT processing. Erlatron has been implemented in Erlang using a C++ library (SABLOTRON [7]) using the approach presented in this paper.

4.1 Erlatron Overview

Figure 3 presents the actors involved in Erlatron. Erlatron adapts the SABLOTRON library by using a port which performs basic transformations of a couple of binaries representing the XSL stylesheet and the XML source. SABLOTRON is based on the EXPAT library [3] and has been designed for performing fast and compact transformations. The port is managed by a generic server that offers XSLT services to any client. This *slave server* constitutes the basic processing unit and, considering the CPU cost of performing XSL transformations, low additional overhead is expected when using from Erlang.

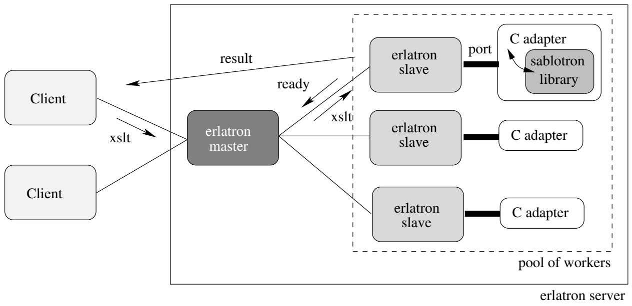

Figure 3. Erlatron actors

In order to exploit a distributed framework, such as the Beowulf cluster presented in [2], a master/slave architecture is deployed. In this setup, a *master server* is used to distribute requests to different slave servers running on a pool of computer nodes. The state of the master server is a collection of pending transformations as well as the information about idle slaves. The dispatching of requests carried out by the master's scheduler consists of pairing a pending transformation with an idle slave server. Figure 4 shows the interaction among actors when solving an XSLT service.

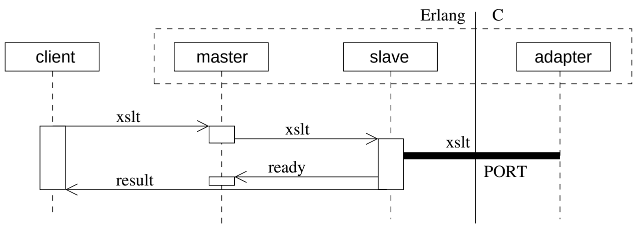

Figure 4. Erlatron interaction

Clients interact with the master server through a global registered name, being possible to take over an erlatron server to a different location without disturbing client interaction.

4.2 Erlatron Benchmark

As the reader can guess, Erlatron architecture seems to be quite interesting for web sites with a high number of requests which involve many XSL independent trasformations for dynamic content generation. We are going to present some results when using Erlatron on a linux Beowulf cluster with a frontend and up to 22 nodes. The frontend (Dual Pentium II 350MHz 96MB) runs `inets`, an Erlang/OTP http server, serving a 64KB HTML generated using a 25KB XML document and a small XSL stylesheet; each node (AMD K6 300-266MHz, 96MB) hosts an Erlatron slave. All the nodes are linked using a switched 100Mb Fast Ethernet.

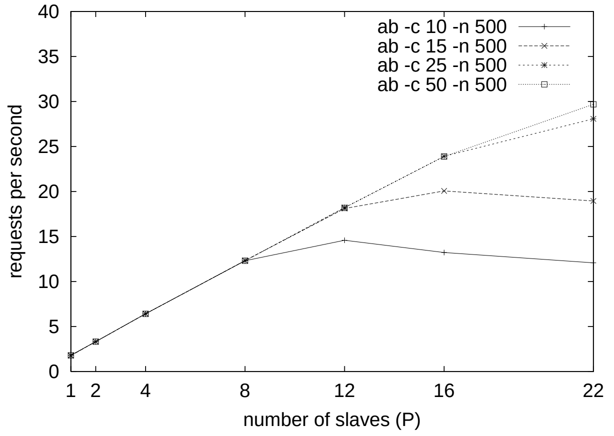

Figure 5. Requests/second varying slaves

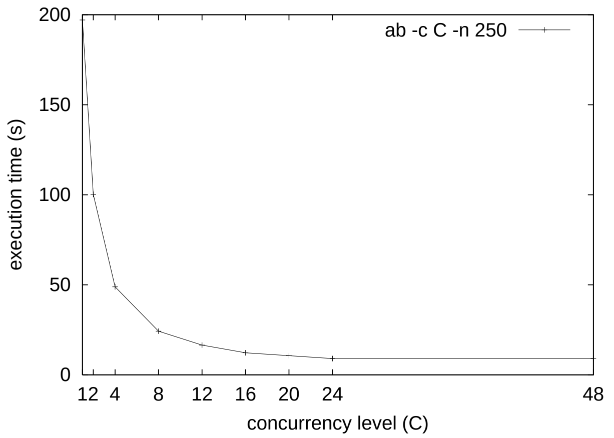

Figure 6. Times when increasing concurrency

Figure 5 shows the requests per second achieved when running apache bench (`ab`), a tool for benchmarking an HTTP server, at the frontend. As new XSLT processors are added to the slave pool, the server is able to increase its service rate until the concurrency level (C, number of simultaneous requests, `-c`) matches the pool size. Figure 6 presents the time taken to attend a collection of requests, varying the concurrency level.

Figure 7 shows how the average time for a request can be kept almost constant while concurrency level is not greater than the number of slaves. It should be pointed out that the overhead for the case $slaves = 22$ and C low is due to the use of nodes 17-22 which are AMD K6 266Mhz, while nodes 1-16 are AMD K6 300Mhz. Master scheduler should take into account this fact and introduce some kind of priority among slaves. This is particularly interesting when some slaves run at the frontend because that will reduce network communications.

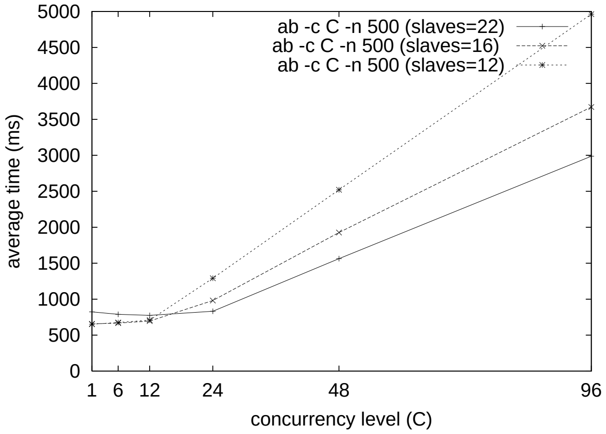

Figure 7. Average service time

4.3 Stylesheet Selection

Given a valid XML document, one of the Erlatron duties must be the selection of an appropriate XSL stylesheet based on the type of the requested document and the nature of the output device, even though other parameters can be taken into account such as system workload (e.g. a simpler formatting can be chosen under heavy load condition). In order to accomplish this requirement, Erlatron subsystem incorporates a module with two basic features:

- *stylesheet registration*

  A Mnesia database is used to store the binding between a document type and an XSL stylesheet. Taken into account that the same document type can have different stylesheets, the database also stores a *rule* that decides whether the style can be applied.

- *stylesheet selection*

  Using the information stored in the XSL repository, a stylesheet, matching its rule in the given context, is chosen.

A *rule* for a given stylesheet is defined as a function that takes a context and an extra argument and returns `true` if the stylesheet can be applied to the document in such context. The *context* is defined as a list of pairs {*Key*, *Value*}, tipically the result of parsing the `QUERY_STRING` or the environment of an HTTP query. Some useful functions are provided to test for an specific {*Key*, *Value*} pair (`equal`) or for matching a regexp (`match`). For example, a rule to trigger a stylesheet when using the *lynx* browser is `lynx`, defined as:

```
lynx(Context, _Extra) ->
  match(Context, {"User-Agent", "Lynx"}).
```

Higher-order functions are used to define complex queries combining simpler rules. For example, the rule `all` takes a list of rules (functions and, optionally, the extra parameters) and tests all the conditions. As an example, the following Erlang term binds the document type `meas` with the XSL `http://my.host/xsl/style.xsl` when using *lynx* and a simple style is required.

```
{meas,
    "simple lynx style for meas",
    "http://my.host/xsl/style.xsl",
    {all, [lynx,
           {equal,{"style", "simple"}}]}}.
```

This additional adaption can be performed at the master process composing the XSLT adaption with the rule selection algorithm.

5 Conclusions

A methodology to exploit legacy sequential libraries on a multicomputer has been shown. The well-known master/slave architecture has been proposed and implemented as a skeleton using the concurrent functional programming language Erlang.

The abstraction of functional languages, in particular, higher-order functions, has contributed to the implementation of generic code that can be applied to different similar problems. The concurrent/distributed facilities of the language simplify the building of the protocol among clients, master, and slave processes, and the Erlang built-in supervision support allows for a simple yet fault-tolerant implementation. As a consequence, functional languages have been found to be especially suitable for robust, verifiable implementations of the successful solutions to common problems described by *design patterns*.

The distributed pattern has been applied to extend a C++ sequential XSLT library to exploit a Beowulf cluster which can be the supporting architecture for a heavy-loaded web server offering dynamic content creation from XML documents. The measurements performed show that nice results can be obtained by wrapping the low-level sequential library and a cluster can be quickly adapted to solve such scalable problems, as long as they meet certain granularity and latency characteristics.

References

[1] J. L. Armstrong, M. C. Williams, C. Wikström, and S. R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition edition, 1995.

[2] M. Barreiro and V. M. Gulias. General issues on cluster administration. In R. Buyya, editor, *High Performance Cluster Computing*, volume I. Prentice Hall, 1999.

[3] J. Clark. *expat - XML Parser Toolkit 1.1*. `http://www.jclark.com/xml/expat.html`, 1999.

[4] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. & 0-13-153289-8.

[6] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, Sept. 1989.

[7] T. Kaiser. *Sablotron 0.44*. Ginger Alliance Ltd. `http://www.gingerall.com/charlie-bin/get/webGA/act/sablot_man.act`, Sept. 2000.

[8] J. S. M. Barreiro, V. Gulias. Monet: A monitoring and instrumentation tool developed in erlang. *Erlang User Conference'00*, Oct. 2000.

[9] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[10] D. C. Schmidt, M. Fayad, and R. E. Johnson. Software patterns. *Communications of the ACM*, 39(10):37–39, Oct. 1996.

[11] P. Wadler. An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, Feb. 1998.

[12] P. Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33(8):23–27, Aug. 1998.