Coarse-grained Delivery Units: from HTML Pages to XML *Leaflet*s*

Felipe Ibáñez, Oscar Díaz, Juan J. Rodríguez
Dpto. de Lenguajes y Sistemas Informáticos
University of the Basque Country, Apdo. 649 - 20080 San Sebastián (Spain)
e-mail: <jipibanf, jipdigao, jibrojij>@si.ehu.es

Abstract

Web applications traditionally follow a thin-browser architecture whereby all of the application control resides on the server, and the browser is only used for rendering purposes. This makes sense for B2C applications where no control is held on the browser configuration. However, this architecture can overload the server and slow down the promptness of the site. B2B applications, which are characterized by a establishment of tighter links among partners, can gain benefit from a thick-browser architecture where moving responsabilities to the browser can lead to a reduction in network traffic and an improvement in the promptness of the site. This paper explores this kind of architecture and proposes a coarse-grained delivery unit: the *leaflet*. A *leaflet* is a cohesive set of data from a browsing perspective (e.g. a catalogue). This work outlines the definition of a *leaflet*, presents a *leaflet* interpreter, and provides some time figures to assess the feasibility of this approach. The outcome is, that this architecture can pay off in a B2B setting where relationships tend to be more stable that in a B2C context. Thus, a business partner might be willing to install a plug-in if better efficiency can be obtained while accessing one of its partner sites in the future. The *leaflet* interpreter has been fully implemented and its use is illustrated by designing and delivering a conference site.

1. Introduction

Web applications traditionally follow a thin-browser architecture whereby most of the application functionality reside on the server, and the browser is only used for rendering purposes. The page becomes both the unit of delivery and the unit of presentation, and there is little control over the browser's configuration. Most e-commerce applications use this architecture as it does not make good business sense to eliminate a sector of customers just because they do not have sufficient browsing capabilities [2]. Data-intensive applications are also good candidates for using this architecture. In these applications, the raw data is not sent to the client but processed at the server, near to where the data is stored, and just the final outcome, the HTML page, is sent to the browser. On the other hand, with this kind of architecture a browser request will be required every time a new page is displayed. This could lead to an increase in network traffic, server bottleneck and low site promptness.

Based on this observation, this work advocates for a coarse-grained delivery unit: the *leaflet*. In terms of design, a *leaflet* is a purposeful unit of "logical pages" (e.g. a catalogue, a syllabus or a conference program) which makes sense to browse together. In terms of implementation, a *leaflet* is an XML document which can be processed to deliver a set of physical pages (i.e. HTML pages). Besides the attractiveness of abstracting away from such a low-level artifact as the HTML page, one of the main rationales for introducing the *leaflet* construct is to reduce network traffic. Although the notion of *leaflet* is orthogonal to where the *leaflet* is processed, this work is concerned with browser *leaflet* processing. More specifically, navigation dependencies, content rendering and some data flow are predominantly achieved at the browser. The *leaflet* becomes the unit of delivery.

This approach allows a whole bulk of pages to be generated from a single *leaflet* with a single connection to the server. This can account for a sensible reduction on the network traffic as connections to the server are limited to requests for an-

*This research was supported by both the Secretaría de Estado de Política Científica y Tecnológica of the Spanish Government under contract TIC 1999-1048-C02-02, and the Departamento de Educación, Universidades e Investigación of the Basque Government under contract UE2000-32.

other *leaflet* (e.g. invoking Web services). Hence, our approach moves to the browser some processes that are currently performed at the server. The down side of this approach is, that it increases the demands on the browser configuration. In particular, the browser has to be able to analyze and process *leaflet*s. In the current implementation, this is achieved by either downloading the interpreter from the server when the first *leaflet* is retrieved, or installing the interpreter as a plug-in on the browser. This requirement could make this approach unsuitable for B2C applications where users could be quite sporadic and can be discouraged by the perspective of having to install a plug-in. However, B2B applications offer a more promising setting. Firstly, the users are more computing-aware and installing a plug-in will not put then off. Secondly, B2B relationships tend to be more stable that B2C relationships and thus, a business partner might be willing to install the plug-in if better efficiency can be obtained while accessing one of its partner sites in the future. In any case, the wide spread use and support received by XML makes us feel confident about the feasibility of this approach. Indeed, Internet Explorer already allows for the presentation of XML documents to be processed at the browser by applying XSLT stylesheets.

This paper outlines the definition of a *leaflet* (section 2), presents the implementation of a plug-in to process *leaflet*s at the browser (section 3) and concludes with a discussion (section 4). Finally, conclusions are given. A conference web site is used as a running example throughout the paper.

2. The *leaflet* model

Broadly speaking, the XML technology allows to separate content from its presentation. Therefore, it realizes the well-know software strategy of "separation of concerns" which facilitates independent evolution of each of the concerns -in this case, how data is rendered. This concern is realized as an XSLT file that indicates how elements of the content document are transformed into an output format. If this format is XHTML, the XSLT file indicates the presentation of the input document. The binding is supported through the processing instruction element *"xml-stylesheet"*. The result of applying an XSLT to an XML document is normally a page fragment. This process can take place at either the server or the browser, the latter is currently only available for Internet Explorer.

The *leaflet* model extends this approach by incorporating a new concern: how the document is navigated. Not only does a *leaflet* indicate how it is rendered but also how its content is traversed. So far, if a document consists of several pages, the navigation logic is hard-coded and distributed in distinct scripts which hinders development and maintenance of the site. The notion of *leaflet* strives to abstract away from HTML pages by conceiving a *leaflet* as a purposeful unit of content (e.g. a catalogue, a syllabus or a conference program), regardless of how this content is finally distributed among distinct pages. Unlike the XML model, the processing of a *leaflet* can produce more that one page. In terms of design, a *leaflet* can be thought of as a realization of the model-view-controller pattern, where the model is the content document, the view is the presentation document and the controller is supported by the navigation document. Consequently a *leaflet*, can provide a useful abstraction for partitioning sets of pages into purposeful units of browsing.

Figure 1 depicts the *leaflet* for a conference example. Its definition is similar to other XML document. Elements are introduced to indicate the aims of the conference, the organizing and program committee, the call-for-papers, the list of accepted papers and so on [1]. Two processing instructions are introduced to indicate how this *leaflet* should be presented and navigated, *leaflet-presentation_document* and *leaflet-navigation_document*.

As a *leaflet* potentially collects a larger number of data than a traditional XML document, it is expected that some data is obtained from external sources. However, some data might be provided right away as part of the *leaflet* definition (i.e. *proprietary elements*). For instance, the logo, the venue or the list of topics of a conference can be provided directly within the *leaflet*. However, the list of accepted papers is a good candidate to be stored in a database, while the weather forecast for the region where the conference takes place can be supported as a Web service. This are examples of *derived elements*, which are elements whose content is not provided directly but obtained from external sources. To abstract away from the concrete external sources, we assume that any source is wrapped by a Web service[2].The content of a de-

[1]The W3C consortium is promoting distinct standards for document schema definition. This work uses XML-Schema [8]. An XML-Schema document is an XML document that describes the structure definition of a particular class of documents, i.e. addressing aspects such as which elements can occur and how can be nested in the XML document that conforms to the schema. Space limitation prevent us from showing the conference schema definition.

[2]Although some database management systems allow to write SQL statements that return a flat XML document, oper-

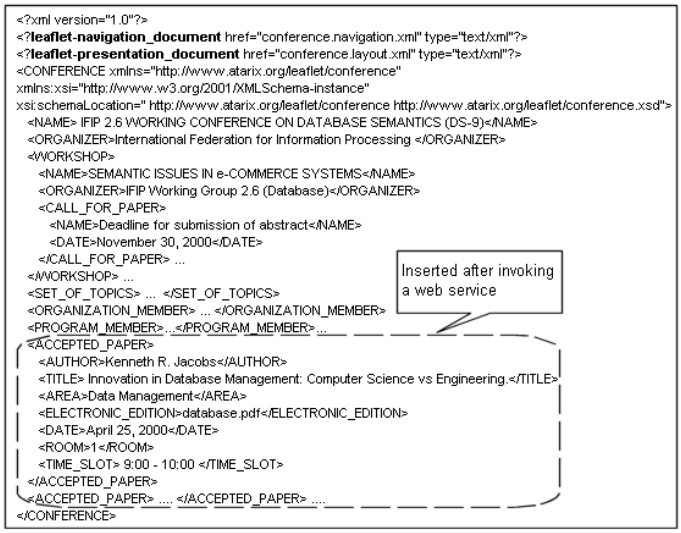

Figure 1: The conference *leaflet*.

rived element is the XML document which the associated Web services returns. Figure 1 gives an example for the <*ACCEPTED_PAPER*> element.

For the purpose of this paper, we only outline how navigation is specified. Refer to [3] for further details.

2.1. The navigation model

The navigation model addresses how this document can be traversed through links. This model is realized by the navigation document (see figure 2). As an example, consider the link that goes from a */CONFERENCE/WORKSHOP* element to its related *CALL_FOR_PAPERs* elements. This is expressed as follows:

<LINK title= "Call For Papers"
from= "/CONFERENCE/WORKSHOP"
to= "CALL_FOR_PAPER"> ... </LINK>

LINK is an XML element of the navigation vocabulary. This element has a set of attributes which describe: (1) the label of the link when rendered on the screen (the *title* attribute); (2) the origin of the link (the *from* attribute) that indicates when the link is available (in this case the link is available when the *WORKSHOP* element is rendered); and (3) the destination of the link (the *to* attribute) which states the element to be rendered when the link is traversed. Besides the origin and destination, a *LINK* is characterized by a navigation and a coupling mode. Each of these aspects will be addressed in the next paragraphs.

Anchor identification. The distinct element tags of the *leaflet* are conceived as potential anchors from which to define origins and destinations. Elements within the *leaflet* are addressed using the W3C standard XPath notation[5]. XPath conceives an XML document as a tree of nodes, and follows a notation similar to the UNIX directory paths to address each node within the document (a *location path* using XML parlance). We have extended this notation in distinct ways to accommodate some navigation requirements.

The startup entries are indicated by the keyword "home". In the conference example, the *CONFERENCE* outline is presented once connected to the site. No-contextual links provide entry points which can be accessed at any point during the navigation. They are denoted by an asterisk. In our example, the *ORGANIZATION*_MEMBER, the *PROGRAM*_MEMBER or the *SET_OF_TOPICS* data are always accessible within the conference scope. By contrast, *WORKSHOP* data can only be accessed from the conference outline, which in turn, leads the way to the *CALL_FOR_PAPER*s data.

Navigation mode. Navigation proceeds from a node to those destination nodes of the chosen link. If a single destination node is available, navigation is straightforward. However, this is rarely the case, and often navigation is one-to-many. The navigation mode indicates how to proceed in this case.

As an example, a *CONFERENCE* element

ations of Web services permit to encapsulate how the returning XML document (of any structure) is obtained. In this way, our content document is isolated from changes on the database schema. The association between the element and the web service is specified in the XML-Schema. This association defines which operation of the web service retrieves the external data.

```
<leaflet:LINKS xmlns:leaflet="http://www.atarix.org/leaflet/navigation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.atarix.org/leaflet/navigation http://www.atarix.org/leafelt/navigation.xsd">
  <!-- startup link -->
  <leaflet:LINK title="Conference" from="home" to="/CONFERENCE">
    <leaflet:CONTENT order="1" couplingMode="embedOnLoad"/>
  </leaflet:LINK>
  <!-- contextual links-->
  <leaflet:LINK title="Workshops" from="/CONFERENCE" to="WORKSHOP">
    <leaflet:CONTENT order="1" couplingMode="embedReplacingOnRequest"/>
  </leaflet:LINK>
  <leaflet:LINK title="Call For Papers" from="/CONFERENCE/WORKSHOP" to="CALL_FOR_PAPER">
    <leaflet:CONTENT order="1" couplingMode="embedOnLoad"/>
  </leaflet:LINK>
(a)
  <leaflet:LINK title="Papers by date, time-slot and room" from="/CONFERENCE"  to="ACCEPTED_PAPER">
    <leaflet:INDEX order="1" title="Index by date and time slot:" couplingMode="newOnRequest">
      <leaflet:INDEX_PROPERTY propertyName="DATE"/>
      <leaflet:INDEX_PROPERTY propertyName="TIME_SLOT"/>
    </leaflet:INDEX>
    <leaflet:INDEX order="2" title="Index by room" couplingMode=" embedReplacingOnRequest ">
      <leaflet:INDEX_PROPERTY propertyName="ROOM"/>
    </leaflet:INDEX>
    <leaflet:CONTENT order="3" couplingMode="embedReplacingOnRequest"/>
  </leaflet:LINK>
  <leaflet:LINK title="Papers by author" from="/CONFERENCE" to="ACCEPTED_PAPER">
    <leaflet:FILTER order="1" title="Filter papers by author" couplingMode="newOnRequest">
      <leaflet:SEARCH_PROPERTY title="Author" propertyName="AUTHOR" predicate="belongsTo"/>
    </leaflet:FILTER>
    <leaflet:CONTENT order="2" couplingMode="embedReplacingOnRequest"/>
  </leaflet:LINK>
<!-- No contextual links -->
  <leaflet:LINK title="Conference Organization" from="*" to="/CONFERENCE/ORGANIZING_MEMBER">
    <leaflet:CONTENT order="1" couplingMode="replaceAllOnRequest"/>
  </leaflet:LINK>
</leaflet:LINKS>
```

Figure 2: The navigation document. This document is bound to the leaftet by means of the *<?leaflet-navigation_document ...?>* processing instruction.

can include a set of *ACCEPTED_PAPER* sub-elements. The question is, when traversing a link from the *CONFERENCE* element to its *ACCEPTED_PAPER* sub-elements, should all of them be processed at once, or is it preferably to browse them one by one? In this case, the designer is confronted with the decision of how the set of papers is traversed. The chosen navigation mode is described by means of *LINK* sub-elements. Based on the WebML modeling language [1], our model supports indexes, filters, and scrolls. A link can sequence some of these constructors to build up an aggregate navigation mode.

An index provides a shortcut to reach the desired destination nodes. An example, is the way in which papers are obtained through an index hierarchy as shown in figure 3: first, a *date+timeSlot* index is presented from where the user selects a value; then, the system dynamically generates a second index on the rooms used on the chosen *date+timeSlot* value; finally, once a *room* has been selected, the system retrieves the data about the paper to be presented in this room.

Index description is achieved through the *INDEX* element. For instance, the index hierarchy shown in figure 3 has been obtained from the description in 2a.

This link specifies how to reach the set of *ACCEPTED_PAPER* nodes (the *to* attribute) from a *CONFERENCE* node (the *from* attribute). The *INDEX_PROPERTY* sub-element indicates which property of the destination element (i.e. *ACCEPTED_PAPER*) serves as an index. The "property" refers either an attribute or a sub-element. In the previous example, the system offers a first index based on *DATE* and *TIME_SLOT*, both are sub-elements of *ACCEPTED_PAPER*. As the number of papers is still large within each *date+timeSlot* value, a second index arranges papers according to the *ROOM* they are presented at. Notice that the order in which the indexes are arranged is important.

Coupling mode. For example, when visiting the CONFERENCE element of the content document an index can be used in order to visualizing the individual papers presented at this conference. The question is whether both the index and the paper should be displayed together with the conference content (*embed*), substitute the conference content (*replace*) or be rendered in a separate window (*new*)? Furthermore, should these questions be posed separately for the index and the paper node so that the conference node, the index, and the paper node could be coupled differently? Moreover, should this navigation option occur on request or take place automatically on load? The coupling mode addresses where and when nodes of the content document, indexes, filters and scrolls are coupled during link traversal[3]. The "where" question admits three answers: *embed*, *replace* and

[3]Similar concerns also appear in the W3C XML Linking Language (XLink) proposal [7].

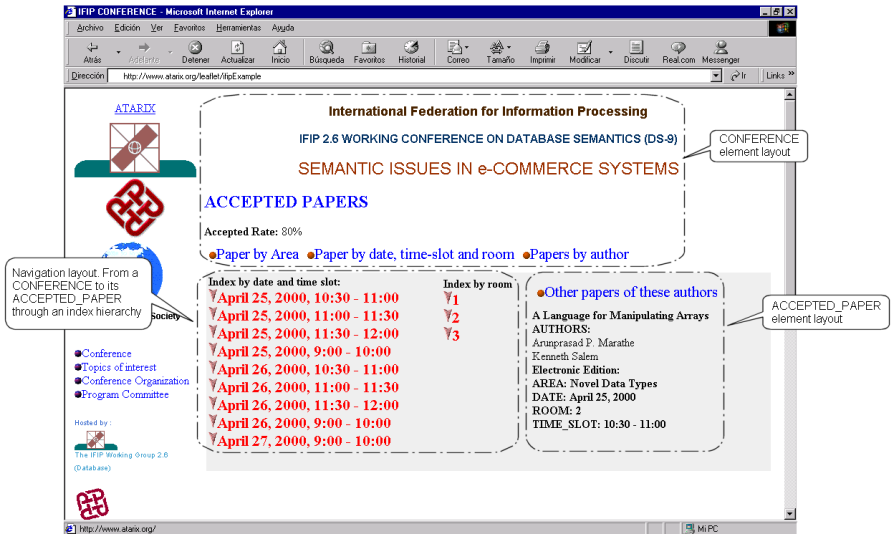

Figure 3: Navigation from *CONFERENCE* to ACCEPTED_PAPER: an index hierarchy along accepted papers.

new. The "when" can be answered either with *onLoad* or *onRequest*.

If distinct modes are used in succession (e.g. an index hierarchy, or an index followed by a filter) the coupling mode applies to the previous mode. For instance, the page from the conference application shown in figure 3 delivers the origin element (i.e. a *CONFERENCE* element) in the same page than both the indexes and the destination elements (i.e. an *ACCEPTED_PAPER* element). Alternatively, both the indexes and the selected *ACCEPTED_PAPER*s can be presented separately in a different page than conference data. In this case, the navigation mode remains the same while the coupling modes are different. Therefore, the designer is free to decide how tightly she wants to distribute the content. Notice that any traversed link implies to generate a new page.

3. A thick-browser architecture: the *leaflet* as the unit of delivery

This paper's main concern is to gain some insight into a thick-browser architecture for Web applications. Unlike more traditional approaches, a thick-browser architecture moves to the browser some of the processing that is currently conducted at the server. Specifically, in this approach the additional processes performed by the browser are page generation and navigation logic. The potential advantages are three-fold: reduction in the network traffic, enhancement of the site promptness, and reduction in the server load. At the same time, the zero-deployment advantage is retained, i.e. code is centralized at the served and instantaneously deployed on the browser.

This architecture implies that the *leaflet* is the unit of delivery. A *leaflet* unit is brought to the browser where a whole bulk of pages are generated locally. In this way, requests to the server are reduced to (1) retrieving the next *leaflet* (2) retrieving some resources (e.g. a gif) or (3) invoking a Web service which is attached to a derived element.

3.1. Implementation

The *leaflet* run-time architecture is shown in figure 4[4]. A *leaflet* container is divided into two frames: a visualization frame which supports a display area, and a hidden *leaflet* interpreter frame that holds the *leaflet* run-time. Specifically, this run-time includes: (1) the *LeafletManager* module which is responsible for loading the *leaflet* and its attached resources, (2) the *NavigationController* which maintains the current context and drives the navigation, (3) the *PresentationController* which generates the HTML pages and (4) the *DeriveElementResolver* which invokes a web service in order to retrieve the external data.

Figure 5 depicts how the previous modules interact during the loading of a *leaflet*. The browser requires an HTML initialization page which acts as the container of the *leaflet*. This page holds the

[4]The diagram follows the UML notation proposed in [2]. This notation allows to model the implementation elements that compose a Web application by using UML stereotypes.

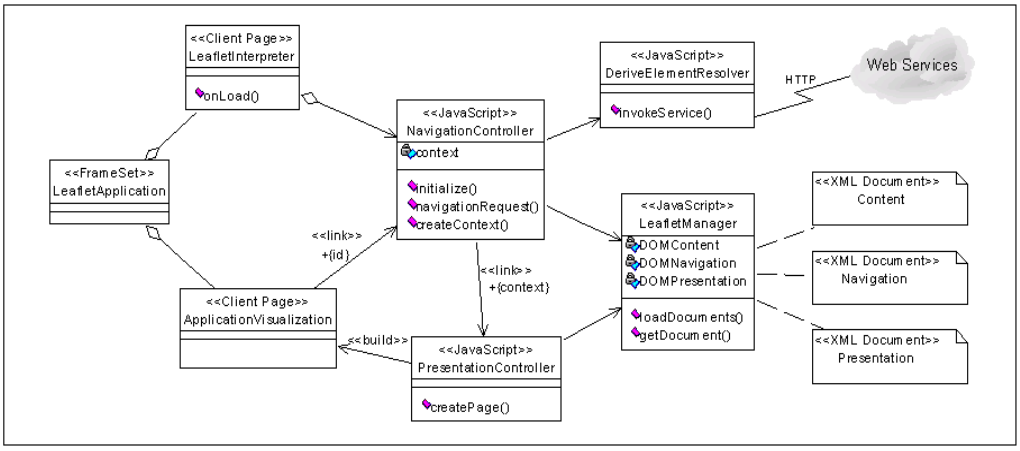

Figure 4: The *leaflet* run-time architecture.

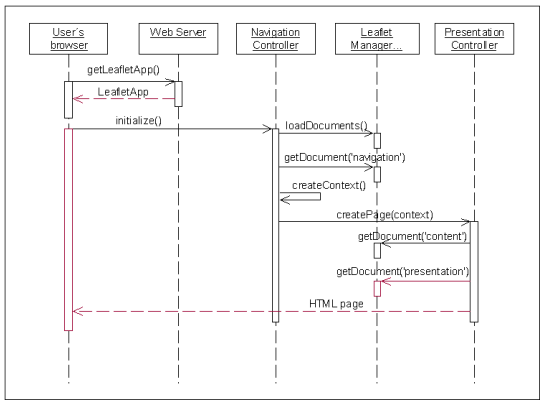

Figure 5: *Leaflet* processing. Loading a *leaflet*.

four modules of the run-time. Next, the *initialize* method of the *NavigationController* is invoked, which renders the "home" page. To this end, the *NavigationController* requests the *LeafletManager* to load all the documents that constitute the *leaflet* (i.e. the content, presentation and navigation document) and extracts from the navigation document the first links to traverse (i.e those having *"home"* as their *from* attribute, see section 2.1.). The *NavigationController* proceeds by traversing the link, updating the current execution context and requesting the rendering of the new page to the *PresentationController*. The controller in turn requests the content and presentation documents to the *LeafletManager*. As a result, an HTML page is generated and rendered.

The Achilles' heel of this architecture is a more demanding browser configuration. So far, the *leaflet* run-time requires:

1. JavaScript libraries(or a plug-in) to interpret a *leaflet* document for navigation control and page generation.

2. A parser that processes the DOM API[4], XSLT 1.0[6] and XML Schema[8][5] .

3.2. Some figures

Rendering a page involves: (1) a request for the page; (2) the generation of the page at the server (if dynamically obtained); and (3) the delivery of the page.

The former only accounts for less than one kilobyte and thus, it can be ignored. The other aspects are influenced by the bandwidth of the network, the size of the page and the amount of traffic both at the web server and the data server. An estimation for the required loading time can be obtained by using the following expression:

$$(size(Page)/bandwidth) + (size(Page)/Cs)$$

where *Cs* is a constant which reflects the server throughput (KB per second). We estimate that it takes 0.06 seconds for a web server to generate one kilobyte of HTML (of course, this is highly variable as it depends for instance on the load currently placed on the servers and the complexity of the SQL query). This gives a value for *Cs* of 15KB/sec.

On the other hand, rendering a *leaflet* involves: (1) a request for the *leaflet;* (2) the delivery of the *leaflet* run-time; (3) the delivery of the *leaflet*; and (4) the processing of the *leaflet* at the browser.

[5]This work uses Microsoft XML parser 4.0. So far, Internet Explorer 6.0 is the only browser that provides this DOM API and both XSLT and XML Schema processors. However, Netscape will also provide XML support in the next release.

Table 1: Size and processing cost for each page generated for the conference example.

page size (KB)	processing cost(sec.)
10,38	0,59
5,66	0,81
7,25	0,67
8,78	0,96
8,77	1,14
8,80	0,96
8,73	0,95
8,76	0,95
7,44	0,70
8,02	0,76
8,91	1,17
8,92	1,18
8,91	1,08
9,20	1,26
10,15	1,42
10,73	1,59

If we ignore the first parameter, a possible formula reflecting these aspects is

$$(size(runtime)/bandwidth)+ \\ (size(Leaflet)/bandwidth)+ \\ (size(Leaflet)/Cb)$$

where *Cb* is a constant which reflects the *leaflet* run-time throughput (KB per second). We estimate that it takes 0.11 seconds for the run-time to generate one kilobyte of HTML. This gives a value for *Cb* of 9KB/sec.

However, this formula calculates the cost of processing a whole document from which several pages are generated. For the comparison to be done in equal terms, the previous formula should be expressed in terms of pages:

$$(size(runtime)/bandwidth)+ \\ (size(Leaflet)/bandwidth)+ \\ \sum(size(GeneratedPage_i)/Cb)$$

Notice that the cost between generating one or several pages only differs on the third term of the formula. In other words, for the *leaflet* to be worth processing at the browser, several pages should be rendered. Otherwise, the cost of bringing both the run-time and the *leaflet* to the browser does not pay off. This is the reason why a thick-browser architecture only makes sense for navigation-intensive applications where a whole bulk of content need to be traversed in distinct ways. Furthermore, the data-intensive is at the very heart of the *leaflet* notion: a data assembly that makes sense to be browsed as a unit.

The question is what is the minimum number of pages to be rendered for the thick-browser architecture to payoff. In other words what is the value

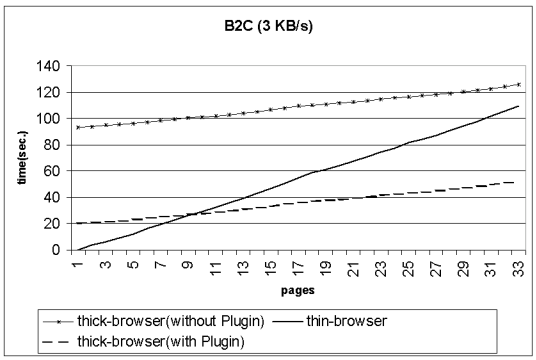

Figure 6: Time figures for a bandwidth of 3Kb/sec.

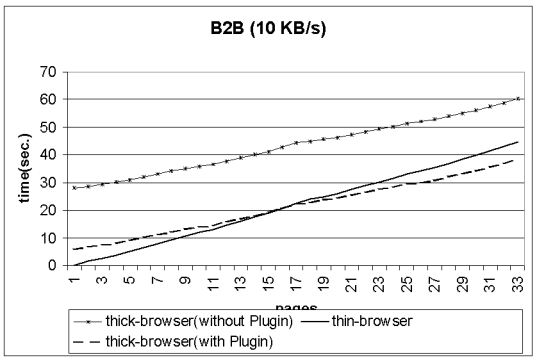

Figure 7: Time figures for a bandwidth of 10Kb/sec.

of *'numPag'* in:

$$(size(runtime/bandwidth)+ \\ (size(Leaflet)/bandwidth)+ \\ numPag \star (avg(size(Page_i))/Cb) \doteq \\ numPag \star [(avg(size(Page))/bandwidth)+ \\ (avg(size(Page))/Cs)]$$

For example, the conference *leaflet* has a size of 60K. This includes the content document, the navigation document and the presentation document. Consider the conference *leaflet* generates 32 pages; the sizes and the processing costs for the first sixteen pages are shown in table 1. As for the run-time, its current size is 220K.

Consider now three possible scenarios:

1. a thin-browser architecture where the *leaflet* is processed at the server,
2. a thick-browser architecture with the run-time is already installed as a plug-in, and the *leaflet* is processed at the browser
3. a thick-browser architecture where both the run-time and the *leaflet* are brought to the browser

Figure 6 and 7 compares this three options with a bandwitch of 3Kb/sec and 10Kb/sec, respectively.

4. Discussion

Although the *leaflet* run-time has not yet been tuned for efficiency optimization, we can already provide some insights from the previous figures:

First. The size of the run-time can make the thick-browser architecture inviable unless the run-time can be installed as a plug-in. This download penalty is payed just once as the plug-in caches the run-time the first time it is downloaded. Hence, this architecture does not fit B2C applications where users can be quite sporadic and could be discouraged by the perspective of having to install a plug-in. However, B2B applications offer a more promising setting. Firstly, the users are more computing-aware, and installing a plug-in will not put them off. Secondly, B2B relationships tend to be more stable that B2C relationships and thus, a business partner would be more willing to install the plug-in if better efficiency can be obtained while accessing one of its partner sites in the future. Another option is to provide a kind of run-time-lite which reduces the size of the system at the expense of reducing its functionality.

Second. The thick-browser architecture only pays off for navigation-intensive applications. As shown in figure 6, this architecture begins to be faster than the traditional thin-browser architecture if 10 or more pages are rendered. For less than 10 pages, the download cost of the whole document does not pay off.

Third. The worse the network, the more interesting is the thick-browser architecture. When comparing figures 6 and 7, it can be concluded that the minimum number of pages have to be rendered for the thick-browser architecture to pay off decreases as the bandwidth deteriorates.

5. Conclusions

This paper investigates a thick-browser architecture whereby more functionality is moved to the browser realm. Specifically, the browser is in charge of *leaflet* processing which involves managing both navigation control and page rendering. This is aligned with current XML technology where some browsers have been already enhanced to process XML documents. Notice however that the notion of *leaflet* it is more a design issue, and it does not preclude any architectural option.

This approach accounts for a reduction in the network traffic, an enhancement on the site's promptness, and a reduction of the server load. However, a downside of this approach is an increase of the demands on the browser configuration. This situation probably prevents the *leaflet* notion from being used for e-commerce applications targeted at final customers. However, B2B or e-procurement projects where more control over the client configuration is possible, could benefit from a thick-browser architecture. The paper has shown that even non-optimized implementations can account for some performance gains for "navigation-intensive" applications.

We are currently working on a hybrid architecture where both the thick- and the thin-architecture approaches can coexist in the very same application. The application is split into a set of *leaflet*s. Where the *leaflet* is processed (i.e. the server or the browser) depending on the expected navigation pattern and how much data is derived from external sources. "Navigation-intensive" *leaflet*s are moved to the browser while "data-intensive" *leaflet*s are preferably kept at the server. These issues will be the focus of future research.

References

[1] S. Ceri, Piero Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[2] J. Conallen. *Building Web Applications with UML*. Addison-Wesley, 2000.

[3] O. Diaz, F. Ibañez, and J. Iturrioz. A model-based approach to web-application development. In *Proceedings of 9th IFIP 2.6 Working Conference on Database Semantics (DS-9)*, pages 313–339, 2001.

[4] W3C. Document Object Model at http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, 1998.

[5] W3C. XML Path Language (XPath) Version 1.0 at http://www.w3.org/TR/xpath.html, 1999.

[6] W3C. XSL Transformations (XSLT) Version 1.0, 1999. at http://www.w3.org/TR/xslt/.

[7] W3C. XML Linking Language (XLinking) Version 1.0, 2001. at http://www.w3.org/TR/xlink/.

[8] W3C. XML Schema Part 1:Structures, 2001. at http://www.w3.org/TR/xmlschema-1/.