Navegante – An Intrusive Browsing Framework

Nuno Carvalho, José João Almeida, and Alberto Manuel Simões

Departamento de Informática
Universidade do Minho
smash@cpan.org,{jj|ambs}@di.uminho.pt

Abstract. Navegante is a generic framework to build superior order proxies for intrusive browsing. This framework provides the means for developing tools that behave as proxies, but perform some processing task on the content that is being browsed. Parallel to this content processing, applications can also run other user-defined functions with different purposes and interfaces, but we'll explain those later. Currently, Navegante only builds applications that run as CGIs, but this is intended to change in a near future. Applications are built writing programs in Navegante's Domain Specific Language (DSL).
Navegante is a work in progress. This article aims to describe the current state of development. What applications can be built and how. Also, we identify some implementation problems, and briefly discuss some future improvements. Finally, we try to illustrate most of the concepts described using a couple of case studies.

1 Introduction

A proxy[6] is, typically, a service that handles requests for a set of clients that are using other services. Proxies are traditionally used for caching or authorization purposes. On most of these cases the content served is kept unchanged. On some cases the response can be blocked or redirected. Some set of tools can be designed as proxies, but that in some way change the content in the response message. By doing this they become intrusive on the content, but they still apply the proxy concept – they indirectly serve information provided by other services. On-line spell checking tools, language detection, accessibility improvers[5] are some first glance examples of intrusive proxies. One way of seeing these tools is as a generic processor (or a higher order function [4]) that given a processing function, analysis the originally served content. So, this processor works as a proxy but it also applies this processing function to the content before delivering it to the final client.

Navegante [1] is a framework that can be used to build such applications. Applications that work as proxies for the Internet but that perform some additional kind of task on the browsed content. These applications are capable of giving feedback to the user about any processed data at any given time and in different ways. Applications are also capable of keeping state information over sequential requests and, have the ability to keep state information associated with browsed content.

This framework consists in a set of tools that are developed in Perl, taking advantage of well known Perl modules available on the Comprehensive Perl Archive Network (CPAN). For content processing we chose to use XML::DT [2], because it can be used to process XML or HTML exactly in the same way. Which means that we can, without much effort, and without changing our defined functions migrate our applications to feed on XML source files as well as HTML. We used the *Parser::YAPP*[3] Module to build the language parsers.

In the following sections, we expect to objectively cover the necessary topics to give the reader a basic understanding of the framework, and it's various components. We also illustrate the process of building applications and highlight the advantages of using the framework instead of building them from scratch.

2 The Proxy Pattern

In a client-server paradigm[8] the typical message flow can be described in three simple steps:

Step 1: The client sends a message to the server specifying the request.
Step 2: The server processes the request and sends a response message to the client.
Step 3: The client receives the response message from the server.

The diagram in Fig. 1 helps to illustrate this simple communication flow. There are no middle-mans in this architecture, and the message is delivered to the client exactly as it was sent by the server, content wise at least.

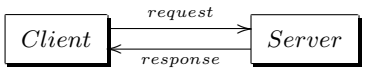

Fig. 1. Simple client-server paradigm.

For our intrusive proxies we add a new layer to the previous architecture. We add a new step in communications on both incoming and outgoing flows so we can perform some task after the server sends the response but before the client receives it. Ideally, we would like to add this extra step in the flow from the server to the client only, but this is not possible since there is no way we can tell the server that that specific response should go into our application before being forwarded to the client, without heavy changes on the client side. The goal here is to be intrusive but we don't want to tamper with every application, we want to keep things working the way they are. So, we upgrade our earlier description to a five step communication architecture:

Step 1: The client sends a message to the proxy specifying the request.

Step 2: The proxy forwards the message to the server unchanged.
Step 3: The server processes the request and sends a response message to the proxy.
Step 4: The proxy catches the response message, processes the content, and forwards the processed content to the client.
Step 5: The client receives the response message from the proxy.

This way we can expect a response from the server, in a given context, and perform some defined transformation on the content before delivering the processed message to the client. The client can be aware of this transformation, or not. The diagram in Fig. 2 helps to illustrate this new approach.

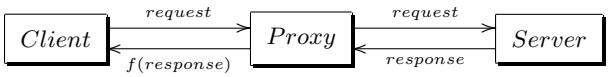

Fig. 2. Simple client-server paradigm with a middle-man.

The current implementation of Navegante only allows the generation of applications as CGIs[7]. So, our final application will be a form where we start by making the initial request (supplying an URL to Navegante's application) and then browse the web as we normally would, but being aware that every request is being processed by our intrusive proxy. At any time, we can tell Navegante to call a previously defined function to display some kind of data compilation or result (to show a state that is maintained while browsing the web).

3 Navegante's Approach

To build an application using Navegante's framework we need to feed the parser a Navegante program file. This file is to be parsed by Navegante, that will use a Perl skeleton file to create an application. This flow is illustrated in Fig. 3.

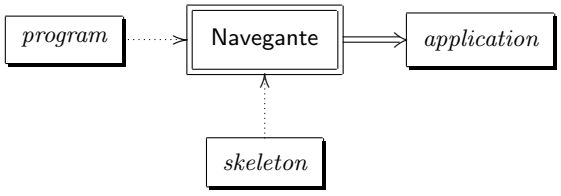

Fig. 3. Navegante's application building flow.

Currently, the only skeleton defined builds a CGI. Therefore, we can only build applications that behave as CGIs. Meaning that they work as standard

HTML pages, and are used as an entry point for intrusive browsing. Once you start browsing using the CGI, the application is already running. Every time a request is made, some processing function defined previously processes the content before delivering it to the client.

Another task being executed, and maybe a not so obvious one, is that every time content is processed, all links are analyzed, so that links to other pages are rewritten. Thus, when these links are followed, we are actually following them using our application, so its resulting content will also be processed. Using this method it is possible to have intrusive browsing without having to change anything in the user or client behavior, since all browsing is done exactly as it would be normally done. There are still some issues with the address rewriting engine, but we will discuss them later.

The application also adds a banner on top of every processed page. This banner provides some interesting features and is illustrated in Fig. 4.

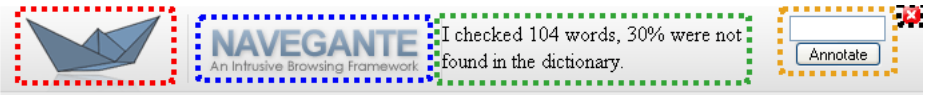

Fig. 4. Sample application banner.

The red outline box is a link to the application's *feedback* function, the function behavior is defined by the application. This is an optional feature, if the application doesn't define a *feedback* function this image it's simply not a link. The blue outline box is just Navegante's logo with a link to Navegante's homepage. The green outline box is the direct result of the function defined as *liveFeedback* that can exist in the application. This function can be used to deliver any kind of data regarding the processed content, and is also optional. The orange outline box is a form, defined in the application, that can be used to annotate pages. This simply stores in the application's state a correspondence between the URL and the user data inserted using the form. Finally, in the black outline box, is illustrated a quit button. This is a last call made to the application that executes a final function when defined by the application. This can be used to provide data summaries of present any other output to the user related to a browsing session.

Another feature that should be pointed out here, is that data can be preserved between requests (data can be made persistent). This means that we can build applications that use data from previous requests and can forward that data along to the following requests. Giving applications a sense of state even that it does not normally exists in protocols like HTTP. This persistent data abstraction is given using *cookies*, but other storage methods can be implemented.

As stated before, because we are using XML::DT, we can take advantage of this module features, and rework our application to process XML structured files.

Without having to change anything on our defined functions because basically, we are using the same processing engine.

3.1 Navegante Programs

Navegante programs are written in plain text files and have a well defined structure. The file should be divided in two major sections, as shown in Fig. 5. These sections are divided by the ## symbols, which has a special meaning for the parser.

```
1 {DSL definitions}
2 ##
3 {generic definitions}
```

Fig. 5. Navegante program structure.

These distinct sections should be used in different ways and with different purposes:

- the first section is used to specify application parameters using a well defined DSL, detailed in the next section.
- the second section is used to define any kind of functions needed by our application in Perl syntax. Everything in this section is almost directly inserted in our final application.

Ideally, it should be possible to write the second section of a Navegante program in any language that can be processed in the environment on which the application is running. Currently Navegante is not *other languages aware*, so this section of the program needs to be written in Perl.

3.2 Navegante's Domain Specific Language

As stated before, Navegante programs are written using a DSL. Most applications parameters are defined using this language. Although in a developing stage, it already can be used to describe many applications behavior. Be aware that most of these statements are CGI specific, because we are mainly building CGIs, but can be used on other contexts as well.

Here is a set of example statements that can be used:

proc: the name of the processing function. It defines the function that is to be called over the content that is being processed;

feedback: the name of the function that is called when feedback is requested, feedback is requested by following the link illustrated in the previous section;

livefeedback: the name of the function that is called every time the banner is generated, as illustrated in Fig. 4;

cginame: the final application CGI name;
formtitle: the title of the newly generated form;
desc: the name of the function that prints a description about the application;
init: the name of the function that is first called when the application starts (mainly used to initialize the application state).
annotate: the name of the function that processes the data submitted using the banner form, and if needed changes the application state;
iform: describe the fields that are to be rendered in the banner form;
quit: defines the function that is going to be used when the application's quit button is clicked.

This directives are used to define the behavior of our application in such a way that Navegante's parser can understand.

3.3 Navegante's Parser

Navegante's parser is currently the main tool in the framework. This tool is responsible for building an intrusive application given a program as described earlier.

The parser is written using *Parse::YAPP*. The parser uses a skeleton definition that can be seen as an application template. That is, the skeleton is defined in such a way that the parser's main job is to contextualize the skeleton using the program's DSL section. The grammar used is actually very simple and is described in Fig. 6.

```
1  Start -> actions fop
2  fop -> '##'
3  actions -> actions 'init' arg
4           | actions 'desc' arg
5           | actions 'proc' arg
6           ...
7           |
```

Fig. 6. DSL grammar.

We have a starting rule that derives in a list of actions and in a terminating symbol. The list of actions is our language specific statements, some of which were described earlier in this section. These statements will precisely describe how the skeleton is converted to a new application. The terminating symbol is simply the **##**, which splits the main sections of our program file.

The second section of our program is directly injected in the application, in order to have access to all the needed functions and variables that will be used during content analysis and/or transformation. Because of this, this section needs to be written in Perl. An idea of improvement would be to allow the use of other languages in this section, because this code is out of the framework scope.

Although it still needs to interact with the framework itself, to access state data for instance.

Currently, included in the parser is also the application skeleton. The application skeleton is nothing more than the final application template and a set of auxiliary functions used by it. The parser uses this template to build the final application.

4 Case Studies

This section describes how to build a couple of very simple applications, using Navegante, for illustration purposes. Keep in mind that the code described in this section most of the times is simplified, and/or not complete, and may actually, by itself, not be enough to implement the application's described behavior.

4.1 *reverse*

The first application we are going to build does not make use of state or feedback related concepts. It simply reverses the content being browsed. To create this application using Navegante, we first need to write a program file describing our application behavior. This example application can be written as show in Fig. 7.

```
1  cginame(./reverse)
2  formtitle(Read Everything Backwards)
3  proc(reverseFunction)
4  desc(reverseDesc)
5  init(reverseInit)
6  ##
7  sub reverseDesc {
8      return "This reverses every sentence in content.";
9  }
10 sub reverseFunction {
11     my $item = shift;
12     reverse($item);
13 }
```

Fig. 7. *reverse* program.

We can clearly see the symbol **##** in line six splitting the file in our two major sections. From line one to line five we are setting a wide range of values for our application using the DSL. We can also see that most of these values are actually callback functions that are going to be later defined in the program. Those are the same exact functions what we are defining from line seven to line thirteen. Our goal is to reverse the content being browsed, so let's take a closer look at line three, where we are using the DSL specific statement *proc* to define the function that will be used to process content, this function name is *reverseFuncion*. We

are defining this function behavior later in our program in line ten. Our function is receiving the content being browsed and reversing it.

After writing the program we can build our application. For this task we only need to run Navegante as shown in Fig. 8.

```
1  $ navegante examples/reverse
```

Fig. 8. Run Navegante.

After running this step a new file called *reverse* is created, as defined in Fig. 7, line one. This new file behaves as a traditional CGI, so we will copy it to a web server, and call it using a browser. The resulting page is illustrated in Fig. 9. Also, we can positively establish that our programs options were used, namely the CGI name, and the description function output.

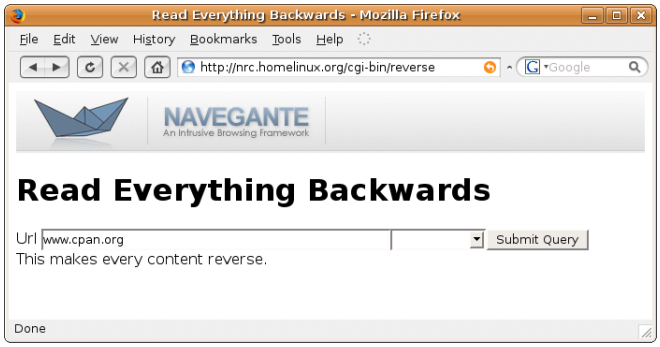

Fig. 9. *reverse* application form.

We can now use this form to start browsing pages, having the content being processed by our application, to be more specific, by the *reverseFunction* in our example.

We can see in Fig. 10 an example of a CPAN visit, where everything that is content was *reversed* by our intrusive application. Note that we are only analyzing content itself, and so the HTML code is kept unchanged, thus pages are correctly rendered by the browser. This is of course the expected behavior, we would not like to have all our HTML or CSS code reversed.

Also note, on both Fig. 9 and Fig. 10, the banner being inserted on the top of the page. Since we didn't define any of the *feedbackLive*, *iform* or *quit* functions for our application, nothing much interesting there.

Fig. 10. *reverse* application being used.

Plus, remember that our processor is rewriting links to make followed content be processed by our application. So, if we follow any link on the page we will browsed the referenced page after being processed by our *reverse* application, and thus all content will be reversed.

4.2 *htspell*

Let's now take a glimpse at a simplified version of an application called *htspell*. This application checks for existing words in a dictionary. In our example we are using the GNU Aspell spell checker to check words, in parallel to this we are also keeping track of not found words in our application's state. We keep being intrusive and changing the content being browsed, by underlining words we can't found in the dictionary. We are using the *feedback* function to summarize words that aren't found for the user and the *feedbackLive* function to present some data to the user about the processed content, namely the number of processed words and the percentage of words not found. We can see some important pieces of this application being defined in Fig. 11.

Now let's give our *htspell* application a try. We visit a simple test case that prints a well know sentence with a couple of misspelled words. We can see the resulting page in Fig. 12. Notice the banner at the top, we can see the output of our *feedbackLive* function printing the processed number of words.

Our template uses a set of auxiliary functions to store state information in browser's *cookies*. Applications can take advantage of this feature by using the *estado* special hash. That's exactly what we are doing in line fifteen in Fig 11. We are keeping track of not found words in this special variable *estado*. We can then

```
1  (...)
2  proc(htspellFunction)
3  feedback(htspellFeedback)
4  livefeedback(htspellLive)
5  ##
6  (...)
7  sub htspellFunction {
8  (...)
9    foreach (@words) {
10     if ($speller->check($_)) {
11       $found++ and push @new, $_;
12     }
13     else {
14       $not_found++ and push @new, "<u>$_</u>";
15       $estado{$_}++
16     }
17   }
18 (...)
19 sub htspellFeedback {
20     h3("Words not found in the dictionary:").
21     small(ul(li([map{"$_ - $estado{$_}"}
22         sort {$estado{$b} <=> $estado{$a}} keys %estado])));
23 }
24 sub htspellLive {
25   "I checked $processed words, ".int($not_found/$processed*100).
26   "% were not found in the dictionary.";
27 }
28 (...)
```

Fig. 11. *htspell* program.

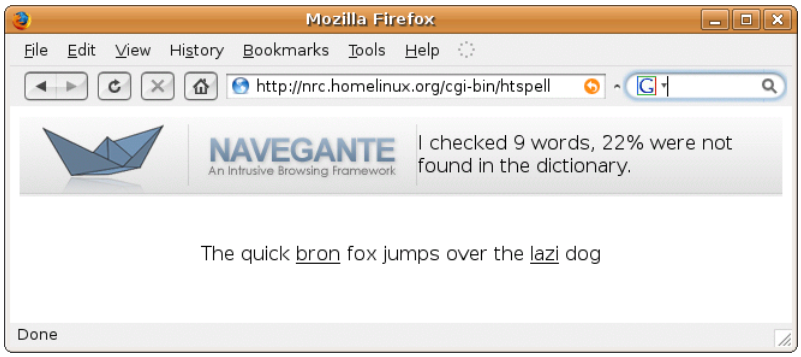

Fig. 12. *htspell* application.

later print this data to the user, using the function *htspellFeedback* as defined in line nineteen. We can verify this by following the *feedback* link, the small sailing boat, and verify the list of occurrences of not found words. This is illustrated in Fig. 13. Remember that our processing functions are nothing more than Perl code, so you can do whatever Perl allows you to do in your application, which is pretty much about everything.

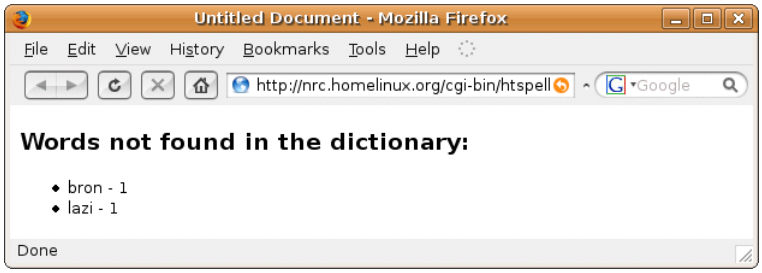

Fig. 13. *htspell* application *feedback* function.

5 Future Work

At the moment Navegante has some limitations:

- the address rewriting engine needs to be javascript aware. Nowadays many links are built using javascript in many obscure ways. This is often hard to detect, and rewriting those specific links can be almost impossible. But some improvements can be made, on this particular area, in order to lessen applications misbehavior.
- design and implement more skeleton templates, to be able to build different types of applications, and not only CGIs. A standalone service, that works like a traditional proxy, would be a nice addition.
- after adding new skeleton files, Navegante's DSL should be refactored, to allow the use of other contexts and options.
- allow the use of other programming languages in Navegante programs besides Perl. Make it possible for applications to fire up other interpreters or compilers to build or run the foreign code, in runtime or buildtime, depending on the language being used.
- implement other options to store data between requests. Currently, data can only be stored in browser's *cookies*.

6 Conclusion

Navegante is a framework aimed to craft applications that behave as proxies, but with an extra processing layer capability. This extra layer, allows to do any kind of transformation, on the fly, over the browsed content before being delivered to the client. We describe this approach, from the point of view of the application, as intrusive browsing. Intrusive browsing can be a very solid solution for many problems, since this approach can easily do complex processing, without adding much entropy to an already working scenario. Another advantage of this approach, in many situations, is that there is no need to change already working solutions to add some extra functionality.

Parallel to content processing, applications can be deployed with a diversified set of functions that can perform many tasks using different interfaces. These tasks can present new output by using the banner provided built-in with every application. Or, use the same banner to ask for user input, and merge it with state information. There's also the feedback option, that can process state information to display any kind of data at any given time. The subtlety of quitting the application, is also built-in in your application if you need it, it's just a way to finish the browsing session if the application needs to deliver any final conclusion regarding state or content information.

There are already some applications built based on this approach, which let us defend the usability and utility of this framework. These applications include:

- an online spell checker [9] that underlines each unknown word, and accumulates the list of unknown words for later reference;
- a terminology collector, that extracts from visited web pages words that are not common, and thus, are probably terminology terms;

Most of the times, this kind of applications are not easy or quick to implement, but with the right tools to aid development, complex tasks can be solved very easily. Also, some edge cases and common issues are already handled by some feature in the framework, like HTTP flow control, address rewriting, or being able to store state information. Navegante is still a work in progress but has already demonstrated to be a valuable tool to develop this specific gender of applications. Also, it is a very clean way to deploy applications that rely on other services, and in many cases even without those services being aware that their content is being used to feed other applications. In sum, there are unarguable facts that intrusive browsing is the natural solution for many complicated problems, and Navegante will be the natural way to implement them.

References

1. José João Almeida and Alberto Simões. Navegante: um proxy de ordem superior para navegação intrusiva. In José Carlos Ramalho, Alberto Simões, and João Correia Lopes, editors, *XATA 2006, Aplicações e Tecnologias Associadas*, pages 376–377, Portalegre, Fev. 2006. ESTGP. poster.
2. José João Almeida and Alberto Manuel Simões. Xml::dt. http://search.cpan.org/perldoc?XML::DT.
3. Francois Desarmenien. Parse::yapp. http://search.cpan.org/perldoc?Parse::Yapp.
4. Mark Jason Dominus. *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann, 2005.
5. António R. Fernandes, Alexandre Carvalho, J. João, and Alberto Simões. Transcoding for Web Accessibility for the Blind: Semantics from Structure. In *ElPub 2006 — Digital Spectrum: Integrating Technology and Culture*, Bansko, Bulgaria, June 2006.
6. Hans Rohnert. The proxy design pattern revisited. pages 105–118, 1996.
7. L.D. Stein. *Official guide to programming with CGI. pm*. Wiley New York, 1998.
8. W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network Programming, Vol. 1*. Pearson Education, 2003.
9. Rui Vilela. Webjspell, an online morphological analyser and spell checker. In *Procesamiento del Lenguaje Natural 39 - SEPLN*, pages 291–292, 2007.