World-wide algorithm animation

Bertrand Ibrahim, Ph.D.
Computer Science Department, University of Geneva
rue du Général Dufour 24, CH-1211 Geneva 4, Switzerland

E-mail: bertrand@cui.unige.ch, fax: (+41 22) 20 29 27, phone: (+41 22) 705 75 08,
www: http://cui_www.unige.ch/eao/www/Bertrand.html

Abstract:

This paper reports on an innovative use of the WWW at the University of Geneva. Indeed, until now, the web has mostly been used to make information publicly available, whether directly in documents, or through search engines querying external databases. With the availability of forms, we are now starting to see WWW viewers used to input information, e.g. to order a pizza or give one's opinion about some topic.

In the Computer Science department, as a semester project for the Software Engineering class, we are now implementing an experiment to allow first year students to have not only on-line access to a hypertextual version of the book used in the Data Structures class, but also to "animate" the algorithms that are described in the book. That is, the students can run, on the server, the program (or program segment) and interact with the execution to put breakpoints in the code, display the contents of variables and advance execution either step by step, or until a breakpoint is met, in much the same way as with a symbolic debugger.

To do this required the development of a whole set of tools to facilitate, and even automate, the preparation of the algorithms to allow them to be started and controlled from a WWW client like Mosaic. It also required designing a mechanism to have the server spawn subprocesses to execute the algorithms and have the server query the appropriate subprocess for what has to be displayed next, based on the users' queries.

This paper will describe the technical solutions that had to be designed to make this remote control feasible. Even though the project described in this paper has educational purposes, many of the solutions described can prove useful in very different contexts.

1. Introduction

When people discover the World-Wide Web for the first time, they probably get the same impression as if they had just opened a door that looks onto the Amazonian forest. The things that must be most striking are its immenseness and its wildness. Crowds of lianas are going in all directions, leading to places which have a completely different outlook from the place where these new travellers were a moment before.

This comparison goes beyond just contrasting lianas and hyperlinks. While the Amazonian forest is victim of man, and is actually diminishing in size, the WWW information forest keeps growing and evolving at an almost frightening speed. But nobody actually sees it grow, except very locally.

This jungle has nevertheless its maps, called *What's New...*[1], *On FAQ...*[2], *About...*[3], *NCSA Starting Points*[4], *CUI W3 Catalog*[5], *World Wide Web FAQs and Guides*[6], *Internet Starting Points*[7], *Internet Resources Meta-Index*[8], and many more. It also has its guides (*JumpStation*[9], *The Clearinghouse for Subject-Oriented Internet Resource Guides*[10], *ALIWEB*[11], etc.) that can bring you to places that are of potential interest to you. These tools are quite useful, but they are not always available[12] and are generally unable to keep up with the rapid growth of the giant.

An interesting aspect of this "creature" is its ability to generate phantom trees, called *indexes*. These phantom trees look like real trees and can bear their own lianas that will transport the traveller to new locations, whether real or virtual. The lianas leading to such imaginary trees are in no way different from other lianas leading to actual trees. These indexes appear as if they were real documents, but they are nowhere to be found, as is, on a physical medium. These documents are created on the spot whenever a link referring to them is activated.

This special form of virtual reality in the world of information is a very powerful concept, as it allows information providers to furnish more than just bare information or facts. They can indeed seamlessly make available a whole world of objects and processes, as long as they can give these processes the appearance of regular documents. Pushing this line of reasoning to its limits, WWW can be considered as a remote display interface that can connect remote users to various applications made available to them all over the world. Some have already taken the plunge, like Paul Burchard[13], from the Geometry Center[14], University of Minnesota, with his W3Kit Object Library for Interactive WWW Applications[15].

The use of the web is thus extending from simply sharing (mostly read-only) information and running search engines, to automatically taking orders[16], and now, to running remote applications/simulations.

Actually, what characterizes the essence of the World-Wide Web is its http protocol[17]. What is around it, that is, server and client applications, can vary widely. Of course, NCSA's Mosaic viewer and httpd server daemon, or CERN's server are the most well known representatives, but

1. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/whats-new.html
2. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/mosaic-faq.html
3. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-about.html
4. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/StartingPoints/NetworkStartingPoints.html
5. http://cui_www.unige.ch/w3catalog
6. http://cui_www.unige.ch/OSG/FAQ/www.html
7. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/StartingPoints/NetworkStartingPoints.html
8. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/MetaIndex.html
9. http://www.stir.ac.uk/jsbin/js
10. http://http2.sils.umich.edu/~lou/chhome.html
11. http://web.nexor.co.uk/aliweb/doc/aliweb.html
12. http://cui_www.unige.ch/eao/www/WWW94/http:_www.ncsa.uiuc.edu_SDG_Software_Mosaic_Docs_whats-new.html
13. http://www.geom.umn.edu/people/burchard.html
14. http://www.geom.umn.edu/
15. http://www.geom.umn.edu/docs/W3Kit/W3Kit.html
16. http://rs560.cl.msu.edu/weather/getmegif.html
17. http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html

a WWW server does not need to access documents to produce information, and a client does not need to display on a computer screen the information it gets. The cohesion factor between these very dissimilar components is the "language" they talk, i.e. the http protocol.

2. Global description of the project

The project[1] we are describing here has many similarities with the W3Kit Object Library[2], in that it focuses on using WWW as a highly portable remote display interface. The first goal of the project is to make the book used in our department's Data Structure class available on-line in hypertextual form[3], to make it easier for the students to browse through it. The second goal of the project is to make it possible for the students to activate the execution of the algorithms described in the book, and to control the execution of these algorithms through some sort of control panel, as with a symbolic debugger.

The specification and the implementation of the system necessary for the project has been given as an assignment to the Software Engineering class, as a semester project. What is presented here is thus the result of a collaborative effort and shouldn't be considered the work of just the author of this document.

WWW was chosen as a support system for this project as it allows for remote access and provides a highly portable display interface abstraction. This includes the Mosaic viewer, as the system we have implemented makes heavy use of **forms**[4] as they are implemented since version 2.0 of Mosaic for X[5]. This approach should make it easy for a student to use this on-line facility from any personal computer or workstation on campus, or even anywhere else in the Internet world.

Since the book was available in electronic form, putting it in hypertextual form was not much of a problem and will therefore not be discussed in this paper. The main problem that had to be solved was to monitor the execution of Pascal programs through WWW, taking into account the fact that the http protocol is stateless, that is, the server does not remember former queries when it processes a new one[6].

---

1. http://cui_www.unige.ch/eao/www/ProjetInfo.html
2. http://www.geom.umn.edu/docs/W3Kit/W3Kit.html
3. http://cui_www.unige.ch/eao/www/std/table.html
4. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html
5. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-on-version-2.0.html
6. The solution of running the programs on the client machines with a regular symbolic debugger was not considered, as it would be highly non-portable. Indeed, it would imply using a specialized MIME type, with a dedicated viewer that had to be loaded on the client machine to make the remote execution possible. Only those people having this special MIME type defined and a version of the dedicated viewer that could run on their machine, would be able to run the program animation.
   In addition, since there is no way for the http server to know the architecture of the client machine, we would have had to add an additional document, when an example program is first activated, to allow the user to specify the kind of machine the program would be executing on.
   It was therefore decided that the Pascal programs would run on the server side, be it on the same machine as the http daemon or on another machine, through a remote procedure call

This problem can be further decomposed into:

1 How to run a WWW-activated program step by step.
2 How to display the intermediate states of the program, including its data structures.
3 How to manage user interactions with the model to view or modify the content of variables or set breakpoints in the code.
4 How to minimize client-server traffic.

A few solutions were considered to allow the execution of Pascal programs from within an HTML[1] document.

- First, we considered using a Pascal interpreter and modifying it to allow external monitoring of its states. This solution was abandoned, as we had available only a Pascal-S interpreter, that didn't include file I/O nor dynamic allocation (no pointer types). In other circumstances, using an interpreter could however be a viable solution. Actually, many http servers are already written in some interpreted language. For instance, the Plexus server[2] is just a Perl[3] script.

- We then considered rewriting every Pascal program as a finite state automaton that could start execution at any state and stop execution either at the next state, or at any state for which there would be a breakpoint. The program state and data would be transmitted to the WWW client, both as text to be displayed and, in a more compact form, as a URL hidden in a hyperlink associated with the *Step* and *Continue* push-buttons (see pseudo-code[4]).

  As an alternative to transmitting the whole program data from the server to the client, than back to the server, this information could be stored on the server side, in a temporary file, and have the URL hold a reference to this file.

  This solution would however not be easy to implement for recursive algorithms, as it would imply the explicit management of a stack to store local variables and parameters.

- The solution we finally chose consists in having the shell script, invoked by the http server, spawn a child process to run a slightly modified version of the Pascal program. The modifica-

1. http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html#4
2. http://www.bsdi.com/server/doc/plexus.html
3. http://www.cs.cmu.edu:8001/afs/cs/user/rgs/mosaic/perl.html
4. (the Pascal program has just been started by the server script and is passed the client request)

```
- extract current state, list of breakpoints and value of program variables from request sent by
  client viewer
- loop
     case CurrentState of
          State1: execute first instruction in main program; CurrentState=State2;
          ...
          StateN: execute n-th instruction in main program; CurrentState=...;
          end case
     if SingleStep or BreakPoint[CurrentState] then exit loop
     end loop
- embed current state and value of program variables in a URL
- output the state of the program in HTML syntax, with Step and Continue buttons containing the URL
  just calculated.
- terminate.
```

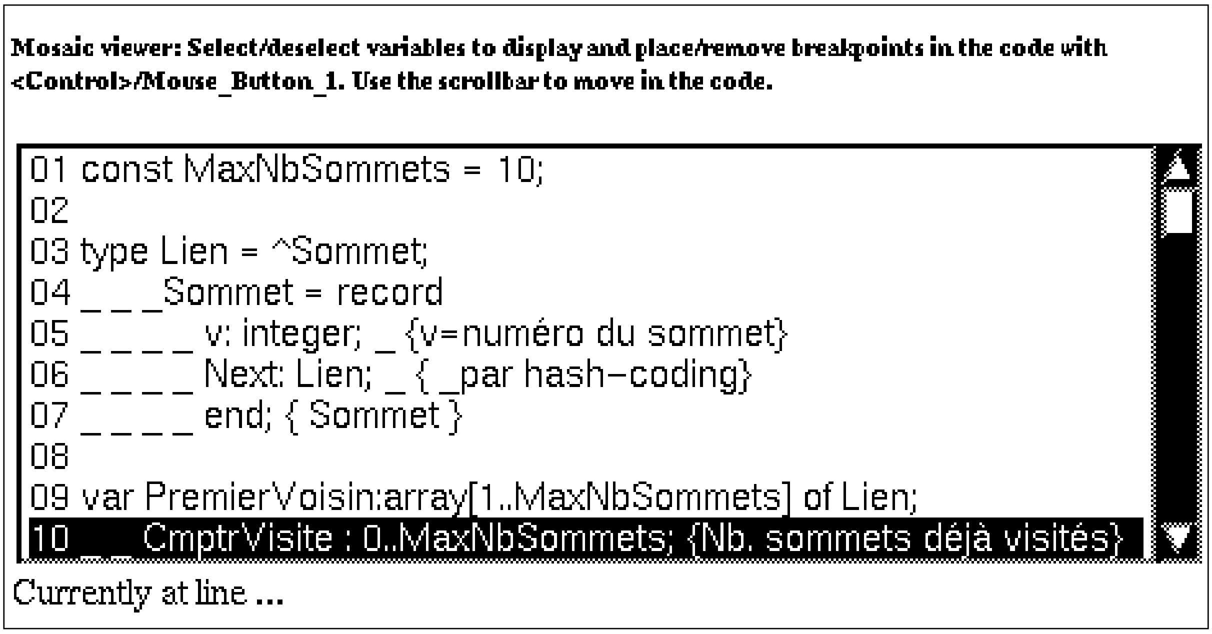
tion consists in adding, between each original Pascal instruction, a call to a synchronization procedure. The synchronization procedure checks whether the execution of the Pascal program should continue, or be interrupted.

Whenever there is a breakpoint on the current instruction, or when the user has selected the single step button, the current state is sent to the client viewer, and the synchronization procedure waits for a signal from the server-activated shell script to resume execution, with possibly new breakpoints and selected variables, based on the latest user request.

3. Screen design and user interface

Before we delve into the details of how a Pascal program can be modified to make it controllable from a WWW client, let us look into what kind of control panel we can present the user with. All the components of the panel have to conform to the HTML[1] syntax and be built with elements that can be displayed by a regular viewer that can handle forms[2]. Now, even text-oriented viewers can handle forms reasonably well, e.g. the Lynx viewer[3]. One can therefore base a WWW tool on the use of forms without reducing by much the generality of the tool.

One of the things we want the user to be able to do is to select, from the source code of an example program, the variables the contents of which should be displayed, and the instructions on which breakpoints should be set. This is best done with a *multiple select* widget like:



Using a *multiple select* widget, where only part of the code is showing, but all of it is available on the client side, has two main advantages:

1. http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html#4
2. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html
3. http://www.cc.ukans.edu/about_lynx/lynx2_3.html

- For long programs, the user can see both the code and other components of the panel, for instance, the display of the contents of variables and maybe even the drawing of a complex data structure.

- The user can look at the rest of the code, and even put breakpoints in it, without having to query the server for it. All the code is at hand, without taking all the screen space, thus saving user's time and network bandwidth.

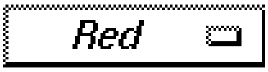
Note: The Mosaic viewer does not take into account leading spaces for the contents of the multiple selection widget, even when the spaces are surrounded with <code> </code>. We therefore had to replace every two spaces with underscores to keep the indentation of the code as much as possible.

3.1. Visual representation of simple data types

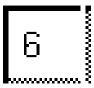
Since we want to allow the user to redefine the value of any data object that is displayed, we have to use input type widgets. These widgets allow the user to see the values and change them without having to send a query to the server for each of them. (*Note:* we are still experimenting with these representations, details may therefore change between what is shown here and the final version)

- **`boolean`** and enumerated types are represented as *single select* widgets of size 1, like:

Done 

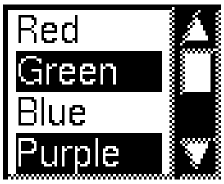
Color 

- **`integer`**, **`character`** and **`string`** types are represented as *text input* widgets, like:

CmptrVisite 

Name 

- **`set`** types of reasonably small size are represented as *multiple select* widgets, like:

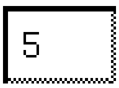
Palette 

- Pointer types are generally not represented as such. It is the pointed object that is displayed, preceded by the name of the pointer object and ^, unless the pointer is nil. Example:

Current: `nil` IntPtr^: 

3.2. Visual representation of data structures

- **`record`** types are represented as a sequence of their fields between square brackets, and are preceded by the name of the record object, e.g.:

Courant^:[.v: 

, .Next^:[.v: 

, .Next: nil]]

- **`array`** types can potentially have numerous elements, which are not all necessarily of interest to the user. Display of **`array`** types is therefore done in two steps. First, when the user selects an **`array`** object, the indices of the **`array`** are displayed as a *multiple select* widget, with no preselected item:

  NumDOrdre[1 2] (click on the up and down arrows of the scrollbar)

  The user can then select the indices for which the value should be displayed, and gets:

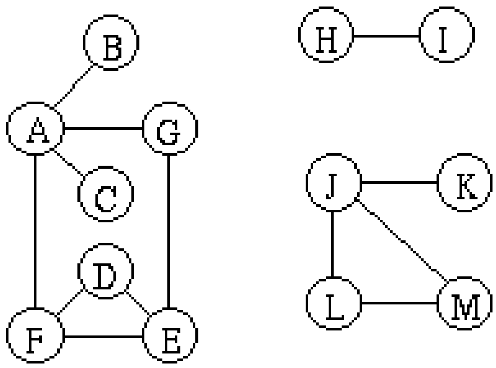
  NumDOrdre[1 2] NumDOrdre[1]: 3 , NumDOrdre[2]: 5 ,

  NumDOrdre[3]: 2 , NumDOrdre[4]: 1 , NumDOrdre[5]: 6 ,

  NumDOrdre[6]: 4

<u>3.3. Graphical representation of complex data structures</u>

For complex data structures such as lists, trees or graphs, a bitmap image can be prepared, like:



to allow the user to point directly to an element of the structure that should be displayed.

A typical screen is shown in Fig. 1.

<u>4. Preparing a program for WWW</u>

Now that we know what we want to achieve, let us see how it can be done. As we said earlier, the solution we chose was to insert, between each Pascal instruction, a call to a procedure (named `Step`) to check for breakpoints or single step mode.

Since the contents of the various variables of the program can potentially have to be sent to the user, this procedure `Step` must have access to both their names and values, and must know about their types, including for local variables within procedures. Therefore, a symbol table must be maintained, that will hold a shadow copy of all variables and parameters used in the program.

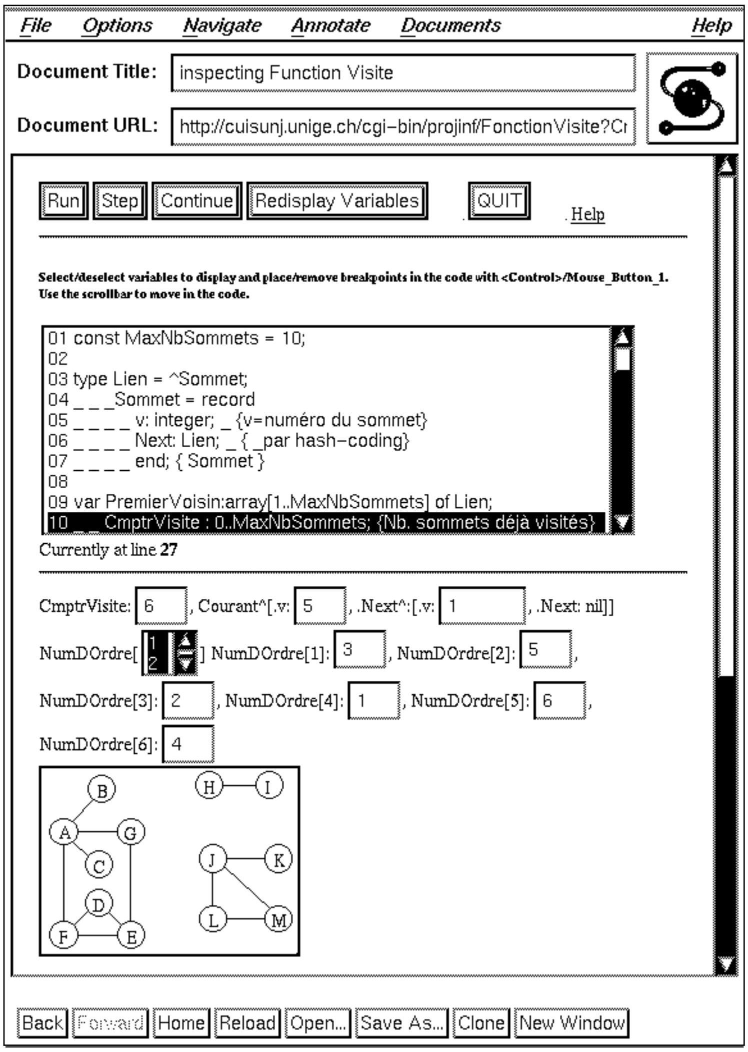


Fig. 1: typical screen

For instance, given the following sample program (line numbers have been added to make it easier to locate the original code in the modified version):

```
01 program Demo;
02 const NbDays = 7;
03 type WeekDays=(Su,Mo,Tu,We,Th,Fr,Sa);
04      Element=record
05                Name: string;
06                Next: ^Element;
07                end;
08 var Num      : real;
09     S        : string;
10     Day      : WeekDays;
11     WorkDays : set of WeekDays;
12     Matrix   : array['a'..'z',1..10] of integer;
13     Chain    : ^Element;
14 begin
```

```
15   Num := 3.1415;
16   S := 'Hi there';
17   Day := Tu;
18   WorkDays := [Mo..Fr];
19   Matrix['d',3] := 67;
20   new(Chain);
21   Chain.Name := 'Brown';
22   Chain.Next := nil;
23 end
```

A declaration will be added after line 11:

```
11     WorkDays : set of WeekDays;
       www_WorkDays: WeekDays;
```

to allow the insertion of a loop to scan the values in `WorkDays` whenever this variable is changed.

Similarly, two declarations will be added after line 12:

```
12     Matrix   : array['a'..'z',1..10] of integer;
       www_Matrix_1: 'a'..'z';
       www_Matrix_2: 1..10;
```

to allow the insertion of two embedded loops to scan the values in `Matrix` whenever this variable is changed.

Then, the executable part of the program will look like:

```
14 begin
     – code added here to create symbol table entries
       for objects declared in this block
     Step;
15   Num := 3.1415;
     – code added here to (re)define value of 'Num' in symbol table
     Step;
16   S := 'Hi there';
     – code added here to (re)define value of 'S' in symbol table
     Step;
17   Day := Tu;
     – code added here to (re)define value of 'Day' in symbol table
     Step;
18   WorkDays := [Mo..Fr];
     – code added here to (re)define value of 'WorkDays' in symbol
       table (this will be done by checking whether each possible
       value in WeekDays is present in WorkDays or not)
     Step;
19   Matrix['d',3] := 67;
     – code added here to (re)define value of index "3" of
       substructure "'d'" in structure Matrix.
     Step;
20   new(Chain);
```

```
- code added here to indicate that 'Chain' points to a new structure
- code added here to create a new entry for 'Chain^' in symbol table
- code added here to create a new entry for each component of
  the 'Chain^' structure.
Step;
Chain^.Name := 'Brown';
- code added here to (re)define value of '.Name' component of
  dynamic structure 'Chain^'
Step;
Chain^.Next := nil;
- code added here to (re)define value of '.Next' component of
  dynamic structure 'Chain^'
Step;

- before exiting the block, remove all local objects
  from symbol table
end;
```

For the technical reader, full detail[1] of the final code produced for the example above is available, as well as a list of the procedures used[2] in the final code.

All this can seem quite complicated, but it can easily be automated. A tool is currently being developed to parse Pascal code and to add all these calls automatically to the original code. Of course, all these lines that are added to the original code will not be displayed to the user.

For applications written in some interpreted language, for which the interpreter gives the programmer direct access to the interpreter's symbol table, the modifications of the original code would be much simpler than in the scheme we just described, since only calls to the `Step` procedure need to be added to the code.

5. Synchronization between the server script and the spawned processes

Once an application has been modified to include calls to the `Step` procedure, it can potentially be used via WWW. The application process will usually be launched from a script invoked by the http server, in response to a specially formatted query.

Most http server software now support a standard mechanism (the Common Gateway Interface[3]) to run external applications. The URL used with such a mechanism can include information that will be passed to the external application, along with query information that a form-based document would generate. The output of the external application is then used to create the document that is sent back to the client viewer.

In our case, the server will first invoke a shell script that will analyze the query sent by the client viewer and either spawn a new process whenever a new "simulation" is started, or pass the query to the spawned process when the user is stepping through an example program that has already been started.

1. http://cui_www.unige.ch/eao/www/WWW94/CompleteCode.html
2. http://cui_www.unige.ch/eao/www/WWW94/ProcList.html
3. http://hoohoo.ncsa.uiuc.edu/cgi/

A portion of the URL used to refer to an example program will thus contain the process Id of the application, another portion will contain the name of the application, and a third portion will contain the name of the machine on which the process is running. When an application has to be started, the process Id is nul. Once it has been started, the HTML output of the application spawned process will include its own process Id in the URL of the ACTION tag of the form structure representing the "control panel". This process Id, along with the machine name, can be used to pipe information between the server script and the appropriate spawned process, thus allowing the server to run many such spawned processes at the same time, possibly on different machines.

One problem that needs to be dealt with is then the elimination of spawned processes that are not needed anymore, even though the program has not reached its end, because the user has moved to other WWW documents or quitted the viewer. The solution we have adopted is to use a timer signal that will wake up a spawned process if the latter doesn't get reactivated by a user query within a certain time limit. If such a timeout occurs, the spawned process will terminate, thus freeing the system resources it was using.

It should be noted that, with the scheme we used in our project, it doesn't make sense to save the URL of an intermediate state of such remotely executable applications, since the process Id is included in the URL, and this Id is not meaningful anymore once the application has finished executing. One can therefore ask oneself whether it is desirable to use such a scheme. In our case, where all the algorithms were already written, the scheme we chose had the advantage of simplicity, since all the algorithms could be adapted with minimal effort.

For applications that have to be developped from scratch, an alternative would be to have the remote applications written as finite state machines and have the URL hold all the information necessary to execute the next step of the automaton. This would make the URL really universal, at the expense of more complex client-server queries.

6. Conclusion

We have presented, in this paper, a technique that allows information providers to actually make available to the worldwide community remotely accessible interactive applications in a highly portable way. In this context, the http protocol and the HTML viewers are used as an abstract input/output device, and the http server, as a remote application server.

The load on a system that runs an http server with spawned application processes should remain reasonable, unless hundreds or thousands of users try to run applications on the same server at the same time. If such a success were to happen, it would be quite easy to limit the number of applications that the server would accept to run at the same time.

If this kind of use of the web increases in scale, it might prove useful to revise the http protocol and extend it to better support such use. Among the extensions that could be considered, there is the incremental modification of displayed documents to reduce network load, and the support for multi-part documents, that is, documents that contain various parts that require different viewers. For instance a document could consist of an HTML part, accompanied by a sound file that could be played at the same time the user is reading the HTML part.

It might also prove useful to extend the mark-up language to include some simple clickable image support **on the client side**. For instance, the inline image tag (<img ...>) could include a list of *rectangle-href* pairs that would allow the viewer to directly send a request for the document mentioned in the *href* part whenever the user clicks in the area of the image defined by the *rectangle* part. This would make it much easier for information providers to include clickable images in their documents without having to embed its handling in the server configuration.

As for security issues, the scheme that has been described in this paper should not create any security breach on either the client or the server side, since the client only sees HTML documents and the server only executes applications that are local to the server and can thus be trusted.

The last point that we want to mention is that pedagogical aspects of such use of the web have not been discussed here since we have not had the opportunity yet to try the system with students. This kind of computer aided learning does not pretend to be as effective as mastery learning programs that run on standalone machines since delivering educational material through the web reduces somewhat the capabilities that can be used. Even in the face of its limitations, the World Wide Web provides a very interesting alternative when it comes to distance learning, and our project is there to prove it.

7. Acknowledgements

I would like to thank all the students who have participated in the design and implementation of the project described in this paper. Their contribution was essential in the success of the project.

The contributers are: Nathalie ABBOUD[1], Frédéric BOURGE[2], Claude CHRISTOPHI[3], Frédéric DEGUILLAUME[4], L'hadi DEHLAL[5], Raki JILALI[6], Mohamed KANE[7], Attila KRUEZSELY[8], Vincent LAEDERACH[9], Angèle MORABITO[10], Peter PASCHE[11], Patrick ROTH[12], David ROUSSIN[13], Alex VILLAZON[14], Marc VUILLEUMIER[15], Wei WANG[16]

1. http://cui_www.unige.ch/eao/projinf/abboud/me.html
2. http://cui_www.unige.ch/eao/projinf/bourge/me.html
3. http://cui_www.unige.ch/eao/projinf/christcl/me.html
4. http://cui_www.unige.ch/eao/projinf/deguilla/me.html
5. http://cui_www.unige.ch/eao/projinf/dehlal/me.html
6. http://cui_www.unige.ch/eao/projinf/jilali/me.html
7. http://cui_www.unige.ch/eao/projinf/kane/me.html
8. http://cui_www.unige.ch/eao/projinf/attila/me.html
9. http://cui_www.unige.ch/eao/projinf/laederac/me.html
10. http://cui_www.unige.ch/eao/projinf/morabito/me.html
11. http://cui_www.unige.ch/eao/projinf/pasche/me.html
12. http://cui_www.unige.ch/eao/projinf/roth/me.html
13. http://cui_www.unige.ch/eao/projinf/roussin/me.html
14. http://cui_www.unige.ch/eao/projinf/villazon/me.html
15. http://cui_www.unige.ch/eao/projinf/mvuilleu/me.html
16. http://cui_www.unige.ch/eao/projinf/wang/me.html